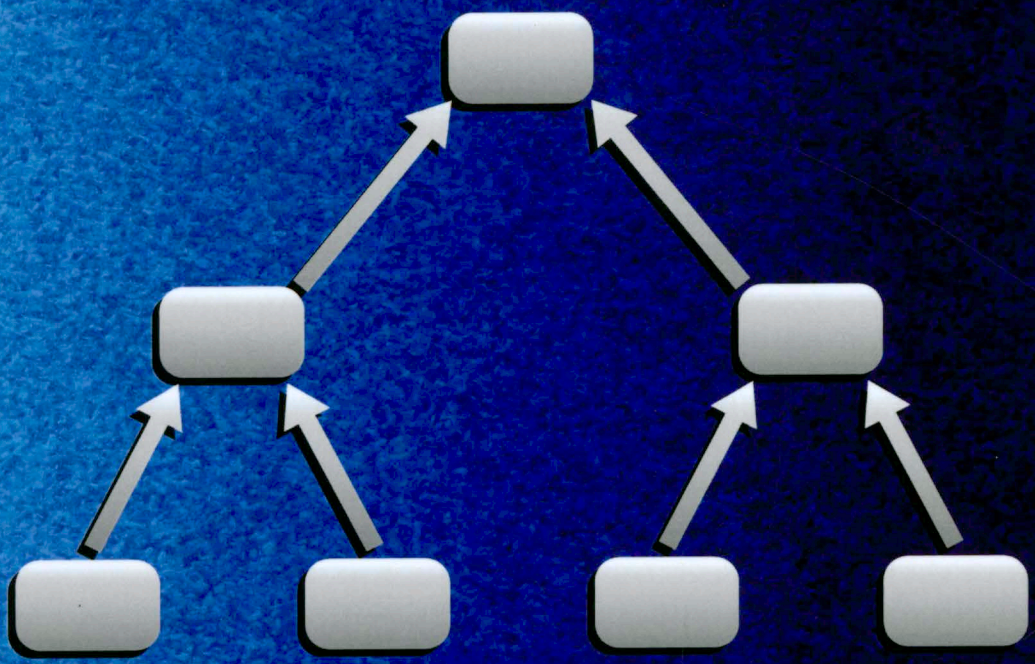
Wojciech Bożejko

A New Class of Parallel Scheduling Algorithms



Oficyna Wydawnicza Politechniki Wrocławskiej

Wojciech Bożejko

A New Class of Parallel Scheduling Algorithms

Oficyna Wydawnicza Politechniki Wrocławskiej
Wrocław 2010

Reviewers
Zbigniew BANASZAK
Jerzy JÓZEFCZYK

Proof-reading
Halina MARCINIAK

Cover design
Marcin ZAWADZKI

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
http://www.oficyna.pwr.wroc.pl
e-mail: oficwyd@pwr.wroc.pl

ISBN 978-83-7493-564-7

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 1098/2010.

Contents

Preface

The main issue discussed in this book is concerned with solving job scheduling problems in parallel calculating environments, such as multiprocessor computers, clusters or distributed calculation nodes in networks, by applying algorithms which use various parallelization technologies starting from multiple calculation threads (multithread technique) up to distributed calculation processes. Strongly sequential character of the scheduling algorithms is considered to be the main obstacle in designing sufficiently effective parallel algorithms. On the one hand, up till now sequential algorithms exhausted the possibilities of significant growth in the power of solution methods. On the other hand, parallel computations offer essential advantages of solving difficult problems of combinatorial optimization, pushing towards theory, methodology and engineering of solution algorithms.

The book is divided into a 'state-of-the-art' part followed by two original parts, concerning single-walk and multiple-walk multiple-threads optimization algorithms applied to solve scheduling problems. At first, an introductional part is placed, including a methodology for parallelization of metaheuristics, introduction to scheduling issues, scheduling problems, classical and the most recent discrete optimization tendencies. This constitutes the 'state-of-the-art', worked out for author's parallel computing and prepared on the basis of the extensive bibliography.

The next two parts make up the core of the book and deal with the author's own novel results. The division into two parts (single- and multiple-walk parallelization) is adjusted to structurally different approaches applied to design parallel algorithms. There are plenty of genuine single-thread search methods proposed in Part II which are designed for homogeneous parallel systems. These methods take into consideration a variety of techniques of parallel algorithm designing process as well as different necessities of modern algorithms of discrete optimization (analysis of the cost function determination, analysis of theoretical speedup). Theoretical estimations of the properties of particular algorithms are derived; a comparative analysis of advantages resulting from application of different approaches has been made.

The third part of this book is concerned with multithread search dedicated for homogeneous and heterogeneous multiprocessor systems, such as mainframe computers, clusters, diffuse systems connected by networks. Some parallel variants of the most promising current methods of combinatorial optimization (tabu search, simulated annealing, genetic methods) have been designed and examined experimentally in the application to selected scheduling problems. Different techniques of computation threads realization and their communication have been discussed, especially for migration models (so-called island models) in evolution methods. A superlinear (orthodox) speedup effect has been observed. In the case

of parallel variants of branch and bound scheme, dedicated for homogeneous and heterogeneous parallel systems, this type of algorithms has been designed and examined for selected class of scheduling problems. In particular chapters not only the parallelization benefit was shown, but (first of all) the methodology for designing parallel algorithms was described on examples of optimization problems. Complex scheduling problems (job shops, flexible and hybrid problems), for which even a feasible solution construction constitutes a hard computation problem, were chosen to be a case study for showing the parallelization process.

The book contains a wealth of information for a wide body of readers, including advanced students, researchers and professionals working in the field of discrete optimization and management. A new methodology of solving strongly NP-hard real-world job scheduling problems is presented here. It allows us to design very efficient and fast approximate and exact algorithms for solving a wide class of discrete optimization problems, not only scheduling problems. Efficiency of the present research has been proved by comprehensive computational experiments conducted in parallel processing environments such as supercomputers, clusters of workstations, multi-core CPUs and GPUs.

The author would like to thank the Wrocław Center of Networking and Supercomputing (WCNS, [266]) for enabling numerical experiments in multiprocessor environment.

Scope

Chapter 1 provides theoretical and practical basis of parallel computations. A methodology for the parallelization of known sequential algorithms is discussed in Chapter 2. Chapter 3 contains a short introduction to job scheduling problems with the extension of some special properties of problems which are generated by the practice for their use in designing parallel algorithms.

Chapters 4 through 7 concern the methodology of designing parallel algorithms for single-walk calculations. The contents of particular chapters is presented below. A methodology for transferring huge neighborhood search technologies in the local search methods into the parallel computing environment is presented in Chapter 4. The methodology is illustrated by examples of several single-machine scheduling problems met in practice. In Chapter 5 there are the new approaches to efficient parallel algorithm design shown for a single solution cost function value determination. The approach is presented on the case of a job shop scheduling problem, enjoying a great interest to practitioners of operations research. Chapter 6 presents the new integrated approaches to the neighborhood structure design and to the methodology of its searching from the point of view of the efficient parallel computing environment usage. This approach is described on the special case of so-called hybrid job shop scheduling problem (scheduling and resources allocation) constituting a base of FMS systems functioning. Chapter 7 provides the new theoretical results in single-walk exploration, complementing the state of the field of knowledge.

Chapters 8 through 14 concern a methodology for designing multiple-walk parallel algorithms. Chapter 8 presents the methodology of parallel algorithm designing based on memetic approach (Lamarck and Baldwin evolution theory) making use of specific properties of the problem and distributed island model. This approach is illustrated by an example of the single machine scheduling problem with E/T penalties. A new genuine population-based approach is proposed in Chapter 9 on the example of the single machine scheduling problem with setup times, modelling single bottleneck industrial nest functioning. In Chapter 10 there is presented a methodology for transferring sequential B&B algorithm into its parallel variant as an exact method and cut B&B as an approximate method. Load balancing of processors has been discussed. This approach has been described on the example of the single machine total weighted tardiness problem. Chapter 11 proposes the methodology of parallel simulated annealing algorithms design on the example of flow shop scheduling problem with the objective of minimizing the makespan as well as with the sum of job completion times objective. An unprecedented methodology of threads cooperating creation has been proposed. A methodology for solving the flow shop problem by using scatter search algorithm is presented in Chapter 12. The proposed parallelization methodology

constitutes a general approach, which increases the quality of obtained solutions keeping comparable costs of computations. A superlinear speedup is observed in cooperative model of parallelism. Chapter 13 presents a multiple-walk parallelization of the island model based genetic algorithm in application to the flow shop scheduling problem. The multi-step crossover fusion operator (MSXF) is used as an inter-island communication method. As compared to the sequential algorithm, parallelization enhances the quality of solutions obtained. Computer experiments show, that the parallel algorithm is considerably more efficient with relation to the sequential algorithm. In Chapter 14 there are two new double-level metaheuristic optimization algorithms applied to solve the flexible job shop problem (FJSP) with makespan criterion. Algorithms proposed in this chapter have two major modules: the machine selection module and the operation scheduling module On each level a parallel metaheuristic algorithm is used, therefore this method is called Meta^{2}Heuristic.

In Chapter 15 there is proposed the new methodology of parallel tabu search approach created with the use of cooperation between concurrently running searching threads. The approach is shown on the example of the flow shop scheduling problem and applied to solved a real-world optimization problem of roadwork scheduling. Some special properties of the problem considered (so-called blocks on the critical path) connected with representatives are used for calculation diversification among searching threads.

List of symbols

A_p	–	a parallel algorithm executed on p processors
$\mathcal{B}$	–	a partition into blocks
B_k	–	the k-th block
$c_{A_p,M}(p)$	–	the cost of solving a problem by using an algorithm A_p in a p-processor parallel machine M
$\mathcal{C}_i$	–	a term of a job i execution finishing
C_j	–	a term of an operation j execution finishing
$C_{\max}$	–	makespan (goal function)
C_{sum}	–	the total execution time (goal function)
d_i	–	due date of the job i execution finishing
d	–	number of layers
Δ	–	an upper bound of the goal function value
$\mathcal{E}_i$	–	earliness of the job i execution finishing
$\eta_{A_p,M}(p)$	–	efficiency of algorithm A_p solving problem P on the machine M making use of p processors
$\mathcal{F}_i$	–	a time of flow of the job i through the system
F	–	the goal (cost) function
$f_i(t)$	–	non-decreasing cost function connected with job i execution finishing in time t
Φ°	–	feasible solution
Φ_n	–	the set of all permutations of an n-element set
G	–	granularity
$\mathcal{J}$	–	a set of jobs
$\mathcal{L}_i$	–	non-timeliness of the job i execution finishing
M	–	a set of machine types
M_j	–	a sequence of machine subsets which define alternative methods of operation execution
m	–	number of machines
$\mathcal{N}(\pi)$	–	neighborhood of solution π
n	–	number of jobs
$\mathcal{O}$	–	a set of operations
o	–	number of operations
o_i	–	number of operations in the job i
Θ	–	feasible solution
p	–	number of processors
p_{ij}	–	a time of execution of an operation j by the i-th method

π	–	permutation
π^*	–	the best known permutation
$\mathcal{Q}$	–	machine workload (an assignment of operations to machines)
r_i	–	the earliest possible term of the job i execution beginning
$S_{A_p,M}(p)$	–	the speedup of algorithm A_p solving the problem P on machine M making use of p processors
S_j	–	a term of an operation j execution beginning
$\mathcal{S}_i$	–	a term of the job i execution beginning
T_{par}	–	the computations time of parallel algorithm
T_{seq}	–	the computations time of sequential algorithm
$T_{A_p,M}(p)$	–	the time of calculations of the algorithm A_p solving the problem P on machine M making use of p processors
T_{A_s}	–	the time of calculations of the sequential algorithm A_s
$\mathcal{T}_i$	–	tardiness of the job i execution finishing
T_c	–	computations time
T_t	–	communication time
$\mathcal{U}_i$	–	unitary tardiness of the job i
U	–	a set of conjunctive arcs
V	–	a set of disjunctive arcs
v_j	–	a method of the operation execution
x_{best}	–	the best known solution

List of abbreviations

ACO	–	Ant Colony Optimization, the method
APRD	–	Average Percentage Relative Deviation
B&B	–	Branch and Bound, the method
CC-NUMA	–	Cache Coherent Non-Uniform Memory Access
COW	–	Cluster Of Workstations
CRCW	–	Concurrent Read Concurrent Write, a kind of PRAM
CREW	–	Concurrent Read Exclusive Write, a kind of PRAM
CUDA	–	Compute Unified Device Architecture, a parallel programming library for GPUs
DM	–	distributed memory
EDA	–	Estimated Distribution Algorithms, the method
EREW	–	Exclusive Read Exclusive Write, a kind of PRAM
ERCW	–	Exclusive Read Concurrent Write, a kind of PRAM
ES	–	Evolution Strategies, the method
E/T	–	Earliness/Tardiness
FJSP	–	Flexible Job Shop Problem
GA	–	Genetic Algorithm, the method
GP	–	Genetic Programming, the method
GPGPU	–	General Purpose Graphic Processing Unit
GPU	–	Graphic Processing Unit
GRASP	–	Greedy Randomized Adaptive Search Procedure, the method
LB	–	Lower Bound, of the goal function
LM	–	Long-term Memory, in the Tabu Search algorithm
MA	–	Memetic Algorithm, the method
MIMD	–	Multiple Instruction set, Multiple Data set, a model of parallel architecture
MISD	–	Multiple Instruction set, Single Data set, a model of parallel architecture
MPI	–	Message Passing Interface, the parallel programming library
MPP	–	Massively Parallel Processor
NC-NUMA	–	Non-cache Coherent Non-Uniform Memory Access
NEH	–	Navaz, Enscore and Ham, an algorithm
NUMA	–	Non-Uniform Memory Access
ParPBM	–	Parallel Population-Based Metaheuristic, the method
ParSS	–	Parallel Scatter Search, the method

PATS	–	Parallel Asynchronous Tabu Search, the method
pSA	–	Parallel Simulated Annealing, the method
PSTS	–	Parallel Synchronous Tabu Search, the method
PBM	–	Population-Based Metaheuristic
PRAM	–	Parallel Random Access Machine, a theoretical model of parallel computations
PRD	–	percentage relative deviation
PVM	–	Parallel Virtual Machine, the parallel programming library
SA	–	Simulated Annealing, the method
SGI	–	Silicon Graphics
SIMD	–	Single Instruction set, Multiple Data set, a model of parallel architecture
SISD	–	Single Instruction set, Single Data set, a model of sequential architecture
SS	–	Scatter Search, the method
sSA	–	Sequential Simulated Annealing, the method
TS	–	Tabu Search, the method
UB	–	Upper Bound, of the goal function
UMA	–	Uniform Memory Access
VNS	–	Variable Neighborhood Search, the method
WCNS	–	Wrocław Center of Networking and Supercomputing

Part I

INTRODUCTION TO PARALLELISM AND JOB SCHEDULING

Chapter 1

Introduction

The development of optimization methods, particularly applied in production tasks arrangement, has proceeded towards modern and more effective sequence approaches since the beginning of this field. At the end of the 1970s, the turning point in the combinatorial optimization methods was the branch and bound (B&B) method regarded those days as a remedy for nearly all problems of great size which could not be solved by means of methods applied at that time. However, it soon occurred that the B&B method only slightly extended the scope of solvable problems (e.g. for a sum-cost, single-machine scheduling problem this size extended from 20 to 40–50 tasks). What is more, the cost necessary to obtain an optimal solution is much too high compared to economic benefits and its use in practice. The conclusion of these investigations was the definition of a bounded area of the B&B scheme application (see Figure 1.1).

The next breakthrough concerned the occurrence of advanced metaheuristic methods: first the simulated annealing method and next the method of genetic algorithms and the tabu search method. Enthusiasm lasted much longer: until around 2005 several dozen of different metaheuristics had been proposed though again those methods reached the limit of their abilities to the moment where the size of effectively solvable problems (i.e., these for which an average deviation from the optimal solutions was smaller than 1%) might be shifted to a number reaching thousands, but not millions or hundred millions. Eventually the concept of 'no-free-lunch' by Wolpert and Macready [271] finished the discussion. With reference to rough methods this concept may be paraphrased in the following way: without using special attributes of examined problems considerable advantage of one metaheuristic over the other cannot be obtained. What is interesting Wolpert and Macready proved that 'free-lunch' was possible to be obtained in co-evolutional, multi-cultural metaheuristics, i.e., parallel in a natural way. Since the mid-1980s, indeed, parallel many-levelled metaheuristics had been developed,

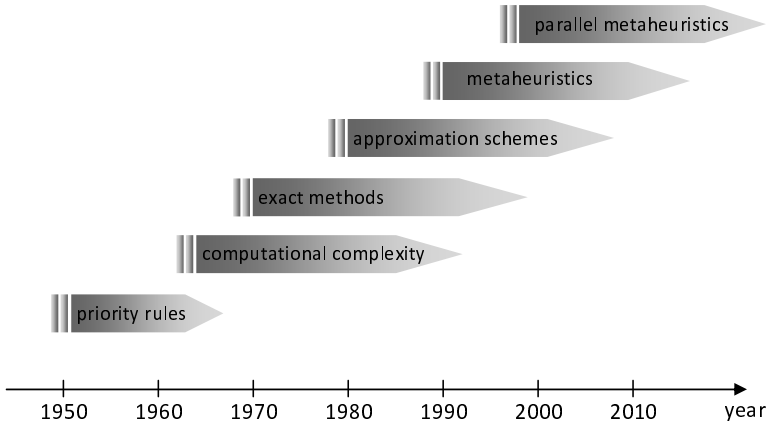


Fig. 1.1. History of the development of solution methods for job scheduling problems.

firstly as simple paralleling of the most time-consuming elements of sequence algorithms (usually as the goal function determination), then since the end of the 1990s as multi-track methods.

A marked enhancement of the quality of designed algorithms started when producers of computer equipment realized that further increase of the speed (i.e., the clock frequency) of processors was very costly, while this goal could be more easily obtained applying multi-core constructions, i.e., parallel calculating environments (and in this context among producers of hardware there also exist the term 'no-free-lunch'). Today processors of popular producers such as Intel or AMD have got 4 cores (some Intel processors have 9 cores, and prototypes even 80 cores) and GPU processors (Graphic Processing Unit) at first being used exclusively as graphic processors and nowadays also as strictly computing ones posses even 960 processors (e.g. products of nVidia Tesla series).

Increasing the number of cores requires however a wide range of algorithms – a sequence metaheuristic algorithm activation on a multi-core processor makes use of just one core, i.e., only a small part of the whole equipment potentiality. The specificity of optimization algorithms as well as procedures concerning determination of the key elements of a problem instance (e.g. the goal function value which is usually formulated in a recurrent way) makes automatic methods of paralleling unsuccessful. Specialized algorithms designed for the purpose of being activated in an environment of parallel calculations for specific kinds of problems are needed.

To date, very few works dealing with the application of parallel metaheuristic for job scheduling problems have been published, which follows from the fact that this is an interdisciplinary area connected with two science disciplines: computer science – as far as algorithm theory and parallel computing are concerned, and

automation – with regard to applications. There is a lack of theoretical properties of parallel scheduling algorithms. A complex synthetic approach would allow us to summarize the present state of research and fill up this gap. This book should fulfill this task.

There are plenty of genuine single-thread search methods proposed in this book, designed for homogeneous parallel systems. These methods take into consideration both dissimilar techniques of parallel algorithm design process and different necessities of modern algorithms of discrete optimization (analysis of one solution, analysis of a local neighborhood). Efficiency, cost and computation speedup depending on the type of problem, its size and environment of parallel system used is given special consideration in this part of the chapter. Theoretical estimations of properties have been derived for particular algorithms, and a comparative analysis of the advantages resulting from application of different approaches has been done.

In the area of multithread search, dedicated to homogeneous and heterogeneous multiprocessor systems (such as mainframe computers, clusters, diffuse systems connected by networks) a parallel variant of metaheuristic methods, such as tabu search, scatter search, simulated annealing, evolutionary algorithm, path-relinking method, population-based approach, has been designed and researched experimentally, in the application of scheduling problems. A concurrent exact method – branch and bound – has also been analyzed as a multiple-walk parallelization.

The present research is of interdisciplinary character, including inter alia: theory and practice of the algorithm design, theory and practice of parallel computing, theory and practice of job scheduling, exact and approximate methods of solving combinatorial optimization problems, artificial intelligence methods and theory of computational complexity.

The results presented in this monograph were obtained by the author while he was working on the following projects:

- 2002–2005 research project founded by the State Committee for Scientific Research No. 4T11A01624 (Wrocław University of Technology);
- 2009–2012, research project founded by the Ministry of Science and Higher Education No. N N514 23223 (Wrocław University of Technology),
- 2010–2011, habilitation research project founded by the Ministry of Science and Higher Education No. N N514 470439 (Wrocław University of Technology),

and as a result of cooperation in the field of practical applications with the Institute of Construction of the Wrocław University of Technology and with Lublin University of Technology [225, 224].

1.1. Performance metrics of parallel algorithms

A *parallel algorithm* can be defined as one that is concurrently executed on many different processing devices. In the language of operating systems a parallel algorithm can be equivalent to a *process* (or a group of processes), as an instance of a computer program being executed which is made up of multiple threads (*multithreaded* process) that follow instructions concurrently.

Many metrics have been applied based on the desired outcome of performance, due to determining the best version of the multithread parallel algorithm, evaluating hardware of the parallel system and examining the benefits of parallelization. In many cases the goal is to design a parallel algorithm whose execution cost (correlated with an electrical energy used or economical cost) is identical to the cost of a sequential algorithm execution solving the same problem. Such an algorithm is called *cost-optimal*. In the further part of this chapter, the performance metrics of parallel algorithms are defined precisely.

Parallel runtime. The execution time of the sequential algorithm is measured as the time elapsed between the beginning and the end of execution on a serial processor. We will denote such a *serial runtime* by T_s. By analogy, the *parallel runtime* T_p is the time which elapses from the moment the parallel computations begin till the moment the last processor stops the calculations.

Speedup. Let us consider a problem P, a parallel algorithm A_p and a parallel machine M with q identical processors. Let us define by $T_{A_p,M}(p)$ the time of calculations the algorithm A_p needs to solve the problem P on the machine M making use of $p \leq q$ processors. Let T_{A_s} be the time of calculations needed by the best (the fastest) known sequential algorithm A_s which solves the same problem P on the sequential machine with the processor identical to processors of the parallel machine M. We define the *speedup* as

$$S_{A_p,M}(p) = \frac{T_{A_s}}{T_{A_p,M}(p)}. \tag{1.1}$$

Thanks to the definition of speedup we can distinguish between: *sublinear speedup* ($S_{A_p,M}(p) < p$), *linear speedup* ($S_{A_p,M}(p) = p$) and *superlinear speedup* ($S_{A_p,M}(p) > p$), however the last one is still controversial. From the theoretical point of view it is not possible to obtain superlinearity of the speedup. If it were possible then one could construct a sequential algorithm by executing a parallel algorithm on $p = 1$ processors and such a sequential algorithm would be faster than the fastest one known. In fact many authors [9, 84, 64, 65, 269, 164, 165, 171, 178, 209] have reported superlinear speedup. One can point out

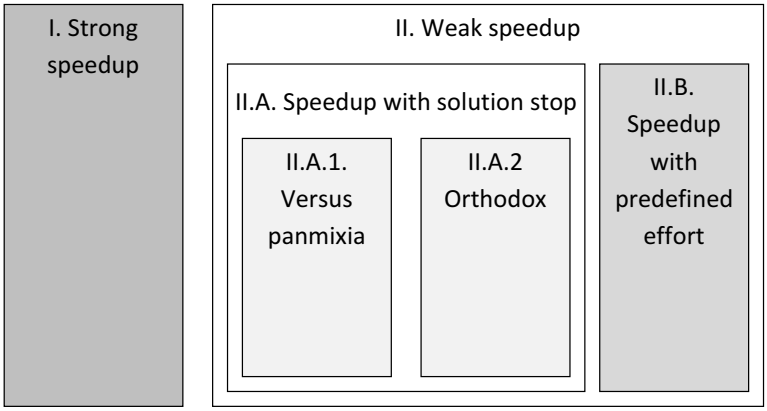


Fig. 1.2. Taxonomy of speedup measures proposed by Alba [7].

several sources behind superlinear speedup, such as *cache* memory influence, data structure properties or non-optimal decision made by sequential algorithm (see Kwiatkowski et al. [160]). More elaborate discussion of the superlinear speedup is given in Section 12.4.

Another difficulty is connected with the meaning of 'the best known sequential algorithm A_s'. For many problems, it is difficult to determine the best algorithm, especially for metaheuristics, where in fact we did not *solve* the problem (in the sense of finding the optimal solution) but we *approximated* the optimal solution – results obtained by using different metaheuristics are usually different. Two approaches to the problem of speedup definition for metaheuristics are proposed in the literature.

Alba [7] proposes the following taxonomy (Figure 1.2). Strong speedup (type I) compares the parallel runtime with the best-so-far sequential algorithm, therefore this definition equals definition (1.1). However due to the difficulty of finding the current most efficient algorithm most researches do not use it. Weak speedup (type II) compares the parallel algorithm against its serial version. Two stopping criteria can be used: result (solution) quality or maximum effort. The author proposes two variants of the weak speedup with solution stop: to compare the parallel algorithm with the 'canonical' sequential version (so-called versus panmixia, type II.A.1) or to compare the runtime of the parallel algorithm on p processors against the runtime of the same algorithm on one processor (orthodox, type II.A.2). The problem is that versus panmixia speedup measure compares two different algorithms. The orthodox speedup measure does not cause that kind of problem, that is why this method is usually applied to determine the speedup value of metaheuristics.

Barr and Hickman [20] propose a different taxonomy: *speedup, relative speedup* and *absolute speedup*. The *speedup* is defined by the ratio between the time of the parallel code using p processors of parallel machine against the time of the fastest sequential code on the same parallel machine. The *relative speedup* is the ratio of the execution time of the parallel code on p processors to the time of the sequential execution with parallel code on one processor (i.e., we set $p = 1$) of the same parallel machine. The *absolute speedup* compares the parallel time on p processors with the fastest sequential algorithm time on any computer.

Both approaches have similarities: *strong speedup* is identified with *absolute speedup* and *relative speedup* is similar to *orthodox speedup with solution stop* (type II.A.2). The last definition seems to be the most practical since there is no need to use the best algorithm.

Efficiency. The *efficiency* $\eta_{A_p,M}(p)$ of the parallel algorithm A_p executed on the parallel machine M is defined as

$$\eta_{A_p,M}(p) = \frac{S_{A_p,M}(p)}{p} \tag{1.2}$$

and describes an average fraction of time used by each processor effectively. The value of efficiency belongs to the range [0,1]. An ideal value of efficiency is 1 (in such a situation we can speak about a linear speedup) and it means that each processor is used as long as possible; therefore, there are no idle times of processors.

Cost. The *cost* $c_{A_p,M}(p)$ of solving a problem by using an algorithm A_p in a p-processor parallel machine M is defined as

$$c_{A_p,M}(p) = p \cdot T_{A_p,M}(p). \tag{1.3}$$

The cost reflects the sum of times of each processor work for solving the problem (see Figure 1.3). For sequential algorithms the time of problem solving by the fastest known algorithm using one processor constitutes also its cost. We can say that a parallel algorithm is *cost optimal* if its executing cost in a parallel system is proportional to the time of execution of the fastest known sequential algorithm on one processor. In such a case the efficiency equals $O(1)$.

Granularity. In parallel computing, granularity G is a qualitative measure of the ratio of computation T_c to communication T_t time units

$$G = \frac{T_c}{T_t}. \tag{1.4}$$

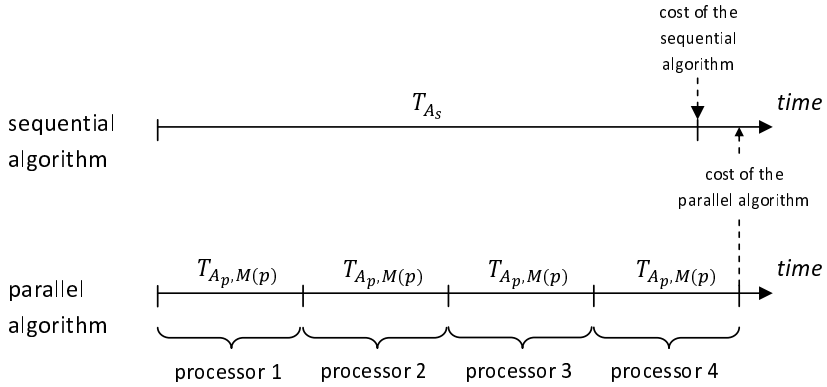


Fig. 1.3. An illustration of the cost definition (4-processor implementation).

Computation periods are typically separated from periods of communication by synchronization events. In *fine-grained parallelism* relatively small amounts of computational work are done between communication events. We can observe low computation to communication ratio. On the contrary in *coarse-grained parallelism* relatively large amounts of computational work are done between communication or synchronization events (see Figure 1.4). High computation to com-

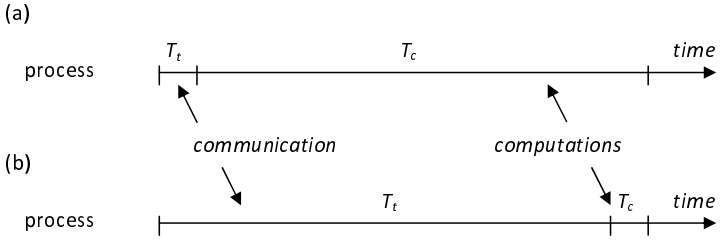


Fig. 1.4. An illustration of the fine-grained (a) and the coarse-grained (b) granularity.

munication ratio can be noticed in this case. This implies more opportunity for performance increase and it is harder to achieve efficient load balance. In order to attain the best parallel performance, the best balance between load and communication overhead needs to be found. If the granularity is too fine, the performance can suffer from the increased communication overhead. On the other hand, if the granularity is too coarse, the performance can suffer from load imbalance. The granularity G can be measured as T_c vs. T_t unit times as well as the sum of computation and communication times during the whole program execution (average empirical granularity). The results of granularity calculations for two supercomputers and two GPUs are shown in Table 1.1. Both supercomputers are coarse-grained parallel computing environments, $G > 100$ FLO/B (Floating Point

Operations per communication Byte). The GPUs presented are fine-grained computing environments, $G < 10$ FLO/B. Systems with $10 \geq G \geq 100$ can be named medium-grained, however it is difficult to find such hardware nowadays. Besides, G ranges change with time (e.g. every decade).

Table 1.1. The granularity G values for various parallel computing environments.

System	T_c	T_t	G (FLO/B)*
SGI Altix 3700 Bx2**	768 GFLOPS	0.54GB/s	1428
Cluster (329 Intel Xeon)**	19 TFLOPS	2.5 GB/s	760
GPU Tesla C1060	933 GFLOPS	102 GB/s	9.15
GPU Tesla C2050	1.3 TFLOPS	148 GB/s	8.78

* FLO/B – Floating Point Operations per communication Byte

** placed in the Wrocław Centre of Networking and Supercomputing [266]

1.1.1. Performance metrics for parallel metaheuristics

Quality metrics of parallel algorithms defined in Section 1.1 works well for programs which provide the same final result as their respective sequential version (e.g. matrix multiplication, determination of paths in graphs, etc.). A given metric can be applied to exact optimization algorithms, because their work effect is a global optimal solution. Metaheuristics create a completely different situation. Each metaheuristic run can give a solution with different goal function value. The quality of solutions thus obtained depends on the time of calculations, i.e., the shorter the time, the worse the quality of solutions. What is more, a sequence of metaheuristics realizes a search trajectory which depends on random variables which are parameters of the algorithm (e.g. simulated annealing, simulated jumping, genetic algorithm, scatter search, etc.). Therefore an output of a parallel algorithm A_p is incomparable with an output of a sequential A_s considering the result obtained – both depend on a data instance $\mathcal{I}$ and a vector $\mathcal{Z}$ of random parameters. Additionally, let us notice that the quality of the solution obtained by the sequential algorithm A_s depends on the number of executed iterations $iter$. Hence we have to consider a sequential runtime $T_{A_s}(\mathcal{I}; \mathcal{Z}; iter)$, a parallel runtime $T_{A_p}(p; \mathcal{I}; \mathcal{Z})$ and a speedup $S_{A_p,M}(p, \mathcal{I}; \mathcal{Z})$. Therefore we can define metrics mentioned above as

$$T_{A_s}^{iter} = \sup_{\mathcal{I},\mathcal{Z}} T_{A_s}(\mathcal{I}; \mathcal{Z}; iter), \tag{1.5}$$

$$T_{A_p}(p; \epsilon; iter) = \sup_{\mathcal{I},\mathcal{Z}} T_{A_p}(p; \mathcal{I}; \mathcal{Z}),\ |K^{A_p} - K_{iter}^{A_s}| < \epsilon, \tag{1.6}$$

where ϵ is an assumed absolute deviation from the parallel algorithm solution K^{A_p} to the sequential algorithm solution $K^{A_s}_{iter}$ obtained by executing $iter$ iterations of the algorithm A_s. A speedup

$$S_{A_p,M}(p;\epsilon;iter) = \frac{T^{iter}_{A_s}}{T_{A_p}(p;\epsilon;iter).} \tag{1.7}$$

Finally, we can approximate a speedup metric by

$$S^{lim}_{A_p,M}(p) = \lim_{\epsilon\to 0}\lim_{iter\to\infty} \frac{T^{iter}_{A_s}}{T_{A_p}(p;\epsilon;iter).} \tag{1.8}$$

Obtaining analytical results for the metrics thus defined is difficult, so we will use experimental metrics of the parallel runtime and speedup.

Apart from designing metrics for parallel algorithms, also standard (i.e., taken form sequential approach) quality and time metaheuristic metrics will be used in the further part of the book:

- PRD – Percentage Relative Deviation from reference solutions given by the formula

  $$PRD = \frac{F_{ref} - F_{alg}}{F_{ref}} \cdot 100\%,$$

  where F_{ref} is the reference criterion function value and F_{alg} is the result obtained by the parallel algorithm examined. This formula is not used when $F_{ref} = 0$,

- $APRD$ – Average Percentage Relative Deviation

  $$APRD = \frac{1}{n_{inst}} PRD_i,$$

  where n_{inst} is the number of benchmark instances and PRD_i is the PRD of the i-th problem instance.

- t_{total} (in seconds) – real time of an algorithm execution,

- t_{cpu} (in seconds) – the sum of time consumption on all processors.

1.2. Parallel architectures

In recent years, several theoretical models of parallel computing systems were proposed. Up till now some of them have been physically realized. These theoretical models take into account only the ways of manipulating instructions (instruction set) and the type of data streams. We extend this taxonomy by adding memory architectures.

1.2.1. Taxonomy

The fundamental classification of parallel architectures was given by Flynn [108]. Here we present it based on a survey taken from [91] and [89].

- **SISD machines.** *Single Instruction stream, Single Data stream.* Classic serial machines belong to this class. They contain one CPU and hence can accommodate one instruction stream that is executed serially. Many large mainframes can have more than one CPU but each of them execute instruction streams that are unrelated. Therefore, such systems still should be regarded as multiple SISD machines acting on different data spaces. Examples of SISD machines are mainly workstations like those of DEC, Hewlett-Packard, IBM and Silicon Graphics.

- **SIMD machines.** *Single Instruction stream, Multiple Data stream.* These systems often possess a large number of processing units, ranging from 100 to 100,000 all of which can execute the same instruction on different data. Thus, a single instruction manipulates many data items in parallel. Examples of SIMD machines are the CPP DAP Gamma II and the Quadrics Apemille. Other subclasses of the SIMD systems embrace the vector processors which manipulate on arrays of similar data rather than on single data items using CPUs with special instructions (e.g. MMX, SSE2). If data can be manipulated by these vector units the results can be delivered at a rate of one, two and three per clock cycle. That is why vector processors work on their data in a parallel way but this only refers to the vector mode. In this case they are several times faster than when executing in conventional scalar mode. An extension of the vector processing idea is GPGPU (general purpose graphic processing unit, see Figure 1.5).

- **MISD machines.** *Multiple Instruction stream, Single Data stream.* This category includes only a few machines, none of them being commercially successful or having any impact on computational science. One type of system that fits the description of an MISD computer is a systolic array which is a network of small computing elements connected in a regular

Fig. 1.5. The nVidia Tesla C2050 with 448 cores (515 GFLOPS).

grid. All the elements are controlled by a global clock. In each cycle, an element will read a piece of data from one of its neighbors, perform a simple operation and prepare a value to be written to a neighbor in the next step.

- **MIMD machines.** *Multiple Instruction stream, Multiple Data stream.* MIMD machines execute several instruction streams in parallel on different data. Compared to the multi-processor SISD machines mentioned above the difference lies in the fact that the instructions and data are related because they represent different parts of the same task to be executed. Therefore, MIMD systems can run many subtasks in parallel in order to shorten the time-to-solution for the main task to be executed. There is a large variety of MIMD systems and especially in this class the Flynn taxonomy proves to be not fully adequate for the classification of systems. If we focus on the number of system processors this class becomes very wide, from a NEC SX-9/B system with 4-512 CPUs or clusters of workstations (see Figure 1.6) to a thousand processors IBM Blue Gene/P supercomputer (see Figure 1.7) and Cray XT5-HE (224162 cores) which breaks the petaflops barrier.

1.2.2. Memory architectures

The Flynn taxonomy does not recognize memory architecture. In our opinion memory architecture types have an influence on parallel algorithm efficiency. Therefore, we propose to select two classes here (see [228]).

- *Shared memory systems.* They have multiple CPUs all of which share the same address space (shared memory). It means that the knowledge of where data is stored is of no concern to the user as there is only one memory

Fig. 1.6. The Nova cluster from the Wrocław Centre of Networking and Supercomputing, 2016 cores (19 TFLOPS). Source: WCNS [266].

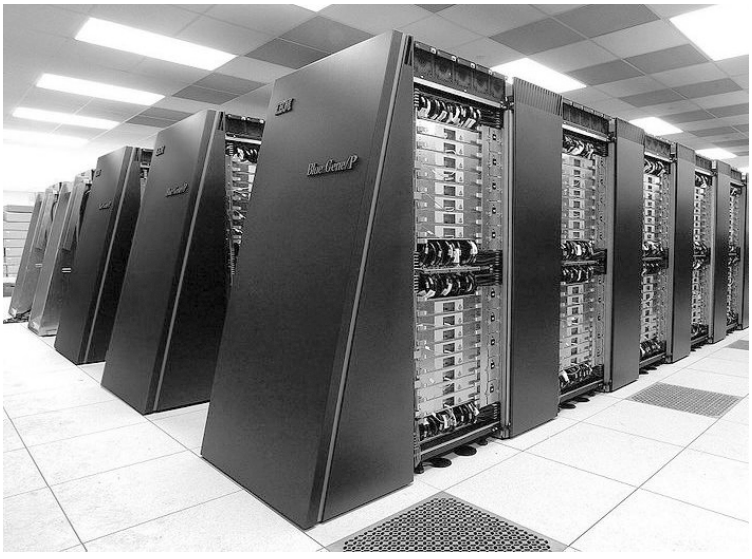


Fig. 1.7. The IBM Blue Gene/P supercomputer at Argonne National Laboratory, 163840 cores (459 TFLOPS).

accessed by all CPUs on equal basis. Shared memory systems can be both SIMD and MIMD. Single-CPU vector processors can be regarded as an example of the former, while the multi-CPU models of these machines are examples of the latter. The abbreviations SM-SIMD and SM-MIMD are usually used for the two subclasses.

- *Distributed memory systems.* Each CPU possesses its own associated memory in this class. The CPUs are connected by a network and they may exchange data between their respective memories if necessary. Unlike with the shared memory machines the user has to be aware of the data location in the local memories, besides they will have to move or distribute these data explicitly if necessary. The distributed memory systems may be either SIMD or MIMD.

Although the difference between shared- and distributed-memory machines seems to be clear, this is not always entirely the case from the user's point of view. Virtual shared memory can be simulated at the programming level. For example, a specification of High Performance Fortran (HPF) was published in 1993 [134] which, by means of compiler directives, distributes the data over the available processors. That is why the system on which HPF is implemented in this case will look like a shared memory machine to the user. Other vendors of Massively Parallel Processing systems (sometimes called MPP systems), like HP and SGI, are also able to support proprietary virtual shared-memory programming models due to the fact that these physically distributed memory systems are able to address the whole collective address space. Therefore, for a user such systems have one global address space spanning all of the memory in the system. Also packages like TreadMarks [11] provide a virtual shared memory environment for networks of workstations.

The other important issue from the user's point of view is the access time to each memory address of the shared memory. If this access time is constant, we say that the system is of UMA (uniform memory access) type, if it is not we call it NUMA (non-uniform memory access). Additionally, there is a distinction if the caches are kept coherent (coherent cache or CC-NUMA) or not (non-coherent cache or NC-NUMA). The extended full classification first developed by Flynn [108] and then improved by Alba [7] is presented in Figure 1.8.

For SM-MIMD systems we can mention OpenMP [76] that can be applied to parallelize Fortran and C++ programs by inserting comment directives (Fortran 77/90/95) or pragmas (C/C++) into the code. Also many packages to realize distributed computing are available. Their examples are PVM (Parallel Virtual Machine, [116]), and MPI (Message Passing Interface, [235]). This programming style, called the 'message passing' model has become so accepted that PVM and

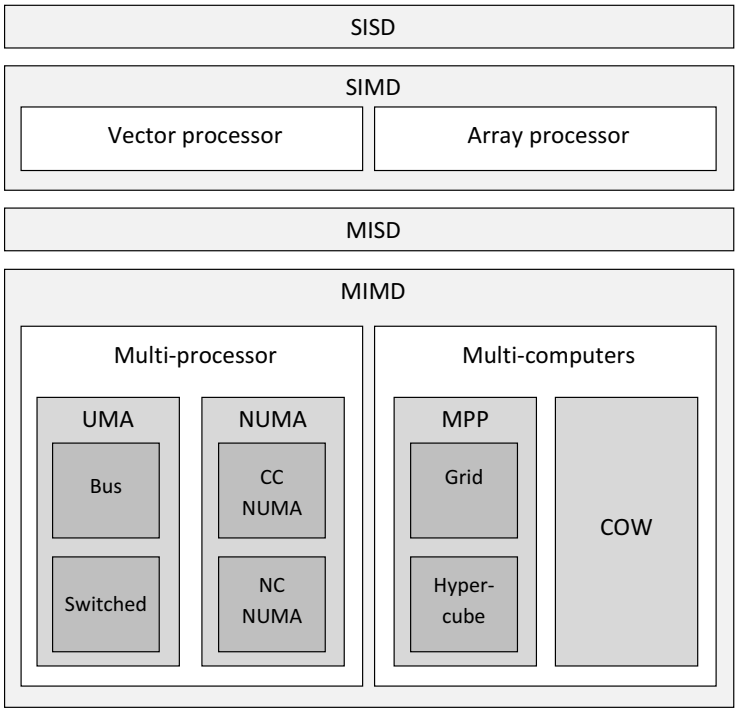


Fig. 1.8. Taxonomy of parallel architectures.

MPI have been adopted by nearly all major vendors of distributed-memory MIMD systems and even on shared-memory MIMD systems for compatibility reasons. In addition, there is a tendency to cluster shared-memory systems, for instance by HiPPI channels, to obtain systems with a very high computational power. E.g., the NEC SX-8, and the Cray X1 have this structure. Thus within the clustered nodes a shared-memory programming style can be applied, whereas between clusters a message-passing should be used. Nowadays, PVM is not applied a lot any longer and MPI has become the standard.

Distributed systems are usually composed of a set of workstations (so-called cluster) connected by a communication network such as Infiniband, Myrinet or Fast Ethernet. Such a *cluster of workstations* (COW) has better price-to-performance ratio, and it is more scalable and flexible compared to multiprocessor systems. On the other hand, MPP (*massively parallel processor*) systems are composed of thousands of processors, which can belong to multiple organizations and administrative domains, creating so-called *grids*, built on the basis of the Internet infrastructure.

1.2.3. Recent trends

For the last few years GPGPU parallel programming model has been used for massive shared-memory applications. GPUs are regarded as SIMD processors (or MIMD when the processors can handle multiple copies of the same code executing with different program fragments, e.g. counters, see Robilliard et al. [223]). In the CUDA programming environment, developed by nVidia, the GPU is viewed as a computing device capable of running a very high number of threads in parallel, operating as a coprocessor of the main CPU. Both the host (CPU) and the device (GPU) maintain their own DRAM, referred to as the host memory and device memory, respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's Direct Memory Access (DMA) engines.

The GPU is especially well-suited to address problems that can be expressed as data-parallel computations – SIMD – with high arithmetic intensity (the number of arithmetic operations is significantly greater than the number of memory operations). Because the same program is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. This property was used by Janiak et al. [147] to design a tabu search metaheuristic for GPU. In practice GPU programming is very close to the PRAM machine model (see Section 4.2) from the programmers' point of view, offering a simple tool for checking the theoretical PRAMs algorithm efficiency (see Bożejko et al. [35]).

1.3. Metaheuristic parallelization strategies

Metaheuristics based on the local search method can be presented as processes of a graph searching in which vertices are the points of the solution space (e.g. permutations) and arcs correspond to the neighborhood relation – they connect vertices which are neighbors in the solution space. We will call it *neighborhood graph*. For all NP-hard problems the related neighborhood has an exponential size. Moving on such a graph defines some *path* (in other words, *trajectory*) in the solution space. Parallel metaheuristic algorithms make use of many processes for parallel generation or search of the neighborhood graph.

One can define two approaches to parallelization of the local search process in relation to the number of trajectories which are concurrently generated in the neighborhood graph:

1. *single-walk parallelization* (single trajectory): fine-grained algorithms for fast communication purposes (the most computationally expensive parts of the algorithm are parallelized),

2. *multiple-walk parallelization* (many trajectories): coarse-grained algorithms, communication is less frequent, compared to the single-walk parallelized algorithms.

These approaches demand that the algorithm meet some requirements as regards communication and synchronization frequency, which implies the kind of granularity. Single-walk parallel metaheuristics are usually fine-grained algorithms (e.g. Bożejko, Pempera and Smutnicki, [39]), multiple-walk metaheuristics – coarse-grained (e.g. Bożejko, Pempera and Smutnicki, [37]).

Table 1.2. Parallel architectures and programming languages presented in particular chapters.

Chapter	Parallel method	Parallel architecture	Programming language	Scheduling problem
single-walk methods				
4	huge neighborhoods	CREW PRAM		single machine problem
5	cost function calculation	SIMD (GPU)	C++ CUDA	job shop problem
6	workload determination	CREW PRAM		flexible job shop problem
multiple-walk methods				
8	memetic algorithm	MIMD	Ada95	single machine problem
9	population-based approach	MIMD	C++ MPI	single machine problem
10	branch and bound	SIMD DM	Ada95	single machine problem
11	simulated annealing	SIMD DM	Ada95	flow shop problem
12	scatter search	SIMD DM	C++ MPI	flow shop problem
13	genetic algorithm	MIMD	Ada95	flow shop problem
14	hybrid approach	MIMD SIMD	C CUDA	flexible job shop problem
15	tabu search	SIMD	Ada95	flow shop problem

Single-walk parallel algorithms. Single walk algorithms go along the single trajectory, but they can use multithread calculations for the neighborhood decomposition (see *representatives* method, [195]) or parallel cost function computation. For example, calculations of the cost function value for more complicated cases are frequently equivalent to determining the longest (critical) path in a graph, as well as maximal or minimal flow. This kind of parallelization will be described in Part II of this book.

Multiple-walk parallel algorithms. Algorithms which make use of a multithread multiple-walk model search concurrently a solution space by searching threads working in parallel. Additionally, these algorithms can be divided into subclasses due to communication among threads (information about current search status): (1) *independent* search processes and (2) *cooperative* search processes. If the multithread application (i.e., concurrently running search processes) does not exchange any information we can talk about *independent* processes of search. However, if information accumulated during an exploration of the trajectory is sent to another searching process and used by it, then we can talk about *cooperative* processes (see Bożejko et al. [37]). We can also come across a mixed model, so-called *semi-independent* (see Czech [92]) executing independent search processes keeping a part of common data. Examples of such a method of parallelization are described in Part III of this book.

Implementation Due to the specificity of the metaheuristic type, as well as parallel environment architecture (SIMD, MIMD, shared memory, etc.) different programming languages are used for coding. As we can see in Table 1.2 SIMD algorithms for GPU are implemented in C++ with CUDA programming library – nowadays it is the most commonly used programming environment for nVidia GPUs. SIMD algorithms for multiprocessor computers without shared memory are implemented in Ada95 high-level programming language, due to the simplicity of designing them. Algorithms for distributed MIMD clusters are implemented in C++ programming language with the use of MPI (*Message Passing Interface*) communication library, also the most commonly used tool for programming clusters.

Chapter 2

The methodology of metaheuristic parallelization

This chapter is aimed at presenting and discussing the methodology of the metaheuristic algorithm parallelization. The majority of practical job scheduling issues belong to the class of strongly NP-hard problems, which require complex and time-consuming solution algorithms. Two main approaches are used to solve these problems: exact methods and metaheuristics. On the one hand, existing exact algorithms solving NP-hard problems are characterized by an exponential computational complexity – in practice they are extremely time-consuming. Although in recent years (1993–2008, see `www.top500.org`) the speed of the best supercomputer increases 10 times each 3 years (as 10^n function), this increase has only a little influence on the size of solvable NP-hard problems (e.g. most permutational job scheduling problems have the solution space of the size $n!$ which behaves[1] as n^n). On the other hand, metaheuristics, a subclass of approximate methods, provide suboptimal solutions in a reasonable time, even being applied in real-time systems.

Quality of the best solutions determined by approximate algorithms depends, in most cases, on the number of solutions being analyzed, therefore on the time of computations. Time and quality demonstrate opposite tendencies in the sense that obtaining a better solution requires significant increase of computing time. The construction of parallel algorithms makes it possible to increase significantly the number of solutions considered (in a unit of time) using effectively multiprocessor computing environment.

The process of an optimization algorithm parallelization is strongly connected with the solution space search method used by this algorithm. The most frequent are the following two approaches: *exploitation* (or *search intensification*)

[1] From the Stirling equation, $n! \sim (\frac{n}{e})^n \sqrt{2\pi n}$.

and *exploration* (or *search diversification*) of the solution space. Due to this classification we can consider major categories of the metaheuristic class such as: local search methods (e.g. tabu search TS, simulated annealing SA, greedy randomized adaptive search procedure GRASP, variable neighborhood search VNS) and population-based algorithms (e.g. genetic algorithm GA, evolutionary strategies ESs, genetic programming GP, scatter search SS, ant colony optimization ACO, memetic algorithm MA, estimated distribution algorithms EDAs). Local search methods (LSM) start with a single initial solution improving it in each step by neighborhood searching. LSMs often find a locally optimal solution – they are focused on the solution space *exploitation*. Population-based algorithms (PBAs) use a population of individuals (solutions), which is improved in each generation. Thus the average goal function of the whole population usually improves, which does not mean that all the individuals are improved. The whole process in randomized, so these methods are almost always non-deterministic. We can say that PBAs are focused on the solution space *exploration*.

2.1. Parallel local search methods

Let us consider a *discrete optimization problem* formulated as follows. Let $\mathcal{X}$ be a discrete solution space and let $F : \mathcal{X} \leftarrow \mathbb{R}^+$ be a non-negative function defined on the solution space $\mathcal{X}$. We are looking for the optimal element $x^* \in \mathcal{X}$ such that

$$F(x^*) = \min_{x \in \mathcal{X}} F(x). \tag{2.1}$$

A major class of discrete optimization problems solving algorithms (apart from population-based methods) is a *local search* approach, in which an algorithm creates a searching trajectory which passes through the solution space $\mathcal{X}$. Before its parallelization, let us formally describe this class of methods.

The well-known local optimization procedure begins with an initial solution x^0. In each iteration for the current solution x^i the neighborhood $\mathcal{N}(x^i)$ is determined. Next, from the neighborhood the best element $x^{i+1} \in \mathcal{N}(x^i)$ is chosen (i.e., with the best cost function value $F(x^{i+1})$) constituting the current solution in the next iteration. The method is exhaustive. An outline of the local search method is presented in Figure 2.1. The method generates a solutions sequence $x^0, x^1, x^2, \ldots, x^s$ such that $x^{i+1} \in \mathcal{N}(x^i)$. We called this sequence a *trajectory*. The problem (2.1) can be replaced by

$$F(x^A) = \min_{x \in \mathcal{Y}} F(x). \tag{2.2}$$

where

$$\mathcal{Y} = \{x^0, x^1, x^2, \ldots, x^s\} \subseteq \mathcal{X}. \tag{2.3}$$

We call the mechanism of a neighbor generation a *move*. More precisely, the move μ is a function $\mu : \mathcal{X} \to \mathcal{X}$ which generates solutions $\mu(x^i) = x^{i+1} \in \mathcal{N}(x^i) \subseteq \mathcal{X}$ from a solution $x^i \in \mathcal{X}$.

```
Algorithm 1.  Local Search Method (LSM)
  Select a starting point x^0;
  x_best ← x^0  i ← 0;
 repeat
    choose the best element y from the neighborhood N(x^i)
     according to a given criterion based on the
    goal function's value F(y);
    x^i ← y  i ← i + 1;
    if F(y) < F(x_best) then
     x_best ← y;
 until F(y) ≠ F(x_best).
```

Fig. 2.1. Outline of the Local Search Method (LSM).

A crucial ingredient of the local search algorithm is the definition of the neighborhood function in combination with the solution representation. It is obvious that the choice of a good neighborhood is one of the key factors ensuring efficiency of the neighborhood search method. A neighborhood $N(x)$ is defined as a subset $N(x) \subset \mathcal{X}$ of solutions 'close to' a solution $x \in \mathcal{X}$. A metric of the 'nearness' can be a distance metric in this solution space (e.g. Hamming's or Caley's, see [99]), or the number of moves.

2.1.1. Parallel local search strategies

Generally, several approaches to convert LSM to parallel LSM (p-LMS) can be formulated:

(a) calculating $F(x)$ faster for a given $x \in \mathcal{X}$,

(b) making a choice of $x^{i+1} \in N(x^i)$ faster,

(c) making a space decomposition among p searching threads, i.e.,

$$F(x^A) = \min_{1 \le k \le p} F(x^{Ak}) \tag{2.4}$$

where

$$F(x^{Ak}) = \min_{x \in \mathcal{Y}^k} F(x), \mathcal{Y}^k = \{x^{0k}, x^{1k}, \ldots, x^{sk}\}. \tag{2.5}$$

(d) using cooperative trajectories.

Alba [7] proposed the following classification:

- *Parallel multi-start model.* In this model several local search processes are executed concurrently, each one starting from the different solution. Either homogeneous or heterogeneous version of this model can be applied. They can be based on the same searching strategy, or have different strategies. Multiple working searching processes can also start from the same starting point, but with different searching strategies (e.g. with different parameters). Simple classification of such algorithms on the tabu search metaheuristic example was proposed by Voss in [261]. This model belongs to the multiple-walk parallelization class.

- *Parallel moves model.* This is a low-level parallelization model which consists in neighborhoods concurrent searching. The main metaheuristic which uses this kind of parallelism, computes the same results as the sequential version but faster. Each processor evaluates a part of neighborhood preparing the best element (so-called representative) as the proposition for the controlling processor which chooses the best solution from all representatives. This model is usually implemented as a master-slave model of parallelization, yet it can be developed both as the single-walk method and the multiple-walk parallelization (i.e., inside a hybrid method as a low level parallelism).

- *Move acceleration model.* The goal function value is calculated in a parallel way in this model. Such a parallelization is problem-oriented and strongly dependent on the goal function form. For example, it is difficult or even impossible to parallelize the function which has a recurrent form. Usually loops, minimum or sum calculations, are parallelized in this model. Because of the input-output intensity that kind of parallelism needs a shared-memory fine-grained parallel environments such as multi-processor mainframe computers or GPUs. Similarly to the previous (parallel moves) model it can be developed both as the single-walk method and as the multiple-walk parallelization.

Most survey works consider only parallel multi-start model of parallel local search metaheuristics, see [7, 8, 13, 16, 58, 61, 85, 106, 155, 177, 244, 245]. This is due to the difficulty of designing parallel moves and move acceleration models which are strongly dependent on the optimization problem formulation (see Bożejko [25] and Steinhöfel et al. [237]). This parallelization also needs to take advantage of the special properties of the optimization approach, i.e., neighborhood determination method, cost function calculation and methods of calculations distribution among processors. In Chapters 4, 5 and 6, we propose genuine single-walk parallelization methods, using special theoretical properties of a problem.

The taxonomy of Alba [7] corresponds with approaches (a)–(c) proposed at the beginning of this section, but it does not include tree-based searching metaheuristics (cooperative trajectories – approach (d)), in which a single processor starts from an initial solution, and next processors begin their searching processes from (usually very good) solution on the trajectory visited. Such concurrent threads create a tree-like trajectory. Therefore, we propose an *extension* of Alba taxonomy of parallel local search methods by including (at least) the following additional model:

- *Parallel tree-based model.* In this model, local search processes are concurrently executed; each one starting from the solution found by another process, i.e., as soon as its best solution is found. The most frequent approaches are: the blackboard broadcasting method using shared memory, and the master-slave model in which the master process is controlling the whole searching process and local search threads are executed on slave processors.

The methods listed above can also be used together as hybrid parallel metaheuristics (see Chapter 14.1).

2.1.2. Simulated Annealing

Simulated Annealing (SA) is a stochastic heuristic algorithm which explores the solution space using randomized search procedure. The method uses a control parameter called temperature to determine the probability of accepting a solution with a worse cost function value (non-improving). The temperature decreases as the algorithm proceeds according to the so-called cooling scheme such that non-improving solutions are accepted at the end of the algorithm work. The main objective is to escape from local optima keeping the convergence of the whole searching process. Because of quick implementation, simulated annealing is a popular method for solving discrete optimization problems, such as single and multi-machine scheduling problems, TSP, QAP, timetabling, etc. Simulated annealing produces the proof of its theoretical convergence to the global optimum, also in its move acceleration model parallelization (see Meise [183]).

Simulated annealing method can be parallelized in several ways:

(i) parallel goal function calculations of a single solution (single-walk parallelization, fine-grained, convergent),

(ii) parallel goal function calculations of a few solutions (single-walk parallelization, fine- or medium-grained, convergent),

(iii) acceleration of achieving thermodynamic equilibrium state in the fixed temperature (single-walk parallelization, medium-grained, convergent),

(iv) multi-threaded independent work (multiple-walk parallelization, coarse-grained, convergent),

(v) multi-threaded cooperative work (multiple-walk parallelization, coarse-grained, convergent).

Most of the SA parallelizations (pSA) can be classified into two categories: (i) move acceleration (Kravitz and Rutenbar [159]) and (ii) parallel moves (Roussel-Ragot and Dreyfus [227]). The parallel moves model has been most frequently investigated. It relies on concurrent evaluation of different moves. This approach has to tackle the problem of inconsistency: the cost function value may be incorrectly computed due to moves executed by other processors. Such an inconsistency can by managed in two ways:

(1) only non-interacting moves are accepted (*domain decomposition approach*),

(2) interacting moves are evaluated and accepted and some errors in the cost function value calculation are allowed – they are corrected after a certain number of moves or before temperature change, using synchronization of processors.

The speed of convergence of the parallel SA, based on the parallel moves model, is comparable to the sequential algorithm convergence. The cost of synchronization has an adverse influence on the parallel algorithm – some authors report negative speedups obtained, as in Haldar et al. [131].

Several parallelizations follow the parallel multi-start model using multiple Markov chains (e.g. Haldar et al. [131], Lee and Lee [167]) and many of them are applied to the cell placement problem. In this approach each processor executes SA on a local copy of the problem data dynamically combining solutions by exchanging the best ones synchronously or asynchronously (see Haldar et al. [131]).

2.1.3. Tabu Search

Tabu Search (TS) method was introduced by Glover [126] in 1986 as an extension of classical local search methods (LSM). It explores the solution space by local search procedure with the use of neighborhoods, that is specific inner heuristic designed to evaluate solutions. Usually the candidate solution is the best found in the neighborhood (*the best improvement* rule), however it can be the first one found, too (*the first improvement* rule). Classic local search procedures such as Descent Search (DS) rely on the monotonic improvement stopping after obtaining local minimal solutions, for which all solutions in the neighborhood are worse (or not better). The main improvement of TS method compared to classic DS is that it can overcome local optima and keep the search going. To prevent its searching trajectory from making cycles, TS keeps the history of the searching process,

e.g. visited solutions on the trajectory. Usually it is enough to remember only a few (e.g. seven) last solutions, however in some theoretical cases it is useful to remember the *whole* searching history, which makes it possible to prove theoretical convergence of such a TS algorithm (see Hanafi [132]).

Most of TS parallel implementations are based on the multi-start model (Talbi et al. [249] for Quadratic Assignment Problem (QAP), Crainic and Gendreau [86] for capacitated network design) or a neighborhood decomposition (Porto and Ribeiro [209] for task scheduling under precedence constraints) or both of them (Cahon et al. [70]) In [209] also the parallel moves model, apart from a neighborhood decomposition, is applied to solve the problem, obtaining near-linear speedup for large problem instances.

From the theoretical point of view, the first taxonomy of parallel tabu search algorithms was proposed by Voss [261] based on the Flynn [108] classification of parallel architectures (SIMD, MIMD, MISD and SISD). Voss proposed to classify parallel tabu search algorithms into four categories with reference to starting points and searching strategies.

- SPSS (*Single Point Single Strategy*) – search starts from a single solution along a single strategy. This model allows us to parallelize on the lowest level only (*parallel moves* or *move acceleration* models),
- SPDS (*Single Point Different Strategies*) – all searching threads start from the same solution with different search strategies, i.e., different neighborhoods, tabu list length, elements remembered on the tabu list, etc.,
- MPSS (*Multiple Point Single Strategy*) – threads begin from different starting solutions using the same searching strategy,
- MPDS (*Multiple Point Different Strategies*) – threads begin from different starting solutions applying different search strategies; this class is the widest and includes all previous classes.

The taxonomy presented above was extended by Crainic, Toulouse and Gendreau [85] by introducing two additional classifications: (1) due to the number of processors which keep control over the algorithm work and (2) the way of control and the type of communication. Considering the number of controlling processors two models can be distinguished:

a) *1-control* – determined central processor controls the work of a parallel algorithm,
b) *p-control* – the algorithm execution control is distributed among p concurrently working processors.

The way of control and the type of communication are determined by the following classes:

(i) *rigid synchronization*,
(ii) *knowledge synchronization*,
(iii) *collegial*, and
(iv) *knowledge collegial*.

The first categorization is connected with the quantity and quality of the information exchanged and shared – first two cases are synchronized ones, next two cases are collegial. The second categorization shows the possibility of using additional knowledge which can be derived from information exchange – cases (ii) and (iv) are using a base of knowledge, cases (i) and (iii) are not.

2.2. Parallel population-based algorithms

Population-based algorithms (genetic, memetic, particle swarm optimization, etc.) are well-suited to parallelization due to their natural partitioning into separate groups of solutions, which are concurrently processed. The method of using population of individuals allows us to *diversify* searching process onto the whole solution space. On the other hand, using cooperation, it is easy to *intensify* searching after finding a good region by focusing individuals onto it. Thanks to its concurrent nature, population-based algorithms are very handy to parallelize, especially in the independent way using multi-start model. Low level parallelization is not so easy because special properties of the problem being considered have to be usually used. We present and discuss such an approach in Chapters 9 and 14.

2.2.1. Genetic Algorithm

Genetic Algorithm (GA) method is an iterative technique that applies stochastic operators on a set of individuals (population). Each individual of the population encodes the complete solution. Starting population is usually generated randomly. A GA applies a recombination operator (crossover) on two solutions in order to introduce diversity of population. Additionally, a mutation operator which randomly modifies an individual is applied being the insurance against stagnation of the search process. Traditionally GA was associated with the binary representation of a solution, however in job scheduling area a permutational solution representation is more popular and useful.

The performance of population-based algorithms, such as GAs, is specially improved when running concurrently. Two strategies of parallelization are commonly used:

1. parallelization of computations, in which operations allied to each individual (i.e., goal function or genetic operators) are performed in parallel, as well as

2. population parallelization in which the population is partitioned into different parts which can be evolved concurrently or exchanged.

We distinguish the following kinds of parallelization techniques which are usually applied to genetic algorithms:

- *Global parallelization.* This model is based on the master-slave type concurrent processes. Calculations of objective functions are distributed among several slave processors while the main loop of the genetic algorithm is executed by the master processor.

- *Independent runs.* This approach runs several versions of the same algorithm with different parameters on various processors, allowing the parallel method to be more efficient. The independent runs model can also be considered as the distribution model without migration.

- *Distributed* (*island*) *model.* This model assumes that a population is partitioned into smaller subpopulations (islands), for which sequential or parallel GAs (usually with different crossover and mutation parameters) are executed. The main characteristic of this model is that individuals within a particular island can occasionally migrate to another island.

- *Cellular* (*diffusion*) *model.* In this model the population is mapped onto neighborhood structure and individuals may only interact with their neighbors. The neighborhood topology is usually taken from the physical network of processors, so this is a fine-grained parallelism where processors hold just a few individuals.

The distribution model is the most common parallelization of parallel GAs since it can be implemented in distributed-memory MIMD machines, such as clusters and grids. This approach leads to coarse-grain parallelization (Bożejko and Wodecki [50], Bożejko et al. [42]). Fine-grained parallel implementations of the cellular (also called diffusion) model are strongly associated with the machines on which they are executed ([94]). Master-slave implementations are also available as general frameworks (e.g. ParadisEO of Cahon et al. [70]).

A special case of GA is *Genetic Programming* (GP) approach, in which evolving individuals are themselves programs instead of finite-length vectors or permutations. The method was proposed by Koza [157] as an extension of evolutionary approach – the GP is a machine learning technique aimed at helping computers to program themselves; it allows us to discover automatically programs that solve or approximately solve a given problem.

Several fine-grained parallelizations of GP were proposed, see e.g. Juille and Pollack [151], Folino et al. [109] for cellular model on distributed-memory computers. The authors also introduced coarse-grained approaches: a master-slave and

a distributed model (Fernández et al. [104]), as multiple-population island-based parallel genetic programming.

Generally, models of parallel and distributed genetic programming presented in the literature can be categorized as follows:

1. parallelizing at fitness level,
2. parallelizing at population level – the island and the grid (cellular) models.

Since individuals in genetic programming method feature both different sizes and complexities, as a result the problem of imbalance in *parallelizing at fitness level* appears. Load balancing can be obtained in an automatic way if a steady-state reproduction is used (instead of generation reproduction).

The *island GP model* can be easily implemented on both distributed memory machines and clusters of networked workstations or grids due to the rare communication frequency. The migration between islands is usually implemented using message-passing interface (such as MPI or PVM). Each island executes a standard GP and individuals are exchanged at fixed synchronization points, or asynchronously, using an additional processor.

In the *grid* (*cellular*) *GP model* each individual is placed in a fixed location on a low dimensional grid. A scalable implementation of cellular GP model was proposed by Folino et al. [109]. They proposed three replacement policies for this model of GP: *direct* (the best of the offspring replaces the current individual), *probabilistic* (the offspring replaces the parent according to the difference between their fitness) and *greedy* (the offspring replaces the parent if the offspring is fitter).

2.2.2. Scatter Search

Scatter Search (SS) is a population-based algorithm which evaluates solutions from a starting set to create new solutions added to this set. The chosen pairs of solutions are used to build a new one using a special procedure (e.g. linear combination, path-relinking). This approach involves different starting solution generation procedures, reference set update procedures, constructed solutions improving procedures, etc. To improve built solutions, local search procedures are usually applied.

Parallelization can be used on each level of the scatter search method: the local improvement procedure (parallel moves model, Synchronous Parallel Scatter Search in García-López et al. [113] for the p-median problem), the solution combination procedure (García-López et al. [114], also Replicated Combination Scatter Search in [113]) and by parallelizing the whole method by introducing a multi-start model (Replicated Parallel Scatter Search in [113]).

The *local improvement procedure* parallelization can be obtained by using the different parallel models of local search methods. Not only does it allow us to

reduce the computational time, but it also lets us obtain better results than a sequential algorithm in the same number of iterations. The *solution combination procedure* parallelization can be obtained by dividing the set of possible combinations among a set of processors. Different combination methods and different parameter settings can be applied in such a parallelization (see [114]). This approach increases the stability (i.e., parameter setting invulnerability) of the parallel SS algorithm and improves its precision without increasing the computational time (see Bożejko and Wodecki [40] for the flow shop scheduling problem). In the whole *scatter search* parallelization model each processor executes an SS procedure. Such a model is a multi-start approach and it increases the diversification of solutions. Its intensification is obtained by exchanging the information about the best solution found.

2.2.3. Memetic Algorithm

Memetic Algorithm (MA) is an evolutionary approach based on the process of natural evolution adhering to the principles of natural selection, crossover and survival. Lamarck's model (see Michalewicz [187]) of evolution is applied to intensify the optimization process. In each generation a certain part of the population is replaced by their local minima simulating a learning effect which can be succeeded by the next generation as a 'meme'. From the current population some subset is drawn. Each individual of this subset is a starting solution for the local optimization algorithm. Thus, there are five essential steps of the MA:

1. *selection* – choosing some subset of individuals, so-called parents,
2. *crossover* – combining parts from pairs of parents to generate new ones,
3. *mutation* – transformation that creates a new individual by small changes applied to an existing one taken from the population,
4. *learning* – an individual is improved (e.g. by a local optimization),
5. *succession* – determining the next generation's population.

New individuals created by crossover or mutation replace all or a part of the old population. The process of evaluating fitness and creating a new population generation is repeated until a termination criterion is achieved.

Similar to the GA, the following kinds of parallelization are usually applied to MAs:

- global parallelization,
- independent runs,
- island model,
- diffusion model,

with similar properties as applied to the classic GA. Additionally, a local search procedure can be parallelized in MA. Such an approach is proposed by Berger and Barkaoui [23] and applied to the Vehicle Routing Problem with Time Windows (VRPTW) by using a master-slave parallel approach. The master controls the memetic algorithm execution, synchronizes and handles parent selection while the slaves execute genetic operations together with local search in parallel. Parallel memetic algorithm was also considered by Bradwell and Brown [66] (asynchronous MA) and Tang et al. [252] (MA based on population entropy).

2.3. Other methods

Greedy Random Search. The so called *Greedy Randomized Adaptive Search Procedure* (GRASP) is an iterated (i.e., multi-start) procedure where each iteration consists of two phases: construction of the solution and local search from the given solution. In the construction phase a feasible solution is built. A set of candidate elements is made up of those elements which can be added to the partial solution without making it unfeasible. Next, the candidate element is evaluated by the greedy function which measures the benefit of including it to the partial solution producing a restricted candidate list which consists of those elements for which the greedy function is not lower than the chosen parameter (threshold). The element to be included into the partial solution is randomly selected from this restricted candidate list. The iterated process is terminated when all the elements are added to the partial solution, that is when the set of candidate elements is empty. The second phase – local search – allows us to provide a local optimum (in the chosen neighborhood) of the solution constructed in the first phase.

Most parallel GRASP implementations are based on the parallel multi-start model, both as multiple-walk independent search (Verhoeven and Aarts [260]) and multiple-walk cooperative search (Cung et al. [90]), also with the path-relinking procedure (Ribeiro and Rosseti [221]). Parallelizations are based on executing the same algorithm on the distributed data (*Single Program Multiple Data* SPMD model). Even in the independent computations very little information is exchanged between processors so almost-liner speedups are often obtained.

A natural way to obtain an efficient load balancing of processors is to distribute iterations among the processors, usually in a dynamic distribution approach. Each processor receives a copy of the sequential algorithm and a copy of the problem instance data. The cost of communication is low because of the independency of iterations. This approach is especially efficient in a heterogeneous multiuser execution environment, e.g. clusters and grids. Such a dynamic approach for parallel GRASP applied to the Steiner problem in graphs is presented in the paper of Martins et al. [179] based on the farmer-worker cycle stealing strategy.

Initially, each worker is assigned a small number of iterations. After performing its iterations, the processor requests additional iterations from the farmer processor, thus faster processors perform more iterations than slower ones.

Recently, there have appeared some works on GRASP hybridization by implementing it together with the path-relinking approach [219], which can be categorized as multiple-walk independent or cooperative-thread where processors exchange information about elite solutions visited during previous algorithm iterations.

Variable Neighborhood Search. The *Variable Neighborhood Search* (VNS) method is based on the idea of exploring the solution space by a single trajectory using successively changed neighborhoods from the predefined set, as well as descent search method to get the local minimum. It can also explore a random neighborhood instead of changing it in the predefined order (however such a way makes this method non-deterministic). The VNS method makes use of the following observations:

1. the local minimum defined in one neighborhood structure is not necessarily the local minimum in another neighborhood structure,
2. the global minimum (or minima) is also a local minimum for all neighborhood structures we can define.

Of course VNS, as a metaheuristic, is not an exact optimization method – it does not need to find a global optimum, it is enough to find a very good approximation of this solution (a suboptimal solution). We assume that for suboptimal solutions the second observation is true, too.

Parallel VNS method is quite a new method, so only a few parallelizations have been designed. We can mark out two major approaches here: simple parallelizations consisting in parallelizing the local search procedure and replicating the whole algorithm in the processors (García-López et al. [112]). More complex approaches to parallelization are based on a synchronous cooperation mechanism through a master-slave model (*Replicated-Shaking VNS* from [112]) or apply a cooperative multi-search method based on a central-memory mechanism (*Cooperative Neighborhood VNS* from Crainic et al. [87] for p-median problem). In [87] the former keeps current the best solution, also updating it, making communications among processors, initiates and terminates the overall procedure. The communication is initiated by workers asynchronously.

In the *Replicated-Shaking VNS* (RS VNS, proposed in [112]) the master processor executes a sequential VNS algorithm sending the current solution to each slave processor which makes a random perturbation obtaining a starting solution for the local search procedure. Solutions obtained by slaves after execution of

the local search algorithm are sent to the master which selects the best one and continues the parallel VNS algorithm.

The *Cooperative Neighborhood VNS* (CN VNS, described in [87]) is also based on the centralized approach, in this case in the form of central-memory mechanism connected with the multi-start parallelization model. Several independent VNS threads cooperate by using asynchronous communication – the information about the best solution found so far is exchanged which allows us to intensify solution space exploration by a number of VNS cooperative threads.

Both GRASP and VNS belong to the parallel local search class. In the further part of this section we briefly present other parallel population-based approaches which were not mentioned in Section 2.2.

Evolution Strategies. *Evolution Strategies* (ESs) belong to Evolutionary Algorithms (e.g. Genetic Algorithm and Genetic Programming), a big subclass of the population-based algorithms. This method is designed to solve continuous optimization problems, usually using elitist selection model and specific mutation without crossover operators. The individual consists of float objective variables and additional parameters guiding the search process. Therefore, Evolution Strategies algorithms are characterized by a self-adaptive control by evaluating the problem variables as well as the strategy parameters.

Parallelization of ESs is based on cellular and distributed models. In the cellular approach individuals are connected with neighbors and can interact only through these communication canals (Weinert et al. [265]). Also distributed parallelizations are provided (De Falco et al. [95], Schütz and Sprave [230]). Parallel models of ESs can be categorized into migration (for MIMD machines) and pollination (for SIMD machines) models. Another model usually used for describing population-based metaheuristics, especially evolutionary ones, is a cellular model, also called either a neighborhood or a diffusion model.

In the *migration* approach to parallel Evolution Strategies the population is partitioned into p subpopulations (where p is the number of processors). Processors connected with subpopulations exchange individuals from time to time. Variables which have to be defined are:

1. migration paths,
2. migration frequency,
3. number of migrants,
4. selection policy for migrants,
5. integration policy for migrants.

The *pollination* model of parallel ESs consists in spreading the genetic information by means of pollination but without moving individuals. On SIMD

machines each individual of the population is placed on a processor and the interaction takes place in the neighborhood defined by the hardware (processors) connection structure, e.g. two or three-dimensional torus, ring or hypercube. For example, if we consider a ring topology and a certain radius r, the neighborhood of each individual could be defined as r individuals to the right and left. This approach can be implemented on MIMD machines, too [236].

Ant Colony Optimization. The *Ant Colony Optimization* (ACO) method was inspired by the behavior of real ants which deposit a pheromone on the ground. This special chemical substance influences choices of an ant, i.e., it increases the probability that an ant selects the path proportionally to the amount of pheromone lying on this path. An ACO algorithm constitutes a method in which artificial ants are stochastic construction procedures which build a solution using heuristic information about the problem and pheromone trails.

Most of the ACO parallelizations use a master-slave model (Talbi et al. [248], Doerner et al. [100]). In this model a pheromone matrix is stored by the master processor and slaves are used to evaluate a portion of solutions as ant colonies. Similarly, a distributed (island) model also uses colonies of ants, however with greater independence – such a multicolony ant algorithm relatively rarely exchanges information between colonies (Mendes et al. [184], Middendorf et al. [188]). An independent model, without communication, was considered in the literature (Stützle [241]), too.

Usually parallel ACO algorithms put more than one ant on each processor. Such a group of ants if called a *colony* – several ants placed on a single processor cooperate better than ants placed on other processors. We call a parallel ACO *muli-colony* if a colony of this algorithm uses its own pheromone matrix and if this matrix is different from the matrixes of other colonies.

Multi-colony ACO algorithms have been originally designed to be applied together with multi-objective optimization (see Kawamura et al. [192]), however this approach also improves the behavior of the standard ACO. Multi-colony ACO algorithms are well suited for parallelization – each processor keeps a colony of ants and there is less information exchanged between colonies than would have been between groups of ants in the standard ACO approach. Most multi-colony parallel ACO algorithms are based on the decentralized approach – this model possesses some similarities with the *island model* of the parallel genetic algorithm. Designing a parallel multi-colony ACO we have to take under consideration the following parameters:

1. communication structure and neighborhood topology:
   (a) all-to-all topology (colonies have connection with each other),
   (b) ring topology (in the directed or undirected version),

   (c) hypercube topology (for p colonies, each colony has $\log p$ neighbors),
   (d) random topology (the neighbors of each colony are defined as a random structure *for each communication step*,

2. type of information exchanged:

   (a) solutions,
   (b) pheromone vectors,
   (c) pheromone matrix,

3. usage of information received from other colonies:

   (a) making comparison with its own elitist solutions,
   (b) adding it to the current generation,
   (c) adding pheromone vector or matrix obtained from another colony to its own pheromone matrix,

4. communication frequency.

Homogenous as well as heterogenous approaches are applied in the parallel multi-colony ACO algorithm implementation. In the homogenous approach all colonies make use of the same ACO parameters and heuristics. In the heterogenous approach parameters and heuristics in each colony are different.

Estimation of Distribution Algorithms. A class of *Estimation of Distribution Algorithms* (EDAs, also called Estimated Distribution Algorithms) was initially proposed by Mühlenbein and Paaß [192] in 1996. The method uses a population of individuals to estimate the distribution of the probability of each variable being kept by each individual. In the further part of algorithms this distribution is used to generate a new set of solutions that are hopefully the nearest to the global optimum. EDAs, as distinct from other evolutionary approaches, use neither crossover operators nor mutation operators.

An EDA algorithm can be parallelized on several levels:

1. *learning* (*or estimation*) *level*, on which an estimation of probability distribution is made (Lobo et al. [176], Mendiburu et al. [185]),
2. *simulation level*, on which sampling of the new individuals is executed (Mendiburu et al. [185], Oceanásek and Schwarz [200]),
3. *population level* (Ahn et al. [5]),
4. *fitness evaluation level.*

Most *learning approaches* possess an exponential computational complexity (e.g. if they use a Bayesian network). This kind of parallelization is the most time-consuming. The problem seems to be even worse because of incomplete data

presence, if it is necessary to resort to approximate algorithms. Parallelization on this level is still a major challenge. In the parallelization on the sampling level the generation of the new individuals is accomplished in a parallel fashion. However, there are cases where the problem representation is complex and sampling from the corresponding distributions is a hard task.

Population level parallelization consists in decentralizing the search process on the population level – a global population is defined virtually on a set of local subpopulations which interact with each other. One of the most popular approaches is the island framework. Parameters of information exchange in the island model are the same as defined in Section 2.3, in the description of the migration model of Evolution Strategies.

Parallel fitness evaluation is almost always based on the master-slave parallelization model. There are not any new approaches on this level compared to other population-based algorithms – the method used for the fitness function evaluation has to be fitted to the problem specificity, e.g. if the fitness function has the recurrent nature, it has to be transformed into the iterative form. On the other hand, the load balance between processors which execute the evaluation of the fitness has to be taken into consideration in the master-slave model.

The parallelization process can also be used on several levels concurrently as a hybrid parallelization. As we can see, most approaches proposed in the literature make use of parallelization on the simulation or the learning level. Only a few works use a different level of parallelism [5]. There is a lack of the low-level fitness evaluation module for EDAs, due to its strong relation with the problem formulation (see also similar problem for p-LSMs described in Section 2.1.1).

2.4. Remarks and conclusions

A great deal of parallel metaheuristic approaches from the literature presented in this chapter use multiple-walk parallelization based on hybridization of the search process or broadcast information about good solution space regions. In the literature there is almost nothing in the field of the parallel single-walk algorithms. Parallel algorithms used to solve job scheduling problems employ from 30 to 50% of time to determine the order of jobs, but only in a few cases some special properties of this problem are utilized. Apart from the approaches presented, there are many metaheuristics which have no parallel versions yet, e.g. artificial immune systems, beam search, music harmony optimization, simulated jumping, bee search, particle swarm optimization, etc. As we have mentioned above, the alternative approach to parallelization, rarely used, is to consider a problem specificity to build a low-level single-walk parallelization based on the special properties. Such an approach is shown in Part II of this book.

Chapter 3

Scheduling problems

This chapter addresses job scheduling problems together with their properties and models. These properties are independent of the computing environment (sequential, parallel). Some of them are new and original and they were designed for improving the efficiency of particular algorithms. Others, known from the literature, were applied for the first time to the parallel algorithm designing. Generally, scheduling problems allow us to model and analyze separate stages of the production systems (distinguished, single machine scheduling problems, see [128]) as well as production systems (flow shop, job shop problems). All the problems described in the sequel belong to the NP-hard class.

3.1. Basic notions and notation

Some elementary notions are used in mathematical model building of a job scheduling problem: a *job* and a *resource*. The job consists in executing a sequence of operations which need some resources. A number of data can be connected with the job: *due date* or *deadline*, possibility of breaking the job (*divisibility*), ways of operation execution (specific requirements of resources, alternative ways of execution), etc. Resources can be renewable (processor, machine, memory) or non-renewable (operational materials, natural resources) and dual-bounded (energy, capital). The features of the resources include: accessibility (in time windows), cost, amount, divisibility. All of these features have to be mathematically formalized by constructing a mathematical model of a problem.

Let $\mathcal{J} = \{1, 2, \ldots, n\}$ be a set of jobs which have to be executed by using a set of types of machines $M = \{1, 2, \ldots, m\}$. Each job i is a sequence of o_i operations $O_i = (l_{i-1}+1, l_{i-1}+2, \ldots, l_i)$, $l_i = \sum_{k=1}^{i} o_k$, $l_0 = 0$. Operations inside a job have to be executed in a defined technological order (in the defined sequence), i.e., an operation j has to be executed after having finished an operation $j-1$ execution

and before starting the execution of an operation $j+1$. A set of operations of a job i will be denoted by $\mathcal{O}_i$ for simplicity of notation. For each operation $j \in \mathcal{O}$, $\mathcal{O} = \bigcup_{i=1}^{n} \mathcal{O}_i$ the following terms are determined:

M_j – sequence of m_j subsets of machines which define alternative methods of operation execution; $M_j = (M_{1j}, M_{2j}, \ldots, M_{m_j,j}), M_{ij} \subseteq M$; an operation j needs a set of machines M_{ij} for its execution, where $1 \leq i \leq m_j$,

p_{ij} – time of execution of an operation j by the i-th method (i.e., on the i-th machine),

v_j – method of executing the operation (decision variable),

S_j – term of an operation execution beginning (decision variable),

C_j – term of an operation execution finishing, $C_j = S_j + p_{v_j j}$ if the operation cannot be broken.

In turn, for a job i the following terms are needed to be determined:

o_i – number of operations in the job,

r_i – the earliest possible term of the job execution beginning,

d_i – due date of the job execution finishing,

$\mathcal{S}_i$ – term of the job execution beginning, $\mathcal{S}_i = S_{l_{i-1}+1}$,

$\mathcal{C}_i$ – term of the job execution finishing, $\mathcal{C}_i = C_{l_i}$,

$\mathcal{L}_i$ – non-timeliness of the job execution finishing, i.e., being tardy or early, $\mathcal{L}_i = \mathcal{C}_i - d_i$,

$\mathcal{T}_i$ – tardiness of the job execution finishing, $\mathcal{T}_i = \max\{0, \mathcal{C}_i - d_i\}$,

$\mathcal{E}_i$ – earliness of the job execution finishing, $\mathcal{E}_i = \max\{0, e_i - \mathcal{C}_i\}$,

$f_i(t)$ – non-decreasing cost function connected with the job i execution finishing in a time $t \geq 0$,

$\mathcal{F}_i$ – a time of flow of the job i through the system, $\mathcal{F}_i = \mathcal{C}_i - r_i$,

$\mathcal{U}_i$ – unitary tardiness of the job i defined as

$$\mathcal{U}_i = \begin{cases} 0 & \text{if } \ C_i \leq d_i, \\ 1 & \text{otherwise.} \end{cases} \tag{3.1}$$

The majority of scheduling problems do not need to define all the above data and decision variables. Usually a minimal set of notions which is sufficient to describe the model is used. For example, if for each $j \in \mathcal{O}$ we have $m_j = 1$, $|M_{1j}| = 1$ it means that the problem has dedicated machines, therefore decision variables v_j do not undergo any choice. Then v does not occur in the model of the problem.

3.2. Taxonomy

To describe precisely the scheduling problem a three-field notation $\alpha|\beta|\gamma$ is applied. This notation was proposed in [123] and next developed in papers [169, 220].

It has three fields $\alpha|\beta|\gamma$ specifying the execution environment α, additional constraints β, and the objective function γ.

Here we propose an extended Graham notation which includes representations of hybrid systems (e.g. hybrid flow shop, see Janiak et al. [148]) or flexible systems with parallel machines (e.g. flexible job shop problem, see Bożejko et al. [30]). This kind of scheduling problems cannot be described by the original Graham notation. We propose to set the symbol α as a composition of three symbols $\alpha_3\alpha_2\alpha_1$ which have the following meaning. The symbol α_1 describes a finite number of machines in the system: 1, 2, ...; if this number is not specified then an empty symbol is put here which means any number of machine m. The symbol α_2 describes the method of jobs flowing through the system, where the following traditional ways are enhanced:

- F – flow shop in which all the jobs have the same technological path and they all have to be executed on all the machines; each machine needs to determine different sequence of input jobs,
- F^* – permutation flow shop, a model which has the same assumptions as F with an additional requirement that a sequence of job execution on all the machines has to be the same (compatible with the order of the sequence of jobs input into the system),
- J – job shop, in which jobs can have different (in terms of the number and the order of visiting machines) technological paths,
- G – general shop, in which each job is a single operation and technological relationship is given by a graph,
- O – open shop, in which all the operations of jobs have to be executed, but the technological order of operations inside the job is not specified.

The number of machines $\alpha_1 = 1$ implicates that both α_3 and α_1 symbols have to be empty. The symbol α_3 determines the mode of executing each operation. If α_3 is an empty symbol then we assume that for each operation a machine has been dedicated on which it will be executed, that is $m_j = 1$, $|M_{ij}| = 1$, $j \in \mathcal{O}$. Otherwise, we assume that $m_j \geq 1$, $|M_{ij}| = 1$, $i = 1, 2, \ldots, m_j$, $j \in \mathcal{O}$ and an operation can be executed on exactly one machine from a set of:

- P – identical parallel machines,
- Q – uniform machines, or
- R – non-uniform machines.

As we have already mentioned both α_3 and α_1 symbols can be empty, which means that any realization mode can be accepted, or (α_1 empty symbol) any (but fixed) number of machines can be used.

The symbol β determines the existence of additional assumptions and constraints, e.g. different release times (the earliest possible times of beginning job execution, r_i), existence of a partial technological order of job execution (*prec*), constraints *no wait*, *no store*, *no idle* (without time gaps), $p_{ij} = 1$ (all times are identical and equal 1), *pmtn* (jobs can be stopped and started again), etc.

The last parameter γ has the symbolic form of the criteria function. Two classes of this function occur in theory and practice of job scheduling, namely

$$f_{\max} = \max_{1 \le i \le n} f_i(\mathcal{C}_i) \tag{3.2}$$

and

$$\sum f_i = \sum_{i=1}^{n} f_i(\mathcal{C}_i), \tag{3.3}$$

where $f_i(t)$ are some non-decreasing functions. These classes include, among others, many frequent criteria from the practice, for example: the length of the schedule (makespan)

$$C_{\max} = \max_{1 \le i \le n} \mathcal{C}_i, \tag{3.4}$$

an average time of the job flow

$$\sum F_i = \frac{1}{n} \sum_{i=1}^{n} \mathcal{F}_i, \tag{3.5}$$

In the second case we may include a different weight of jobs $w_i \ge 0$ in the cost function $f_i(t) = w_i t$. For jobs with due dates d_i one can construct measures $f_i(t) = \max\{0, t - d_i\}$ or $f_i(t) = w_i \max\{0, t - d_i\}$. Therefore, we obtain

$$T_{\max} = \max_{1 \le i \le n} \mathcal{T}_i = \max_{1 \le i \le n} \max\{0, \mathcal{C}_i - d_i) \tag{3.6}$$

or the weighted sum of job tardiness

$$\sum w_i T_i = \sum_{i=1}^{n} w_i \mathcal{T}_i = \sum_{i=1}^{n} w_i \max\{0, \mathcal{C}_i - d_i\}. \tag{3.7}$$

In the further parts of this book we will concentrate mainly on the following criteria: makespan ($C_{\max}$), sum of job finishing times (C_{sum}, $\sum C_i$) and weighted

sum of job tardiness ($\sum w_i T_i$). The criteria cited above are known as typical in practice and they generate troubles during optimization (they are difficult).

3.3. Single machine scheduling problems

In this section, we present and discuss a fundamental case of the single machine problem as well as an industrial case (with using setup times) and a single machine problem case with earliness and tardiness penalties.

3.3.1. Overview

Single machine scheduling problems constitute a dominant class of optimization problems. There are a number of reasons for such a situation. They are easy to define and explain. Although weak usable in practice, single machine problems can be applied to analyze an allocated critical element of a production system. Single machine scheduling problems are the base of OPT (*Optimized Production Technology*, critical nest monitoring). Some of them can be almost directly transformed to the traveling salesman problem (TSP), offering the possibility to use a wide class of methods and approaches designed for TSP, probably the oldest combinatorial optimization problem (i.e., exponential-size neighborhoods searched in the polynomial time) to solve a scheduling problem under consideration.

Single machine problems can be met both as an element of the more complex, multi-machine systems (e.g. it can be a bottleneck element of this system), and as a stand-alone optimization problem. A solution is usually represented as a permutation, so it is easy to design exact or approximate solution algorithms because of their clarity. While the problem formulation is simple, solution methods are universal and they can be used to solve more complex systems, such as parallel shops, job shops or flexible scheduling problem based on the same concept.

3.3.2. Fundamental case

In the single machine total weighted tardiness problem ($TWTP$), a *set of jobs* $\mathcal{J} = \{1, 2, \ldots, n\}$ has to be processed without interruption on a single machine that can handle only one job at a time. All jobs become available for processing at time zero. Each job $i \in \mathcal{J}$ has an integer *processing time* p_i, a necessary finishing time called a *due date* d_i, and a *positive weight* w_i. For a given sequence of the jobs (earliest) completion time C_i, the *tardiness* $T_i = \max\{0, C_i - d_i\}$ and the cost $f_i(C_i) = w_i \cdot T_i$ of job $i \in \mathcal{J}$ can be computed. The goal is to find a job sequence which minimizes the sum of the costs given by $\sum_{i=1}^{n} f_i(C_i) = \sum_{i=1}^{n} w_i \cdot T_i$. Each schedule of jobs can be represented by a permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$.

The total cost is

$$F(\pi) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)}), \tag{3.8}$$

where $C_{\pi(i)}$ is a completion time of a job $\pi(i)$.

The single machine total weighted tardiness problem is denoted by $1||\sum_i w_i T_i$ in the literature and it belongs to the strongly NP-hard class (Lawler [162], Lenstra et al. [168]). A large number of studies have been devoted to the problem. Emmons [102] proposed several dominance rules that restrict the search process for an optimal solution. These rules are used in many algorithms. Enumerative algorithms that make use of dynamic programming and branch and bound (B&B) approaches to the problem were described by Fischer [105], Potts and Van Wassenhove [214] and Wodecki [268] (a parallel B&B algorithm). Other proposed algorithms were discussed and tested in a review paper by Abdul-Razaq et al. [3]. Although these algorithms constitute a significant improvement to the exhaustive search, they still remain laborious and are applicable only to relatively small problems (with the number of jobs not exceeding 50 for sequential algorithms and 120 jobs for parallel B&B algorithm [268]). The enumerative algorithms require considerable computer resources both in terms of computation times and the core storage. Thus, many algorithms have been proposed to find near optimal schedules in a reasonable time. These algorithms can be classified as construction and improvement methods.

The *construction methods* use dispatching rules to come up with a solution by fixing a job in a position at each step. Several constructive heuristics were described by Fischer [105] and in a review paper by Potts and Van Wassenhove [213]. Despite of being very fast, their quality is not good.

The *improvement methods* start from an initial solution and repeatedly try to improve the current solution by local changes. The interchanges are continued until a solution which cannot be improved is obtained. Such a solution is a local minimum. To increase the performance of local search algorithms, there are used metaheuristics like tabu search (Crauwels et al. [88], Bożejko, Grabowski and Wodecki [53]), simulated annealing (Potts and Van Wassenhove [213]), genetic algorithms (Crauwels et al. [88]), ant colony optimization (Den Basten et al. [96]). A very effective local search method was proposed by Congram et al. [81] and next improved by Grosso et al. [125]. The key aspect of the method is its ability to explore an exponential-size neighborhood in polynomial time, using a dynamic programming technique.

3.3.3. Setup times

We employ setup times in the problem formulation described above. These setups are taken from practice: they are encountered in real industrial applications to model times needed for servicing machines between operations execution. Let $\mathcal{J} = \{1, 2, \ldots, n\}$ be a set of n jobs for which we define a time of execution p_i, a weight of the cost function w_i and a deadline d_i for each job $i \in \mathcal{J}$ as in the previous section. Let s_{ij} be a setup time representing a time needed to prepare the machine for executing a job j after having completed a job i. Additionally, s_{0i} is a time required to prepare a machine for executing the first job i (at the beginning of the machine work). The problem considered consists in determining such a sequence of executing of jobs which minimizes the *sum of costs of tardiness*, i.e., $\sum w_i T_i$ where C_i $(i = 1, 2, \ldots, n)$ is the time of completing a job i, $T_i = \max\{0, C_i - d_i\}$ denotes a tardiness and $f_i(C_i) = w_i T_i$ represents a cost of tardiness of a job i.

The single machine total weighted tardiness problem with sequence-dependent setup times ($SDST$) is denoted in the literature as $1|s_{ij}|\sum w_i T_i$ and it is strongly NP-hard, as $1||\sum w_i T_i$ (with $s_{ij} = 0$) is strongly NP-hard (see Lenstra, Rinnoy Kan and Brucker [168]). To date the best construction heuristics for this problem has been the Apparent Tardiness Cost with Setups (ATCS – Lee, Bhaskaran and Pinedo [166]).

Many metaheuristics have been proposed, too. Tan et al. [250] presented a comparison of four methods of solving this problem: branch and bound, genetic search, random-start pair-wise interchange and simulated annealing. Gagné, Price and Gravel [110] compared the ant colony optimization algorithm with other heuristics. Cicirello and Smith [78] proposed benchmarks for the single machine total tardiness problem with sequence-dependent setups by generated 120 instances and applied stochastic sampling approaches: Limited Discrepancy Search (LDS), Heuristic-Biased Stochastic Sampling (HBSS), Value Biased Stochastic Sampling (VBSS), Value Biased Stochastic Sampling seeded Hill-Climber (VBSS-HS) and simulated annealing. The best goal function value obtained by their approaches was published in the literature and presented on the web site `http://www.ozone.ri.cmu.edu/benchmarks.html` as upper bounds of the benchmark problems. These upper bounds were next improved by Cicirello [79] by genetic algorithm, Lin and Ying [173] by tabu search, simulated annealing and genetic algorithm, and Liao and Juan [172] by the ant optimization. Bożejko and Wodecki [45] proposed a parallel metaheuristic for the problem under consideration improving the best known solution values for the benchmark instances of Cicirello and Smith [78].

The lower bound from the assignment problem

The linear *Assignment Problem* (AP) can be applied to determine the lower bound of the value of the sequence dependent setup times ($SDST$) problem solution. This problem can be formulated as follows

$$\min_{\pi\in\Phi_n}\sum_{i=1}^{k} c_{i\pi(i)}, \tag{3.9}$$

where $\pi \in \Phi_n$ is a permutation and elements of the matrix $[c_{i,j}]_{k\times k}$ are called *costs*. Next, we define a cost matrix by using setup times such that the solution of AP gives us a lower bound of the goal function.

Blocks in solutions

For the $SDST$ problem, each schedule of jobs can be represented by permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n-1), \pi(n))$ of the set of jobs $\mathcal{J}$. If $C_{\pi(i)}$ is the time of finishing a job $\pi(i) \in \mathcal{J}$ then the job is considered *on time* if it is completed before its due date ($C_{\pi(i)} \leq d_{\pi(i)}$), and *tardy* if completed after its due date (i.e., $C_{\pi(i)} > d_{\pi(i)}$).

Let

$$B_{ab} = (\pi(a), \pi(a+1), \ldots, \pi(b-1), \pi(b)) \tag{3.10}$$

be a subsequence ($1 \leq a < b \leq n$) of a permutation $\pi \in \Phi_n$. Therefore

$$WT(B_{ab}) = \sum_{i=a}^{b} w_{\pi(i)} \cdot T_{\pi(i)}, \tag{3.11}$$

is a *cost* and $L(B_{ab}) = C_{\pi(b)} + s_{\pi(b),\pi(b+1)}$ is a *length* B_{ab}, where

$$C_{\pi(j)} = s_{0,\pi(1)} + \sum_{i=1}^{j-1}(p_{\pi(i)} + s_{\pi(i),\pi(i+1)}) + p_{\pi(j)}, \quad i = 1, 2, \ldots, n. \tag{3.12}$$

By $\Phi(B_{ab}) \in \Phi_n$ we define a set of permutations which differ from π only in the sequence of elements in positions $a, a+1, \ldots, b$, that is,

$$\Phi(B_{ab}) = \{\beta \in \Phi : \ \pi(i) = \beta(i), \ \ i = 1, 2, \ldots, a-1, b+1, \ldots, n$$
$$\text{and } \ \mathcal{Y}(B_{ab}) = \mathcal{Y}(\beta^B)\}, \tag{3.13}$$

where $\mathcal{Y}(B_{ab}) = \{\pi(a), \pi(a+1), \ldots, \pi(b-1), \pi(b)\}$ is a set of elements belonging to a subsequence B_{ab}. For B_{ab}, let

$$F^{\min}(B_{ab}) \leq \min\{F(\delta) : \ \delta \in \Phi(B_{ab})\}, \tag{3.14}$$

$$F^{\max}(B_{ab}) \geq \max\{F(\delta) : \ \delta \in \Phi(B_{ab})\}. \tag{3.15}$$

That is why it is the lower (3.14) and upper bound (3.15) of value of the goal function F for permutations from a set $\Phi(B_{ab})$. Then, if a permutation $\gamma \in \Phi(B_{ab})$, so $F^{\min}(B_{ab}) \leq F(\gamma) \leq F^{\max}(B_{ab})$. Function F can be a cost WT or a length L of a permutation.

Subsequence B_{ab} is *θ-close to optimal subsequence* (we call it *θ-optimal*), if

$$F(B_{ab}) \in [F^{\min}(B_{ab}),\ F^{\min}(B_{ab}) + \theta \cdot (F^{\max}(B_{ab}) - F^{\min}(B_{ab}))] \tag{3.16}$$

where $\theta \in [0, 1]$ is a parameter determined experimentally.

For an *SDST* problem the goal is to minimize the sum of tardiness costs. Let B_{ab} be a subsequence (defined in (3.10)) of permutation $\pi \in \Phi$. The sum of tardiness costs can be found as

$$WT(\pi) = \sum_{i=1}^{n} w_{\pi(i)} T_{\pi(i)} = \sum_{i=1}^{a-1} w_{\pi(i)} T_{\pi(i)} + \sum_{i=a}^{b} w_{\pi(i)} T_{\pi(i)} + \\ + \sum_{i=b+1}^{n} w_{\pi(i)} T_{\pi(i)}. \tag{3.17}$$

The change of the schedule of elements in B_{ab} can generate a permutation with lower cost from the permutation π, if

1. the cost of execution of jobs from B_{ab} decreases, or
2. the length of B_{ab} gets shorter.

Next, we shall define two types of subsequences consisting of on-time and tardy jobs, which we call *$\mathcal{T}$-blocks* or *$\mathcal{D}$-blocks*, respectively. They are applied to eliminate 'worse' elements from neighborhoods determined in a scatter search algorithm. We will not consider permutations generated from π by changing an order of elements in any $\mathcal{T}$ or $\mathcal{D}$ block.

Blocks of on-time jobs

A subsequence $\mathcal{T}_{ab} = (\pi(a), \pi(a+1), \ldots, \pi(b-1), \pi(b))$ is a *$\mathcal{T}$-block* in permutation $\pi \in \Phi_n$, if

(a) each job from $\mathcal{T}_{ab}$ is on-time in a permutation π, that is,
$\forall j \in \mathcal{Y}(\mathcal{T}_{ab}),\ C_{\pi(j)} \leq d_{\pi(j)}$,

(b) due to the length L, subsequence $\mathcal{T}_{ab}$ is *θ-optimal*.

Further on, we shall present procedures of calculating the lower bound $L^{\min}(\mathcal{T}_{ab})$ and the upper bound $L^{\max}(\mathcal{T}_{ab})$ of the length $L(\beta)$ ($\beta \in \Phi_n(\mathcal{T}_{ab})$, that is, values which appear in the definition of *θ-optimality* (see 3.16).

Lower bound of the length $L^{\min}(\mathcal{T}_{ab})$

Execution times of jobs from $\mathcal{Y}(\mathcal{T}_{ab})$ and setup times influence the length $L(\mathcal{T}_{ab})$. Changing an order of jobs changes setup times but it does not have any influence on times of job finishing. Therefore, to minimize $L(\mathcal{T}_{ab})$ we have to determine an order of elements from $\mathcal{Y}(\mathcal{T}_{ab})$ due to setup times. A lower bound of the length $L^{\min}(\mathcal{T}_{ab})$ will be determined by solving an assignment problem.

By $h = b - a + 1$ we mean a number of jobs in $\mathcal{T}_{ab}$. For a job AP we define a matrix of costs C of size $(h+2) \times (h+2)$:

$$c_{ij} = s_{\pi(a+i-1)\pi(a+i)},\ i,j = 0,1,2,\ldots,h+1,\ i \neq j \tag{3.18}$$

and $c_{ii} = \infty,\ i = 0,1,2,\ldots,h+1,\ c_{0(h+1)} = \infty$.

If $a = 1$ (a job $\pi(a)$ is the first element in π), then $\pi(0) = 0$ and $c_{1,i} = s_{0\pi(i)}$. Similarly, if $b = n$, thus $\pi(n+1) = n+1$ and $c_{\pi(n)\pi(n+1)} = s_{\pi(n)0}$. If $C_{AP}^{\min}$ is a value of optimal solution of an Assignment Problem (with a cost matrix defined in (3.18)), then

$$LB^L(\mathcal{T}_{ab}) = C_{\pi(a-1)} + C_{AP}^{\min} + \sum_{i=a}^{b} p_{\pi(i)} \tag{3.19}$$

is the lower bound of the length $L(\beta^{\mathcal{T}})$ of any subsequence $\beta^{\mathcal{T}}$, where $\beta \in \Phi_n(\mathcal{T}_{ab})$. Thus, as the lower bound of the length of permutations from a set $\Phi_n(\mathcal{T}_{ab})$ we can take

$$L^{\min}(\mathcal{T}_{ab}) = LB^L(\mathcal{T}_{ab}) + \sum_{i=b+1}^{n} p_{\pi(i)} + s_{\pi(i)\pi(i+1)}. \tag{3.20}$$

Upper bound of the length $L^{\max}(\mathcal{T}_{ab})$

The definition of cost matrix $C = [c_{i,j}]_{(h+2)\times(h+2)}$ for the Assignment Problem has been presented in the previous section (*Lower bound of the length* $L^{\min}(\mathcal{T}_{ab})$). We change the weight of the edge $\{0, h+1\}$ in C by assigning $c_{0(h+1)} = -\infty$ and $c_{ii} = -\infty,\ i = 0,1,\ldots,h+1$. Next, we solve the 'modified' assignment problem (with a new matrix C) to determine the *maximal* solution. Let $C_{AP}^{\max}$ be the value of this solution. Then

$$UB^L(\mathcal{T}_{ab}) = C_{\pi(a-1)} + C_{AP}^{\max} + \sum_{i=a}^{b} p_{\pi(i)}, \tag{3.21}$$

is an upper bound of the length $L(\beta^{\mathcal{T}})$ of any subsequence $\beta^{\mathcal{T}}$, $\beta \in \Phi_n(\mathcal{T}_{ab})$. That is why we take

$$L^{\max}(\mathcal{T}_{ab}) = UB^L(\mathcal{T}_{ab}) + \sum_{i=b+1}^{n} p_{\pi(i)} + s_{\pi(i),\pi(i+1)} \tag{3.22}$$

as the upper bound of the length of permutation from the set $\Phi_n(\mathcal{T}_{ab})$.

Blocks of tardy jobs

The subsequence

$$\mathcal{D}_{ab} = (\pi(a), \pi(a+1), \ldots, \pi(b)) \ (1 \leq a < b \leq n) \tag{3.23}$$

is a $\mathcal{D}$-*block* in a permutation $\pi \in \Phi_n$, if

(a1) any job from $\mathcal{Y}(\mathcal{D}_{ab})$ moved in the first position in $\mathcal{D}_{ab}$ is tardy,

(b1) $\mathcal{D}_{ab}$ is θ-*optimal* (due to cost function WT),

(c1) $\mathcal{D}_{ab}$ is θ-*optimal* (due to the length (function L).

In the sequel, methods of determining constraints for the goal function value will be presented. They appear in the θ-*optimality* definition (point b1).

Lower bound of a cost $WT^{\min}(\mathcal{D}_{ab})$

The lower bound of a cost will be determined by solving an assignment problem. Elements of the matrix are determined as follows: let $\mathcal{D}_{ab}$ be a subsequence determined by (3.23) and $h = b - a + 1$ number of its elements. We construct a full graph $\mathcal{G}(\mathcal{D}_{ab}) = (\mathcal{V}, \mathcal{E})$ with weighted vertices and arcs. A set of vertices $\mathcal{V} = \{0, 1, 2, \ldots, h, h+1\}$ and arcs $\mathcal{E} = \{\{u, v\} : \ u, v \in \mathcal{V} \ \wedge \ u \neq v\}$. A vertex i $(i = 0, 1, 2, \ldots, h+1)$ is connected with a job $\pi(a-i-1)$ when 0 represents a job $\mathcal{D}_{ab}$ which is a predecessor: $\pi(a-1)$ (if $a = 1$ then $\pi(0) = 0$). On the other hand, $h+1$ represents a first job after $\mathcal{D}_{ab}$, a job $\pi(b+1)$ (if $b = n$ then $\pi(n+1) = n+1$). A weight of a vertex i is the time of executing a job $p_{\pi(a+i-1)}$, $(i = 1, 2, \ldots, h)$. A weight of the vertex 0 is 0 and a weight of a vertex $h+1$ is ∞. This graph has also weighted arcs. A weight of an arc $\{u, v\} \in \mathcal{E}$ equals the setup time between jobs $\pi(a+u-1)$ and $\pi(a+v-1)$ (therefore $s_{\pi(a+u-1)\pi(a+v-1)}$), and a weight of an arc $\{0, h+1\}$ is ∞. A model graph $\mathcal{G}(\mathcal{D}_{ab})$ for a 3-element subsequence is presented in Figure 3.1 (only some weights of arcs and vertices have been shown). Based on this kind of a graph we shall determine the earliest times of finishing jobs placed in particular positions in $\mathcal{D}_{ab}$.

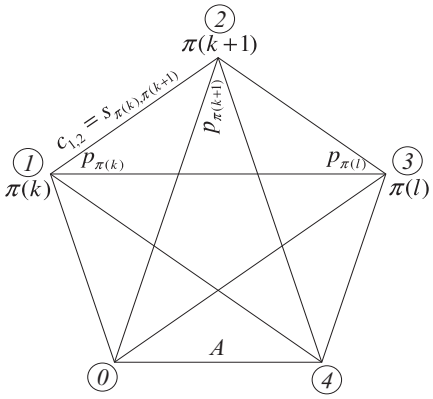


Fig. 3.1. An example of a graph with weighted vertices and arcs for a 3-element subsequence.

Let $t_{k,j}$ be the earliest possible time of finishing a job $k \in \mathcal{Y}(\mathcal{D}_{ab})$ which is in the position $j,\ a \leq j \leq b$ (then a job k is preceded by $j-a$ elements from $\mathcal{Y}(\mathcal{D}_{ab})$). From the construction of the graph $\mathcal{G}(\mathcal{D}_{ab})$ it follows that $t_{k,j}$ is a length of the shortest path from the vertex 0 to k consisting of *exactly* $j-a$ vertices (excluding vertices 0 and k). These paths and their lengths can be determined by using the Floyd-Warshall algorithm. Using these paths we can determine elements of the cost matrix $D = [d_{ij}]_{h\times h}$ for the assignment problem in the following way

$$d_{ij} = w_{\pi(a+i-1)} \cdot max\{0, t_{\pi(a+i-1),j} - d_{\pi(a+i-1)}\} \tag{3.24}$$

for $i \neq j,\ i,j = 1,2,\ldots,h$ and for $d_{jj} = \infty,\ j = 1,2,\ldots,h$ i $d_{1h} = \infty$. The value of d_{ij} is lower bound of the cost of job i, executed as the j-th one among all jobs from the set $\mathcal{Y}(\mathcal{D}_{ab})$.

Let $W_{AP}^{\min}(\mathcal{D}_{ab})$ be the value of the optimal (i.e., minimal) solution of an assignment problem with the cost matrix determined in (3.24). In the beginning part of Section 3.3.3 the method of calculation of the lower bound value for the length $LB^L(\mathcal{T}_{ab})$ was introduced (see (3.19)). Applying this procedure we can calculate $LB^L(\mathcal{D}_{ab})$ – the same approximation for the $\mathcal{D}$-block. Hence, the lower bound of finishing times of jobs which follow $\mathcal{D}_{ab}$ in π, C_i $(i = b+1,\ldots,n)$ can be determined from the following recurrent relations

$$C_{\pi(b+i)} = C_{\pi(b+i-1)} + s_{\pi(b+i-1)\pi(b+i)} + p_{\pi(b+i)},\ i = 2,3,\ldots,n, \tag{3.25}$$

where $C_{\pi(b+1)} = LB^L(\mathcal{T}_{ab}) + min\{s_{i\pi(b+1} :\ i \in \mathcal{Y}(\mathcal{D}_{ab})\} + p_{\pi(b+i)}$.

Therefore, the lower bound of costs of permutations from the set $\Phi_n(\mathcal{D}_{ab})$

$$WT^{\min}(\mathcal{D}_{ab}) = \sum_{i=1}^{a-1} w_{\pi(i)} \cdot max\{0, C_{\pi(i)} - d_{\pi(i)}\} + W_{AP}^{\min}(\mathcal{D}_{ab}) +$$

$$+ \sum_{i=b+1}^{n} w_{\pi(i)} \cdot max\{0, C_{\pi(i)} - d_{\pi(i)}\}. \tag{3.26}$$

The first sum is the cost of executing jobs $\pi(1), \pi(2), \ldots, \pi(a-1)$. The second component of the sum is a lower bound of cost of executing jobs from $\mathcal{D}_{ab}$. The last sum is the lower bound of execution of jobs $\pi(b+1), \pi(b+2), \ldots, \pi(n)$.

Upper bound of the cost $WT^{\max}(\mathcal{D}_{ab})$

On the basis of the graph (presented in the last section) with weighted vertices and arcs $\mathcal{G}(\mathcal{D}_{ab})$ we determine a cost matrix $D = [d_{ij}]_{h\times h}$ (due to (3.24)), where $d_{jj} = -\infty,\ j = 1, 2, \ldots, h$ and $d_{1h} = -\infty$. Next, we solve the modified assignment problem (with cost matrix D) determining a *maximal* solution. Let $W_{AP}^{\max}$ be the value of this solution.

Similarly as in the determination procedure of the upper bound of the length $L^{\max}(\mathcal{T}_{ab})$, we can calculate approximation of the length of a $\mathcal{D}$-block. Then the upper bound of finishing times of jobs from $\mathcal{D}_{ab}$ can be calculated from the following recurrent relation

$$C_{\pi(b+i)} = C_{\pi(b+i-1)} + s_{\pi(b+i-1)s\pi(b+i)} + p_{\pi(b+i)}, \quad i = 2, 3, \ldots, n, \tag{3.27}$$

where $C_{\pi(b+1)} = UB^{L}(\mathcal{D}_{ab}) + min\{s_{i\pi(b+1} : \ i \in \mathcal{Y}(\mathcal{D}_{ab})\} + p_{\pi(b+i)}$. The upper bound of costs of permutations from the set $\Phi_n(\mathcal{D}_{ab})$ is expressed by

$$WT^{\max}(\mathcal{D}_{ab}) = \sum_{i=1}^{a-1} w_{\pi(i)} \cdot max\{0, C_{\pi(i)} - d_{\pi(i)}\} + W_{AP}^{\max}(\mathcal{D}_{ab}) +$$

$$+ \sum_{i=b+1}^{n} w_{\pi(i)} \cdot max\{0, C_{\pi(i)} - d_{\pi(i)}\}. \tag{3.28}$$

The first sum is the cost of execution of jobs $\pi(1), \pi(2), \ldots, \pi(a-1)$. The second component of the sum is the upper bound of cost of jobs from the set $\mathcal{D}_{ab}$. The last sum is the upper bound of cost of jobs $\pi(b+1), \pi(b+2), \ldots, \pi(n)$. An algorithm of determining this upper bound has computational complexity $O(n^3)$.

Determining blocks in permutation

At the beginning the value of θ parameter (which appears in the definition of *θ-optimality*, see (3.16)) has to be experimentally determined. We will consider only blocks which contain at least 3 elements. So we are finding a subsequence

which fulfills condition (a) (or (a1)) and next we are checking all the remaining conditions.

Let a permutation $\pi \in \Phi_n$. We consider consecutively elements of permutation $\pi(1), \pi(2), \ldots, \pi(n)$ distinguishing two situations (cases): a current job is on-time or it is late.

Case 1. Let us assume that a job $\pi(1)$ is on-time in the permutation π. Then, it is a candidate to be the first element in a $\mathcal{T}$-*block*. If jobs $\pi(2)$ and $\pi(3)$ are also on-time then we are checking if subsequence $\mathcal{T}_{ab} = (\pi(1)\pi(2), \pi(3))$ fulfills constraint (b) in the definition of $\mathcal{T}$-*block*. If yes, we have the first 3-element block, which we would like to enhance by checking if after adding succeeding jobs we will also obtain a $\mathcal{T}$-*block*. If it is impossible to enhance the block by adding the next job then this job becomes the first one of the consecutive block. In turn, if a job $\pi(2)$ or $\pi(3)$ is not on-time, or jobs $\pi(1), \pi(2)$ and $\pi(3)$ are on-time but do not fulfill constraint (b) in the definition of $\mathcal{T}$-*block*, then the construction of the next block has to start with job $\pi(2)$, which is a candidate to be the first element in the block.

Case 2. Let us assume that a job $\pi(1)$ is late in a permutation π. Then, this job is a candidate to be the first one in a $\mathcal{D}$-block. Next, we proceed as in Case 1 (checking constraints (a), (b1) and (c1) for a $\mathcal{D}$-*block*).

In such a way we determine blocks (if they exist) in any permutation from the set of solutions Φ_n. The sequential computational complexity of the algorithm of block determination in permutation is $O(n^3)$. The existence of blocks and their number depend largely on the value of parameter θ.

3.3.4. Earliness/tardiness penalties

There are some types of manufacturing systems, called Just In Time (*JIT*), where costs are connected not only with executing a job too late, but also too early. Such a situation occurs especially when tasks are connected directly with the Web, e.g. routing, agents, similarity classification, etc. This induces formulation of many optimization problems with goal functions, where there is a penalty for both tardiness and earliness of a job. The problem of scheduling with earliness and tardiness (total weighted earliness/tardiness problem, *TWET*) is one of the most frequently considered in the literature. In this problem, each job from a set $\mathcal{J} = \{1, 2, \ldots, n\}$ has to be processed, without interruption, on a machine, which can execute at most one job at a time. By p_i we represent the execution time of a job $i \in \mathcal{J}$, and by e_i and d_i we mean the required earliest and latest moments of finishing the processing of a job. If scheduling of jobs is established and C_i is the moment of finishing a job i, then we call $E_i = \max\{0, e_i - C_i\}$ an *earliness* and $T_i = \max\{0, C_i - d_i\}$ a *tardiness*. The expression $u_i E_i + w_i T$ is the *cost* of executing a job, where u_i and w_i ($i \in \mathcal{J}$) are nonnegative coefficients of a goal

function. The problem consists in minimizing the sum of costs of jobs, that is, in finding a job sequence $\pi^* \in \Phi_n$ such that for the goal function

$$F(\pi) = \sum_{i=1}^{n} (u_{\pi(i)} E_{\pi(i)} + w_{\pi(i)} T_{\pi(i)}), \quad \pi \in \Phi_n, \tag{3.29}$$

we have

$$F(\pi^*) = \min_{\pi \in \Phi_n} F(\pi). \tag{3.30}$$

This problem is represented by $1||\sum (u_i E_i + w_i T_i)$ in the literature and it belongs to a strongly NP-hard class (if we assume $u_i = 0$, $i = 1, 2, \ldots, n$, we will obtain a strongly NP-hard problem $1||\sum w_i T_i$, Lawler [162] and Lenstra et al. [168]). Baker and Scudder [17] proved that there can be an idle time in an optimal solution (jobs need not to be processed directly one after another), that is $C_{\pi(i+1)} - p_{\pi(i+1)} \geq C_{\pi(i)}$, $i = 1, 2, \ldots, n-1$. Solving the problem amounts to establishing a sequence of jobs and their starting times. Hoogeven and van de Velde [138] proposed an algorithm based on the branch and bound method. Because of the computation time growing exponentially, this algorithm can be applied only to solve instances where the number of jobs is not greater than 20. Therefore, in practice almost always approximate algorithms are used. The best ones are based on artificial intelligence methods. Calculations are performed in two stages.

- Determining the scheduling of jobs (with no idle times).
- Establishing optimal starting times of jobs.

There is an algorithm in the paper of Wan and Yen [262] based on this scheme – a tabu search algorithm is used to determine a schedule. Bożejko and Wodecki [57] proposed a parallel coevolutionary algorithm for the problem under consideration.

Block properties

For the *TWET-no-idle* problem, each schedule of jobs can be represented by permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ of elements of the set of jobs $\mathcal{J}$. Let Φ_n denote the set of all such permutations. The total cost $\pi \in \Phi_n$ is $F(\pi) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)})$, where $C_{\pi(i)}$ is the completion time of the job $\pi(i), C_{\pi(i)} = \sum_{j=1}^{i} p_{\pi(j)}$. The job $\pi(i)$ is considered *early* if it is completed before its earliest moment of finishing ($C_{\pi(i)} < e_{\pi(i)}$), *on time* if $e_{\pi(i)} \leq C_{\pi(i)} \leq d_{\pi(i)}$, and *tardy* if the job is completed after its due date (i.e., $C_{\pi(i)} > d_{\pi(i)}$).

Each permutation $\pi \in \Phi_n$ is decomposed into subpermutations (subsequences of jobs) $B = (B^1, B^2, \ldots, B^v)$ called *blocks* in π, where

1. $B^i = (\pi(a_i), \pi(a_i+1), \ldots, \pi(b_i-1), \pi(b_i))$, and
   $a_i = b_{i-1}+1, \;\; 1 \le i \le v, \;\; a_0 = 0, \; b_v = n.$

2. All the jobs $j \in B^i$ satisfy the following conditions:
   $$e_j > C_{\pi(b_i)}, \tag{C1}$$
   $$e_j \le C_{\pi(b_{i-1})} + p_j \text{ and } d_j \ge C_{\pi(b_i)}, \tag{C2}$$
   $$d_j < C_{\pi(b_{i-1})} + p_j. \tag{C3}$$

3. B_i are maximal subsequences of π in which all the jobs satisfy either Condition $C1$ or Condition $C2$ or Condition $C3$.

By definition, there exist three types of blocks implied by either C1 or C2 or C3. To distinguish them, we will use the *E-block*, *O-block* and *T-block* notions respectively. For any block Υ in a partition B of permutation $\pi \in \Phi_n$, let

$$F_\Upsilon(\pi) = \sum\nolimits_{i \in \Upsilon} (u_i E_i + w_i T_i). \tag{3.31}$$

Therefore, the value of a goal function

$$F(\pi) = \sum\nolimits_{i=1}^{n} (u_i E_i + w_i T_i) = \sum\nolimits_{\Upsilon \in B} F_\Upsilon(\pi). \tag{3.32}$$

If Υ is a T-block, then every job inside is early. Therefore, an optimal sequence of the jobs within Υ of the permutation π (that is minimizing $F_\Upsilon(\pi)$) can be obtained, using the well-known Weighted Shortest Processing Time (*WSPT*) rule, proposed by Smith [229]. The *WSPT* rule creates an optimal sequence of jobs in a non-increasing order of the ratios w_j/p_j. Similarly, if Υ is an E-block, then an optimal sequence of the jobs within it can be obtained, using the Weighted Longest Processing Time (*WLPT*) rule which creates a sequence of jobs in a non-decreasing order of the ratios u_j/p_j. Partition B of the permutation π is *ordered* if there are jobs in the *WSPT* sequence in any T-block, and if there are jobs in the *WLPT* sequence in any E-block.

Theorem 3.1 ([53]). *Let Υ be an ordered partition of a permutation $\pi \in \Phi_n$ into blocks. If $\beta \in \Phi_n$ and $F(\beta) < F(\pi)$, so at least one job of some block of π has been moved before the first or after the last job of this block in the permutation β.*

Note that Theorem 3.1 provides the necessary condition for obtaining a permutation β from π such that $F(\beta) < F(\pi)$. Let $B = (B^1, B^2, \ldots, B^v)$ be an ordered partition of the permutation $\pi \in \Phi_n$ into blocks. If a job $\pi(j) \in B^i$ ($B^i \in B$), then moves which can improve the goal function value consist in reordering a job $\pi(j)$ before the first or after the last job of this block. Let N_j^{bf} and N_j^{af} be sets

of such moves ($N_j^{bf} = \emptyset$ for $j \in B^1$ and $N_j^{af} = \emptyset$ for $j \in B^v$). Therefore, the neighborhood of the permutation $\pi \in \Phi_n$,

$$N(\pi) = \bigcup_{j=1}^{n} N_j^{bf} \cup \bigcup_{j=1}^{n} N_j^{af}. \tag{3.33}$$

As computational experiments show, the size of the neighborhood defined in (3.33) is half that of the neighborhood of all the insert moves.

3.4. Flow shop problems

We can see the process of jobs flowing through machines (processors) in many practical problems of scheduling: in computer systems as well as in production systems. Thus the flow shop scheduling problem represents a wide class of possible applications, depending on the cost function definition. For each of them, an corresponding discrete model has to be constructed and analyzed. Some of them (e.g. with the makespan criterion and with total weighted tardiness cost function) have got a special elimination-criteria (so-called *block properties*) which significantly speed up the calculation, especially in the multithread computing environment.

3.4.1. Formulation of problems

The problem has been introduced as follows. There are n jobs from a set $\mathcal{J} = \{1, 2, \ldots, n\}$ to be processed in a production system having m machines, indexed by $1, 2, \ldots, m$, organized in the line (sequential structure). A single job reflects one final product (or sub product) manufacturing. Each job is performed in m subsequent stages, in a way common to all the tasks. The stage i is performed by a machine i, $i = 1, 2, \ldots, m$. Each job $j \in \mathcal{J}$ is split into a sequence of m operations $O_{1j}, O_{2j}, \ldots, O_{mj}$ performed on machines. The operation O_{ij} reflects processing of job j on machine i with processing time $p_{ij} > 0$. Once started the job cannot be interrupted. Each machine can execute at most one job at a time, each job can be processed on at most one machine at a time.

The flow shop problem with makespan criterion

The sequence of loading jobs into a system is represented by a permutation $\pi = (\pi(1), \ldots, \pi(n))$ of elements of the set $\mathcal{J}$. The optimization problem is to find the optimal sequence π^* so that

$$C_{\max}(\pi^*) = \min_{\pi \in \Phi_n} C_{\max}(\pi) \tag{3.34}$$

where $C_{\max}(\pi)$ is the makespan for a permutation π and Φ_n is the set of all permutations of elements of the set $\mathcal{J}$. Denoting by C_{ij} the completion time of job j on machine i we have $C_{\max}(\pi) = C_{m,\pi(n)}$. Values C_{ij} can be found by using either the recursive formula

$$C_{i\pi(j)} = \max\{C_{i-1,\pi(j)}, C_{i,\pi(j-1)}\} + p_{i\pi(j)}, \tag{3.35}$$

$i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$, with initial conditions $C_{i\pi(0)} = 0$, $i = 1, 2, \ldots, m$, $C_{0\pi(j)} = 0$, $j = 1, 2, \ldots, n$, or a non-recursive one

$$C_{i\pi(j)} = \max_{1=j_0 \le j_1 \le \ldots \le j_i = j} \sum_{s=1}^{i} \sum_{k=j_{i-1}}^{j_i} p_{s\pi(k)}. \tag{3.36}$$

Computational complexity of (3.35) is $O(mn)$, whereas for (3.36) it is

$$O(\binom{j+i-2}{i-1}(j+i-1)) = O(\frac{(n+m)^{n-1}}{(n-1)!}). \tag{3.37}$$

In practice the former formula has been commonly used. It should be noticed that the problem of transforming sequential algorithm for scheduling problems into parallel one is nontrivial because of the strongly sequential character of computations carried out using (3.35) and other known scheduling algorithms.

Johnson [150] gave an $O(n \log n)$ algorithm for $F|2|C_{\max}$ and Garey et al. [115] shown that $F|3|C_{\max}$ is strongly NP-hard. The best available branch and bound algorithms are these of Ignall, Schrage [140], Lageweg, Lenstra & Rinnooy Kan [161] and Grabowski [120]. Their performance is not entirely satisfactory though they experience difficulty in solving instances with 20 jobs and 5 machines.

Various serial and parallel local search methods are available for the permutation flow shop problem. Tabu search algorithms were proposed by Taillard [243], Reeves [216], Nowicki and Smutnicki [196], Grabowski and Wodecki [118]. Simulated annealing algorithms were proposed by Osman, Potts [203], Ogbu and Smith [201] and Ishibushi, Misaki and Tanaka [142]. Reeves [217] proposed a genetic algorithm which uses the reorder crossover. Bożejko and Wodecki [38] applied this method in the parallel path-relinking method used to solve the flow shop scheduling problem. Bożejko and Wodecki also proposed a parallel scatter search [40] and parallel tabu search [58] method for this problem. Bożejko and Pempera [41] presented a parallel tabu search algorithm for the permutation flow shop problem of minimizing the criterion of the sum of job completion times. Bożejko and Wodecki also proposed a simulated annealing algorithm for the flow shop problem with $C_{\max}$ [63] and C_{sum} [59] criterion. Bożejko, Hejducki and Wodecki [42] proposed the fuzzy blocks conception in application to the genetic

algorithm for this problem. Bożejko and Wodecki [43] proposed applying multi-moves in parallel genetic algorithm for the flow shop problem. The theoretical properties of these multi-moves were considered by Bożejko and Wodecki in papers [46, 48, 49]. A survey of single-walk parallelization methods of the cost function calculation and neighborhood searching for the flow shop problem can be found in Bożejko [62].

In the batching systems (all jobs are available at the beginning) a rating is made by a maximal flow time (makespan) or an average flow time (equivalent to $\sum C_j$). Thus these two problems are most interesting from the practical point of view. The minimal makespan maximizes simultaneously the utilization rate of the machine park; $\sum C_j$ minimizes the volume of the work in progress.

The flow shop problem with $C_{\max}$ criterion

The objective is to find a schedule which minimizes the sum of job completion times. The problem is denoted by $F||C_{\max}$. There are plenty of good heuristic algorithms for solving $F||C_{\max}$ flow shop problem, with the objective of minimizing maximal job completion times. Due to the special properties (blocks of a critical path, see previous section and [121]) it is regarded as an easier one than a problem with objective $C_{\max}$. Unfortunately, there are no similar properties (which can speed up computations) for the $F||C_{sum}$ flow shop problem. Constructive algorithms (LIT and SPD from [264]) possess low efficiency and can only be applied to a limited range. There is a hybrid algorithm in [215], consisting of elements of tabu search, simulated annealing and path relinking methods. The results of this algorithm, applied to Taillard benchmark tests [243] are the best known ones in the literature nowadays. A theoretical analysis of the flow shop problem with the mean completion time criterion, which is a derivative of the criterion considered here, was made by Smutnicki [232, 233].

The flow shop problem with the criterion of the sum of job completion times can be formulated using notations from the previous paragraph. We wish to find a permutation $\pi^* \in \Phi_n$ such that

$$C_{\max}(\pi^*) = \min_{\pi \in \Phi_n} C_{\max}(\pi), \text{ where } C_{\max}(\pi) = \sum_{j=1}^{n} C_{m\pi(j)}. \tag{3.38}$$

The formula $C_{i\pi(j)}$ denotes the time required to complete the j-th job on the machine i in the processing order given by the permutation π. The completion time of job $\pi(j)$ on machine m can be found by applying the same formulas (3.35) or (3.36) as in the problem with a makespan criterion.

3.4.2. Models

Values C_{ij} from equations (3.35) and (3.36) can also be determined by means of a graph model of the flow shop problem. For a given sequence of job execution $\pi \in \Phi_n$ we create a graph $G(\pi) = (M \times N,\ F^0 \cup F^*)$, where $M = \{1, 2, \ldots, m\}$, $N = \{1, 2, \ldots, n\}$.

$$F^0 = \bigcup_{s=1}^{m-1} \bigcup_{t=1}^{n} \{((s,t),(s+1,t))\} \tag{3.39}$$

is a set of technological arcs (vertical) and

$$F^* = \bigcup_{s=1}^{m} \bigcup_{t=1}^{n-1} \{((s,t),(s,t+1))\} \tag{3.40}$$

is a set of sequencing arcs (horizontal).

Arcs of the graph $G(\pi)$ have no weights, but each vertex (s,t) has as weight $p_{s\pi(t)}$. A time C_{ij} of finishing a job $\pi(j)$, $j = 1, 2, \ldots, n$ on machine i, $i = 1, 2, \ldots, m$ equals the length of the longest path from vertex (1,1) to vertex (i,j) including the weight of the last one. A sample 'mesh' graph $G(\pi)$ is shown in Figure 3.2. The mesh is always the same, vertices weights depend on the π. For the $F||C_{\max}$ problem the value of the criterion function for fixed sequence π equals the length of the critical path in the graph $G(\pi)$. For the $F||C_{\text{sum}}$ problem the value of the criterion function is the sum of lengths of the longest paths which begin from vertex $(1,1)$ and ends on vertices $(m,1),(m,2),\ldots,(m,n)$.

The graph $G(\pi)$ is also strongly connected with formulas (3.35) and (3.36) of completion times C_{ij} calculation. By using formula (3.35), it is enough to generate consecutive vertices, column after column (or row after row) taking in the vertex (i,j), connected with the C_{ij}, a greater value from the left vertex, $C_{i,j-1}$, and from the upper one, $C_{i-1,j}$, and adding p_{ij} to it. Such a procedure generates the longest path in the graph $G(\pi)$ in time $O(nm)$. Formula (3.36) can also be presented as the longest path generation algorithm, but its conception is based on the all horizontal sub-paths generation and its computational complexity is exponential.

3.4.3. Properties

The longest path in graph $G(\pi)$ for a solution π of the flow shop problem defined in Section 3.4.2 is called a *critical path* with respective π. Its length is $C_{\max}(\pi)$. The critical path is decomposed into subsequences B_1, $B_2, \ldots, B_m$ called *blocks* in π, where

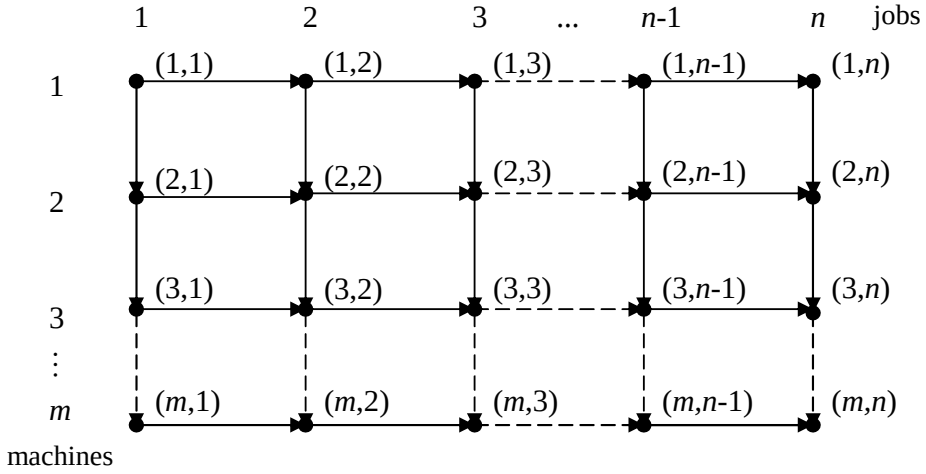


Fig. 3.2. Graph $G(\pi)$ (from Bożejko et al. [35]).

a) $B_k = (\pi(f_k), \pi(f_k+1), \ldots, \pi(l_k-1), \pi(l_k)), f_k \leq l_k, f_1 = 1, l_k = k$ and $\pi(l_k) = \pi(f_{k+1})$, $k = 1, 2, \ldots, m-1$,

b) B_k contains operations processed on the same machine, for $k = 1, 2, \ldots, m$,

c) two consecutive blocks contain operations processed on different machines.

In other words, the block is a maximal subsequence of the critical path which contains operations processed on the same machine. Operations $\pi(f_k)$ and $\pi(l_k)$ in B_k are called the *first* and the *last* ones, respectively.

Theorem 3.2 ([120]). *Let $G(\pi)$ be a graph with blocks B_k, $k = 1, 2, \ldots, m$. If the graph $G(\omega)$ has been obtained from $G(\pi)$ by the interchange of jobs and if $C_{\max}(\omega) < C_{\max}(\pi)$, then in $G(\omega)$:*

(i) at least one job $j \in B_k$ precedes job $\pi(f_k)$, for some $k = 2, \ldots, m$, or

(ii) at least one job $j \in B_k$ succeeds job $\pi(l_k)$, for some $k = 1, 2, \ldots, m-1$.

This theorem provides the necessary condition for obtaining a permutation such that $C_{\max}(\omega) < C_{\max}(\pi)$. We need to move jobs from the set B_k before the first job $\pi(f_k)$, and jobs from the set B_k after the last $\pi(l_k)$, $k = 1, 2, \ldots, m$ to obtain a permutation ω from a permutation π such that $C_{\max}(\omega) < C_{\max}(\pi)$.

For a job $j \in B_k \backslash \{\pi(f_k)\}$ let

$$\Delta_k^+(j) = \begin{cases} p_{j,k-1} - p_{\pi(f_k),k-1}, & j \neq \pi(l_k), \\ p_{j,k-1} - p_{\pi(f_k),k-1} + p_{\pi(l_k-1),k+1} - p_{j,k+1}, & j = \pi(l_k), \end{cases} \tag{3.41}$$

and for $j \in B_k \backslash \{\pi(l_k)\}$

$$\Delta_k^-(j) = \begin{cases} p_{j,k+1} - p_{\pi(l_k),k+1}, & j \neq \pi(f_k), \\ p_{\pi(f_k+1),k-1} - p_{j,k-1} + p_{j,k+1} - p_{\pi(l_k),k+1}, & j = \pi(f_k), \end{cases} \tag{3.42}$$

where $k = 1, 2, \ldots, m$ and $p_{\pi(i)j} = 0$, $i > n$, $j < 1$ or $j > k$.

Theorem 3.3 ([120]). *For each $\pi \in \Phi_n$, if β is the permutation obtained from π by moving job j, ($j \in B_k$) before the first or after the last job in block B_k, then we obtain*

$$C_{\max}(\beta) \geq C_{\max}(\pi) + \Delta_k^+(j) \;\; or \;\; C_{\max}(\beta) \geq C_{\max}(\pi) + \Delta_k^-(j). \tag{3.43}$$

By moving job $j \in B_k$ before $\pi(f_k)$ or after $\pi(l_k)$ in π, we generate permutation β and the lower bound on the value $C_{\max}(\beta)$ is $\geq C_{\max}(\pi) + \Delta_k^-(j)$ or $C_{\max}(\pi) + \Delta_k^+(j)$. Thus, the values $\Delta_k^-(j)$ and $\Delta_k^+(j)$ can be used to decide which job should be moved.

3.4.4. Transport times

The flow shop problem with transport times can be defined as follows. For each job there is defined the transport time of jobs between machines. Variable t_{ij} determines the transport time of a job j from machine i to machine $i+1$, $i = 1, \ldots, m-1$. A case $t_{ij} \geq 0$ has a natural practical justification and it requires no commentary. A case $t_{ij} < 0$ means the permission for 'overlapping' of subsequent job operations, or the start of the next job operation with some time delay compared to the start of the current operation and before its completion.

Assuming that the sequence of carrying out jobs is determined by a permutation π in a permutational flow shop problem with transport, the times of job completion can be determined on the basis of the following conditions

$$C_{i\pi(j)} \geq C_{i\pi(j-1)} + p_{i\pi(j)}, \;\; i = 1, 2, \ldots, m, \; j = 1, 2, \ldots, n, \tag{3.44}$$

$$C_{i\pi(j)} \geq C_{i-1,\pi(j)} + p_{i\pi(j)} + t_{i-1,\pi(j)}, \;\; i = 1, 2, \ldots, m, \; j = 1, 2, \ldots, n, \tag{3.45}$$

which let us obtain the recurrent formula (for $C_{\max}$ and C_{sum} criteria)

$$C_{i\pi(j)} = \max\{C_{i,\pi(j-1)}, C_{i-1,\pi(j)} + t_{i-1,\pi(j)}\} + p_{i\pi(j)}, \tag{3.46}$$

for $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$, where $\pi(0) = 0$, but for negative t_{ij} there should be $C_{i0} = 0$, $i = 1, 2, \ldots, n$, $C_{01} = 0$, $C_{0j} = -\infty$, $j = 2, 3, \ldots, m$.

3.5. Job shop problems

Job shop scheduling problems follow from many real-world cases, which means that they have good practical applications as well as industrial significance.

3.5.1. Problem definition

Let us consider a set of jobs $\mathcal{J} = \{1, 2, \ldots, n\}$, a set of machines $M = \{1, 2, \ldots, m\}$ and a set of operations $\mathcal{O} = \{1, 2, \ldots, o\}$. The set $\mathcal{O}$ is decomposed into subsets connected with jobs. A job j consists of a sequence of o_j operations indexed consecutively by $(l_{j-1}+1, l_{j-1}+2, \ldots, l_j)$ which have to be executed in this order, where $l_j = \sum_{i=1}^{j} o_i$ is the total number of operations of the first j jobs, $j = 1$, $2, \ldots, n$, $l_0 = 0$, $\sum_{i=1}^{n} o_i = o$. An operation i has to be executed on machine $v_i \in M$ without any idleness in time $p_i > 0$, $i \in \mathcal{O}$. Each machine can execute at most one operation at a time. A feasible solution constitutes a vector of times of the operation execution beginning $S = (S_1, S_2, \ldots, S_o)$ such that the following constraints are fulfilled

$$S_{l_{j-1}+1} \geq 0, \quad j = 1, 2, \ldots, n, \tag{3.47}$$

$$S_i + p_i \leq S_{i+1}, \quad i = l_{j-1} + 1, \quad l_{j-1} + 2, \ldots, l_j - 1, \quad j = 1, 2, \ldots, n, \tag{3.48}$$

$$S_i + p_i \leq S_j \quad \text{or} \quad S_j + p_j \leq S_i, \quad i, j \in O, \quad v_i = v_j, \quad i \neq j. \tag{3.49}$$

Certainly, $C_j = S_j + p_j$. An appropriate criterion function has to be added to the above constraints. The most frequent are the following two criteria: minimization of the time of finishing all the jobs and minimization of the sum of job finishing times. From the formulation of the problem we have $\mathcal{C}_j \equiv C_{l_j}$, $j \in \mathcal{J}$.

The first criterion, the time of finishing all the jobs

$$C_{\max}(S) = \max_{1 \leq j \leq n} C_{l_j}, \tag{3.50}$$

corresponds to the problem denoted as $J||C_{\max}$ in the literature. The second criterion, the sum of job finishing times

$$C(S) = \sum_{j=1}^{n} C_{l_j}, \tag{3.51}$$

corresponds to the problem denoted as $J||\sum C_i$ in the literature.

Both problems described are strongly NP-hard and although they are similarly modelled, the second one is found to be harder because of the lack of some specific properties (so-called block properties, see [196]) used in optimization of execution time of solution algorithms. Because of NP-hardness of the problem heuristics and metaheuristics are recommended as 'the most reasonable' solution methods. The majority of these methods refer to the makespan minimization. We mention here a few recent studies: Jain, Rangaswamy, and Meeran [145]; Pezzella and Merelli [206]; Grabowski and Wodecki [119]; Nowicki and Smutnicki [198]; Bożejko and Uchroński [32]. Smutnicki and Tyński [231] proposed a new crossover operator

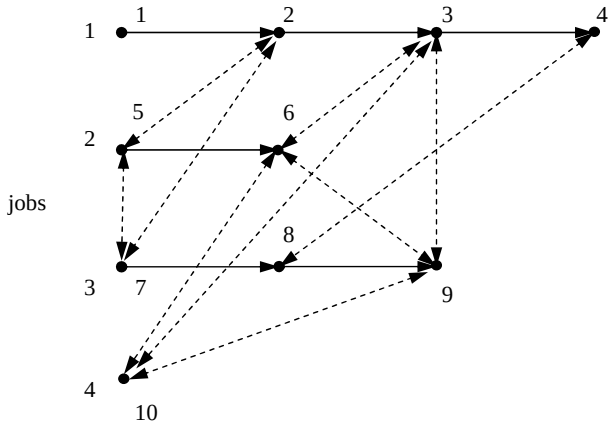


Fig. 3.3. An example of disjunctive graph for the job shop problem.

for the job shop problem used in a genetic algorithm. Also heuristics algorithms based on dispatching rules are proposed in papers of Holthaus and Rajendran [137], Bushee and Svestka [69] for the problem under consideration. For the other regular criteria such as the total tardiness there are proposed metaheuristics based on various local search techniques: simulated annealing [133], [263], tabu search [14] and genetic search [182].

3.5.2. Models and properties

The most commonly used models of job shop scheduling problems are based on the disjunctive or the combinatorial approaches. Both these models are presented in this section.

Disjunctive model

The disjunctive model is most commonly used, however it is very unpractical from the point of view of efficiency (and computational complexity). It is based on the notion of disjunctive graph $G = (O, U \cup V)$. This graph has a set of vertices O which represent operations, a set of conjunctive arcs (directed) which show technological order of operation execution

$$U = \bigcup_{j=1}^{n} \bigcup_{i=l_{j-1}+1}^{l_j-1} \{(i, i+1)\} \tag{3.52}$$

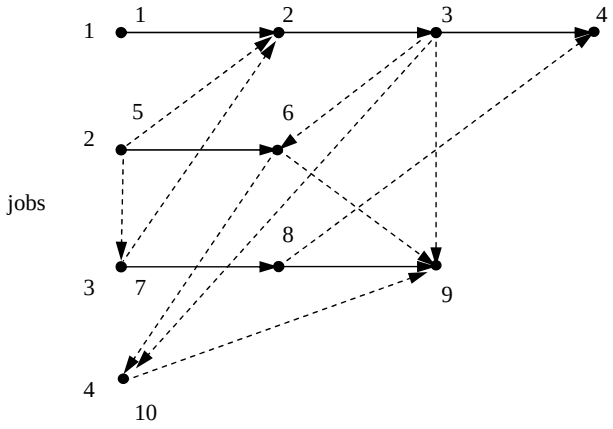


Fig. 3.4. An example of the graph $G(W)$ for the job shop problem.

and the set of disjunctive arcs (non-directed) which show possible schedule of operations execution on each machine

$$V = \bigcup_{i,j \in O, i \neq j, v_i = v_j} \{(i,j),(j,i)\}. \tag{3.53}$$

An example of the disjunctive graph is presented on Figure 3.3 (numbers near vertices are operation numbers, jobs are placed in rows and connected by solid arrows; disjunctive arcs are drawn as broken lines). Disjunctive arcs $\{(i,j),(j,i)\}$ are in fact pairs of directed arcs with inverted directions, which connect vertices i and j.

A vertex $i \in O$ has a weight p_i which equals the time of execution of operation O_i. Arcs have the weight zero. A choice of exactly one arc from the set $\{(i,j),(j,i)\}$ corresponds to determining a schedule of operations execution – 'i before j' or 'j before i'. A subset $W \subset V$ consisting of exclusively directed arcs, at most one from each pair $\{(i,j),(j,i)\}$, we call a *representation* of disjunctive arcs. Such a representation is complete if all the disjunctive arcs have determined direction. A complete representation, defining a precedence relation of jobs execution on the same machine, generates one solution, not always feasible, if it includes cycles. A feasible solution is generated by a complete representation W such that the graph $G(W) = (O, U \cup W)$ is acyclic (see Figure 3.4). For a feasible schedule values S_i of the vector of operations execution starting times $S = (S_1, S_2, \ldots, S_o)$ can be determined as a length of the longest path incoming to the vertex i (without p_i). As the graph $G(W)$ includes o vertices and $O(o^2)$ arcs, therefore determining the value of the cost function for a given representation W takes the time $O(o^2)$ by using Bellman algorithm of paths in graphs determination.

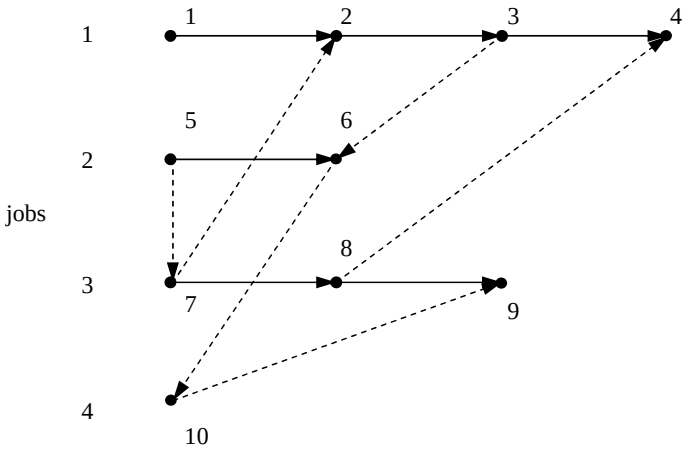


Fig. 3.5. An example of the $G(\pi)$ graph of combinatorial model for the job shop problem.

Combinatorial model

In the case of many applications a combinatorial representation of a solution is better than a disjunctive model for the job shop problem. The presented model follows that of Smutnicki [234]. It is void of redundance, characteristic of the disjunctive graph, that is, the situation where many disjunctive graphs represent the same solution of the job shop problem. A set of operations $\mathcal{O}$ can be decomposed into subsets of operations executed on a single, determined machine $k \in M$, $M_k = \{i \in \mathcal{O} : v_i = k\}$ and let $m_k = |M_k|$. The schedule of operations execution on a machine k is determined by a permutation $\pi_k = (\pi_k(1), \pi_k(2), \ldots, \pi_k(m_k))$ of elements of the set M_k, $k \in M$, where $\pi_k(i)$ means such an element from M_k which is in position i in π_k. Let $\Phi_n(M_k)$ be a set of all permutations of elements of M_k. A schedule of operations execution on all machines is defined as $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$, where $\pi \in \Phi_n$, $\Phi_n = \Phi_n(M_1) \times \Phi_n(M_2) \times \ldots \times \Phi_n(M_m)$. For a schedule π we create a directed graph (digraph) $G(\pi) = (\mathcal{O}, U \cup E(\pi))$ with a set of vertices $\mathcal{O}$ and a set of arcs $U \cup E(\pi)$), where U is a set of constant arcs representing the technological order of operations execution inside a job, and a set of arcs representing an order of operations execution on machines is defined as

$$E(\pi) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k - 1} \{(\pi_k(i), \pi_k(i+1))\} \tag{3.54}$$

Each vertex $i \in \mathcal{O}$ has the weight p_i, each arc has the weight zero. A schedule π is feasible if the graph $G(\pi)$ does not include a cycle. For a given π, terms of

operations beginning can be determined in time $O(o)$ from the recurrent formula

$$S_j = \max\{S_i + p_i, S_k + p_k\}, j \in \mathcal{O}. \tag{3.55}$$

where an operation i is a direct technological predecessor of the operation $j \in \mathcal{O}$ and an operation k is a directed machine predecessor of the operation $j \in \mathcal{O}$ for a fixed π. We assume $S_j = 0$ for these operations j which have not any technological or machine predecessors.

An example of the graph $G(\pi)$ is given in Figure 3.5 for the same data, as a disjunctive graph from Figure 3.3 – it is visible that the $G(\pi)$ has a more transparent structure and is void of redundance connected with superfluous arcs of the disjunctive representation. For a given feasible schedule π the process of determining the cost function value requires the time $O(o)$, which is thus shorter than for the disjunctive representation.

3.6. Flexible job shop problems

Flexible job shop problems constitute a generalization (hybridization) of the classic job shop problem. In this section, we discuss a flexible job shop problem in which operations have to be executed on one machine from a set of dedicated machines. Then, as a job shop problem it also belongs to the strongly NP-hard class. Although exact algorithms based on a disjunctive graph representation of the solution have been developed (see Pinedo [208]), they are not effective for instances with more than 20 jobs and 10 machines.

Many approximate algorithms, chiefly metaheuristic, have been proposed. Nowicki and Smutnicki [197] proposed a tabu search approach using block properties for the special case of the problem considered (flow shop problem with parallel machines). Hurink [139] developed the tabu search method for this problem. Also Dauzère-Pérès and Pauli [93] used the tabu search approach extending the disjunctive graph representation for the classic job shop problem taking into consideration the assignment of operations to machines. Mastrolilli and Gambardella [180] proposed a tabu search procedure with effective neighborhood functions for the flexible job shop problem.

Many authors have proposed a method of assigning operations to machines and then determining sequence of operations on each machine. This approach was followed by Brandimarte [67] and Pauli [204]. These authors solved the assignment problem (i.e., using dispatching rules) and next applied metaheuristics to solve the job shop problem. Genetic approaches have been adopted to solve the flexible job shop problem, too. Recent works are those of Jia et al. [149], Ho and Tay [135], Kacem et al. [152], Pezzella et al. [207] and Bożejko et al. [31]. Gao et al.

[111] proposed the hybrid genetic and variable neighborhood descent algorithm for this problem.

3.6.1. Problem formulation

The flexible job shop problem (FJSP), also called the general job shop problem with parallel machines, can be formulated as follows. Let $\mathcal{J} = \{1, 2, \ldots, n\}$ be a set of jobs which have to be executed on machines from the set $\mathcal{M} = \{1, 2, \ldots, m\}$. There exists a partition of the set of machines into types, i.e., subsets of machines with the same functional properties. A job constitutes a sequence of some operations. Each operation has to be executed on a dedicated type of machine (from the nest) within a fixed time. The problem consists in the allocation of jobs to machines of dedicated type and in determining the schedule of jobs execution on each machine to minimize the total jobs finishing time. The following constraints have to be fulfilled:

- (*i*) each job has to be executed on only one machine of a determined type at a time,
- (*ii*) machines cannot execute more than one job at a time,
- (*iii*) there are no idle times (i.e., the job execution must not be broken),
- (*iv*) the technological order has to be obeyed.

Let $\mathcal{O} = \{1, 2, \ldots, o\}$ be the set of all operations. This set can be partitioned into sequences corresponding to jobs where the job $j \in \mathcal{J}$ is a sequence of o_j operations which have to be executed in an order on dedicated machines (i.e., in the so-called technological order). Operations are indexed by numbers $(l_{j-1} + 1, \ldots, l_{j-1} + o_j)$ where $l_j = \sum_{i=1}^{j} o_i$ is the number of operations of the first j jobs, $j = 1, 2, \ldots, n$, where $l_0 = 0$ and $o = \sum_{i=1}^{n} o_i$.

The set of machines $\mathcal{M} = \{1, 2, \ldots, m\}$ can be partitioned into q subsets of the same type (*nests*) where the i-th ($i = 1, 2, \ldots, q$) type $\mathcal{M}^i$ includes m_i machines which are indexed by numbers $(t_{i-1} + 1, \ldots, t_{i-1} + m_i)$, where $t_i = \sum_{j=1}^{i} m_j$ is the number of machines in the first i types, $i = 1, 2, \ldots, q$, where $t_0 = 0$ and $m = \sum_{j=1}^{q} m_j$.

An operation $v \in \mathcal{O}$ has to be executed on machines of the type $\mu(v)$, i.e., on one of the machines from the set (nest) $\mathcal{M}^{\mu(v)}$ in time p_{vj} where $j \in \mathcal{M}^{\mu(v)}$.

Let

$$\mathcal{O}^k = \{v \in \mathcal{O} : \ \mu(v) = k\} \tag{3.56}$$

be a set of operations executed in the k-th nest ($k = 1, 2, \ldots, q$). A sequence of operations sets

$$\mathcal{Q} = (\mathcal{Q}^1, \mathcal{Q}^2, \ldots, \mathcal{Q}^m), \tag{3.57}$$

such that for each $k = 1, 2, \ldots, q$

$$\mathcal{O}^k = \bigcup_{i=t_{k-1}+1}^{t_{k-1}+m_k} \mathcal{Q}^i \text{ and } \mathcal{Q}^i \cap \mathcal{Q}^j = \emptyset,\ i \neq j,\ i, j = 1, 2, \ldots, m, \tag{3.58}$$

we call an *assignment of operations from the set $\mathcal{O}$ to machines from the set $\mathcal{M}$* (or shortly, machine workload).

A sequence $(\mathcal{Q}^{t_{k-1}+1}, \mathcal{Q}^{t_{k-1}+2}, \ldots, \mathcal{Q}^{t_{k-1}+m_k})$ is an *assignment* of operations to machines in the i-th nest (shortly, an assignment in the i-th nest). In a special case a machine can execute no operations and then a set of operations assigned to be executed by this machine is an empty set.

Example 3.1. Let $\mathcal{J} = \{J_1, J_2, J_3\}$ $(n = 3)$ be a set of jobs which have to be executed on machines from the set $\mathcal{M} = \{M_1, M_2, M_3, M_4, M_5, M_6\}$ $(m = 6)$. The set of machines can be partitioned into three types (nests):

a) $\mathcal{M}^1 = \{M_1, M_2\}$,
b) $\mathcal{M}^2 = \{M_3, M_4\}$,
c) $\mathcal{M}^3 = \{M_5, M_6\}$.

Due to the notion introduced above, the number of machines $m = 6$, the number of machine types (nests) $q = 3$ and $t_0 = 0$, $t_1 = 2$, $t_2 = 4$, $t_3 = 6$.

Each job has to be executed on one machine from the nest $\mathcal{M}^1$, next from the nest $\mathcal{M}^2$ and at the end from the nest $\mathcal{M}^3$ (satisfying technological requirements). Each job consists of three operations. If $\mathcal{O} = \{O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8, O_9\}$ is the set of all operations, then the job $J_1 = \{O_1, O_2, O_3\}$, $J_2 = \{O_4, O_5, O_6\}$ and $J_3 = \{O_7, O_8, O_9\}$. Therefore, $\mu(O_1) = \mu(O_4) = \mu(O_7) = 1$ (these operations have to be executed on one of the 1-st type machines, i.e., $\mathcal{M}^1$), $\mu(O_2) = \mu(O_5) = \mu(O_8) = 2$ (type $\mathcal{M}^2$) and $\mu(O_3) = \mu(O_6) = \mu(O_9) = 3$ (type $\mathcal{M}^3$). In this example the number of operations $o = 9$ and $l_0 = 0$, $l_1 = 3$, $l_2 = 6$, $l_3 = 9$. Job execution times on each machine are presented in Table 3.1. ■

If the assignment of operations to machines has been completed, then the optimal schedule of operations execution determination (including a sequence of operations execution on machines) leads to the classic scheduling problem solving, that is, the job shop problem (see Section 3.5 and Grabowski and Wodecki [119]).

Let $K = (K_1, K_2, \ldots, K_m)$ be a sequence of sets where $K_i \in 2^{\mathcal{O}^i}$, $i = 1, 2, \ldots, m$ (in a particular case elements of this sequence can be empty sets). By $\mathcal{K}$ we denote the set of all such sequences. The number of elements of the set $\mathcal{K}$ is $2^{|\mathcal{O}^1|} \cdot 2^{|\mathcal{O}^2|} \cdot \ldots \cdot 2^{|\mathcal{O}^m|}$.

Table 3.1. Job execution times on machines.

	Nest $\mathcal{M}^1$		Nest $\mathcal{M}^2$		Nest $\mathcal{M}^3$	
	M_1	M_2	M_3	M_4	M_5	M_6
job J_1	1	2	2	3	3	2
job J_2	1	3	3	1	5	2
job J_3	2	3	3	3	4	2

If $\mathcal{Q}$ is an assignment of operations to machines then $\mathcal{Q} \in \mathcal{K}$ (of course, the set $\mathcal{K}$ includes also sequences which are not feasible; that is, such sequences do not constitute assignments of operations to machines).

For any sequence of sets $K = (K_1, K_2, \ldots, K_m)$ $(K \in \mathcal{K})$ by $\Pi_i(K)$ we denote the set of all permutations of elements from K_i. Thereafter, let

$$\pi(K) = (\pi_1(K), \pi_2(K), \ldots, \pi_m(K)) \tag{3.59}$$

be a concatenation of m sequences (permutations), where $\pi_i(K) \in \Pi_i(K)$. Therefore

$$\pi(K) \in \Pi(K) = \Pi_1(K) \times \Pi_2(K) \times, \ldots, \Pi_m(K). \tag{3.60}$$

It is easy to observe that if $K = (K_1, K_2, \ldots, K_m)$ is an assignment of operations to machines then the set $\pi_i(K)$ $(i = 1, 2, \ldots, m)$ includes all permutations (possible sequences of execution) of operations from the set K_i on the machine i. Further, let

$$\Phi = \{(K, \pi(K)) : \ K \in \mathcal{K} \ \wedge \ \pi(K) \in \Pi(K)\} \tag{3.61}$$

be a set of pairs, where the first element is a sequence set and the second – a concatenation of permutations of elements of these sets. Any feasible solution of the FJSP is a pair $(\mathcal{Q}, \pi(\mathcal{Q})) \in \Phi$ where $\mathcal{Q}$ is an assignment of operations to machines and $\pi(Q)$ is a concatenation of permutations determining the operations execution sequence which are assigned to each machine fulfilling constraints (i)–(iv). By Φ° we denote a set of feasible solutions for the FJSP. Of course, there is $\Phi^\circ \subset \Phi$.

Example 3.2. For the data from Example 3.1, operations:

- O_1, O_4, O_7 should be executed on machines of the nest $\mathcal{M}^1$ (i.e., machines from the set $\{M_1, M_2\}$),
- O_2, O_5, O_8 should be executed on machines of the nest $\mathcal{M}^2$ (i.e., machines from the set $\{M_3, M_4\}$),

- O_3, O_6, O_9 should be executed on machines of the nest $\mathcal{M}^3$ (i.e., machines from the set $\{M_5, M_6\}$.

Set of operations:

- $\mathcal{Q}^1 = \{O_1, O_4, O_7\}$ and $\mathcal{Q}^2 = \emptyset$ constitute a machine-to-operation assignment in the nest $\mathcal{M}^1$ (all the operations are executed on machine M_1),
- $\mathcal{Q}^3 = \{O_2, O_5, O_8\}$ and $\mathcal{Q}^4 = \emptyset$ constitute a machine-to-operation assignment in the nest $\mathcal{M}^2$ (all the operations are executed on machine M_3),
- $\mathcal{Q}^5 = \{O_3, O_6, O_9\}$ and $\mathcal{Q}^6 = \emptyset$ constitute a machine-to-operation assignment in the nest $\mathcal{M}^3$ (all the operations are executed on machine M_5).

Let us take a sequence of sets

$$\mathcal{Q} = (\mathcal{Q}^1, \mathcal{Q}^2, \mathcal{Q}^3, \mathcal{Q}^4, \mathcal{Q}^5, \mathcal{Q}^6), \tag{3.62}$$

where $Q^i,\ 1 = 1, 2, \ldots, 6$ is a set of operations assigned to be executed on machine $M_i \in \mathcal{M}$. It is easy to present that $\mathcal{Q}$ constitutes an assignment of operations of the set $\mathcal{O}$ to machines from the set $\mathcal{M}$.

Now, we shall determine a sequence of operations execution on each machine. Let $\pi_1 = (O_1, O_4, O_7)$ be a permutation (execution schedule) of operations from the set $\mathcal{Q}^1$ on machine M_1. Similarly let $\pi_3 = (O_2, O_5, O_8)$ be a permutation of operations from the set $\mathcal{Q}^3$ on the machine M_3 and let $\pi_5 = (O_3, O_5, O_8)$ be a permutation of operations from the set $\mathcal{Q}^5$ on machine M_5. Moreover, we presuppose $\pi_2 = \pi_4 = \pi_6 = \emptyset$.

Permutation concatenation

$$\pi(\mathcal{Q}) = (\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6), \tag{3.63}$$

determines an operations execution sequence on each machine. It is easy to check out that the pair $\Theta = (\mathcal{Q}, \pi(\mathcal{Q}))$ constitutes a feasible solution of the FJSP instance described in Example 3.1. ■

3.6.2. Graph models

Any feasible solution $\Theta = (\mathcal{Q}, \pi(\mathcal{Q})) \in \Phi^\circ$ (where $\mathcal{Q}$ is an assignment of operations to machines and $\pi(\mathcal{Q})$ determines the operations execution sequence on each machine) of the FJSP can be presented as a directed graph with weighted vertices $G(\Theta) = (\mathcal{V}, \mathcal{R} \cup \mathcal{E}(\Theta))$ where $\mathcal{V}$ is a set of vertices and a $\mathcal{R} \cup \mathcal{E}(\Theta)$ is a set of arcs, with:

1) $\mathcal{V} = \mathcal{O} \cup \{s, c\}$, where s and c are additional (fictitious) operations which represent 'start' and 'finish', respectively. A vertex $v \in \mathcal{V} \setminus \{s, c\}$ possesses two attributes:

- $\lambda(v)$ – a number of machines on which an operation $v \in \mathcal{O}$ has to be executed,
- $p_{v,\lambda(v)}$ – a weight of vertex which equals the time of operation $v \in \mathcal{O}$ execution on the assigned machine $\lambda(v)$.

Weights of additional vertices $p_s = p_c = 0$.

2)

$$\mathcal{R} = \bigcup_{j=1}^{n} \left[\bigcup_{i=1}^{o_j-1} \{(l_{j-1}+i, l_{j-1}+i+1)\} \cup \{(s, l_{j-1}+1)\} \cup \{(l_{j-1}+o_j, c)\} \right]. \tag{3.64}$$

A set $\mathcal{R}$ includes arcs which connect successive operations of the job, arcs from vertex s to the first operation of each job and arcs from the last operation of each job to vertex c.

3)

$$\mathcal{E}(\Theta) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{|\mathcal{O}^k|-1} \{(\pi_k(i), \pi_k(i+1))\}. \tag{3.65}$$

It is easy to notice that arcs from the set $\mathcal{E}(\Theta)$ connect operations executed on the same machine (π_k is a permutation of operations executed on the machine M_k, that is, operations from the set $\mathcal{O}^k$).

Arcs from the set $\mathcal{R}$ determine the operations execution sequence inside jobs (a technological order) and arcs from the set $\mathcal{E}(\pi)$ the operations execution sequence on each machine.

Remark 3.1. *A pair $\Theta = (\mathcal{Q}, \pi(\mathcal{Q})) \in \Phi$ is a feasible solution for the FJSP if and only if the graph $G(\Theta)$ does not include cycles.*

Example 3.3. A directed graph $G(\Theta)$ for the FJSP instance from Example 3.1 is presented in Figure 3.6 for the feasible solution $\Theta = (\mathcal{Q}, \pi(\mathcal{Q}))$ from Example 3.2. Arcs from the set $\mathcal{R}$ are represented by a solid line; a dashed line represents arcs from the set $\mathcal{E}(\Theta)$. The number inside a circle is the vertex number; the number near a circle presents the vertex weight. ■

A sequence of vertices $(v_1, v_2, \ldots, v_k)$ in $G(\Theta)$ such that an arc $(v_i, v_{i+1}) \in \mathcal{R} \cup \mathcal{E}(\Theta)$ for $i = 1, 2, \ldots, k-1$, we call a *path* from vertex v_1 to v_k. By $C(v, u)$ we denote the longest path (called a *critical path*) in the graph $G(\Theta)$ from the vertex v to u ($v, u \in \mathcal{V}$) and by $L(v, u)$ we denote a *length* (sum of vertex weights) of this path.

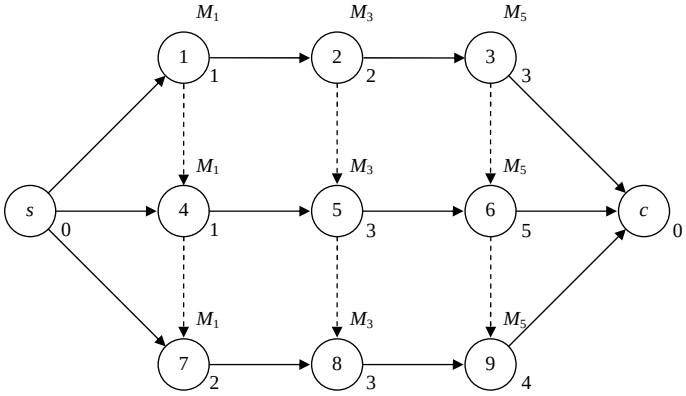


Fig. 3.6. A directed graph for a solution $\Theta = (\mathcal{Q}, \pi(\mathcal{Q}))$ from Example 3.2.

It is easy to notice that the time of all operations execution $C_{\max}(\Theta)$ related with the assignment of operations $\mathcal{Q}$ and schedule $\pi(\mathcal{Q})$ equals the length $L(s,c)$ of the critical path $C(s,c)$ in the graph $G(\Theta)$. A solution of the FJSP amounts to determining a feasible solution $\Theta = (\mathcal{Q}, \pi(\mathcal{Q})) \in \Phi^{\circ}$ for which the graph connected with this solution $G(\Theta)$ has the shortest critical path, that is, it minimizes $L(s,c)$.

Let $C(s,c) = (s, v_1, v_2, \ldots, v_w, c)$, $v_i \in \mathcal{O}$ $(1 \leq i \leq w)$ be a critical path in the graph $G(\Theta)$ from the starting vertex s to the final vertex c. This path can be divided into subsequences of vertices

$$\mathcal{B} = (B^1, B^2, \ldots, B^r), \tag{3.66}$$

called *blocks* in the permutations on the critical path $C(s,c)$ (Grabowski [122], Grabowski and Wodecki [119]) where:

(a) a block is a subsequence of verticesfrom the critical path including successive operations executed directly one after another,

(b) a block includes operations executed on the same machine,

(c) a product of any two blocks is an empty set,

(d) a block is a maximal (according to the inclusion) subset of operations from the critical path fulfilling constraints (a)–(c).

Next, only these blocks are considered for which $|B^k| > 1$, i.e., non-empty blocks. If B^k $(k = 1, 2, \ldots, r)$ is a block on the machine M_i $(i = 1, 2, \ldots, m)$ from the nest t $(t = 1, 2, \ldots, q)$ then we shall denote it as follows

$$B^k = (\pi_i(a^k), \pi_i(a^k + 1), \ldots, \pi_i(b^k - 1), \pi_i(b^k)), \tag{3.67}$$

where $1 \leq a^k < b^k \leq |Q^i|$. Operations $\pi(a^k)$ and $\pi(b^k)$ in the block B^k are called *the first* and *the last*, respectively. In turn a block without the first and

the last operation we call an *internal block*. The definitions given are presented in Figure 3.7.

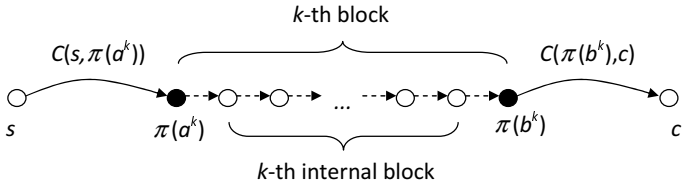


Fig. 3.7. Blocks on the critical path.

Example 3.4. There exist two critical paths (both with the length 15) in the graph $G(\Theta)$ shown in Figure 3.6:

1. $(s, O_1, O_2, O_5, O_6, O_9, c)$ including two blocks: $B^1 = (O_2, O_5)$ and $B^2 = (O_6, O_9)$. As these blocks consist of two elements, there are no internal blocks. The operation $a^1 = O_2$ is the first one, $b^1 = O_5$ is the last operation of the block $B^1 = (O_2, O_5)$.
2. $(s, O_1, O_2, O_3, O_6, O_9, c)$ including one block $B^1 = (O_3, O_6, O_9)$ consisting of three elements. The operation $a^1 = O_3$ is the first one, $b^1 = O_9$ is the last operation of this block. The internal block consists of the single operation O_6. ■

In the work of Grabowski [122] there are theorems called *elimination criteria* of blocks in the job shop problem.

Theorem 3.4 ([122]). *Let $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ be a sequence of blocks of the critical path in the acyclic graph $G(\Theta)$, $\Theta \in \Phi^\circ$. If the graph $G(\Omega)$ is feasible (i.e., it represents a feasible solution) and if it is generated from $G(\Theta)$ by changing the order of operations execution on some machine and $C_{\max}(\Omega) < C_{\max}(\Theta)$ then in the $G(\Omega)$:*

(i) at least one operation from a block B^k, $k \in \{1, 2, \ldots, r\}$ precedes the first element $\pi(a^k)$ of this block, or

(ii) at least one operation from a block B^k, $k \in \{1, 2, \ldots, r\}$ occurs after the last element $\pi(b^k)$ of this block.

Changing the order of operations in any block does not generate a solution with lower value of the cost function. At least one operation from any block should be moved before the first or after the last operation of this block to generate a solution (graph) with smaller weight of the critical path. We use this property to reduce the neighborhood size, i.e., do not generate solutions with greater values (compared to the current solution) of the cost function.

Part II

SINGLE-WALK PARALLELIZATION

Chapter 4

Single machine scheduling

The main aim of this chapter is to show the effectiveness of the transfer of an original technology of a huge neighborhood searching into the parallel computing environment. We propose a single-walk parallel algorithm to solve the single machine total weighted tardiness problem using the technique of an exponential-size neighborhood searching in polynomial time, based on dynamic programming. The approach proposed is strong enough to solve also $1|r_i|\sum w_iT_i$, $1||\sum w_iU_i$, $1||\sum(w_iE_i + u_iT_i)$ and $1||\sum w_iC_i$ problems (all of these methods are NP-hard). Nowadays this method is one of the most powerful approximate methods for the single machine total weighted scheduling problem. The neighborhood is generated by executing series of swap moves.

4.1. Introduction

The goal of this section is to provide the fundamental well-known facts of the theoretical parallel computing on the PRAM machine, as well as the literature review for the single-walk parallelization of the discrete optimization problems solving algorithms. We consider an algorithm which employs a single process (thread) to guide the search. The thread performs in a cyclic way (iteratively) two leading tasks:

(A) goal function evaluation for a single solution or a set of solutions,
(B) management, e.g. solution filtering and selection, collection of history, updating.

Part (B) takes statistically 1–3% total iteration time, thus its acceleration is useless. Part (A) can be accelerated in a multithread environment in various manners – our aim is to find either *cost optimal* method or non-optimal one in terms of cost while offering *the shortest running time*. It is noteworthy to observe

that if Part (B) takes β percentage of the 1-processor algorithm and if it is not parallelizable, the speedup of the parallel algorithm for *any* number of processors p cannot be greater than $\frac{1}{\beta}$ (according to Amdahl's law). In practice, if Part (B) takes 2% of the total execution time, the speedup can achieve at most the value of 50.

4.2. PRAM computation model

We make a complexity analysis of the cost function determination algorithms for their implementations on Parallel Random Access Machine (PRAM model). A PRAM [91, 238] consists of many cooperating processors, each being a random access machine (RAM), commonly used in theoretical computer science. Each processor can make local calculations, e.g. additions, subtractions, shifts, conditional and unconditional jumps and indirect addressing. All the processors in the PRAM model are synchronized and have an access to a shared global memory in constant time $O(1)$. There is also no limit on the number of processors in the machine, and any memory cell is uniformly accessible from any processor. The amount of shared memory in the system is not limitable.

We make use of three kinds of PRAMs here: CRCW (*Concurrent Read Exclusive Write*) in which simultaneous reading and writing are allowed, CREW (*Concurrent Read Exclusive Write*) where processors can read from the same memory cell concurrently, and EREW (*Exclusive Read Exclusive Write*) where the concurrency of reading is forbidden. Both CREW and EREW models resemble GPU programming model. The first, CRCW model is very useful to design special, very fast parallel algorithms, although this is difficult to achieve in practice. There are three kinds of CRCW PRAMs with different ways of handling concurrent writing:

- '*common*' model in which all processors writing to the same location concurrently are required to write the same data,
- '*arbitrary*' model in which an arbitrary processor succeeds,
- '*priority*' model in which the lowest numbered processor succeeds with writing.

We take advantage of the following well-known facts for the PRAM parallel computer model (Cormen et al. [83]):

Fact 4.1. *A sequence of prefix sums* $(y_1, y_2, \ldots, y_n)$ *of input sequence* $(x_1, x_2, \ldots, x_n)$ *such that* $y_k = y_{k-1} + x_k = x_1 + x_2 + \ldots + x_k$ *for* $k = 2, 3, \ldots, n$ *where* $y_1 = x_1$ *can be calculated in time* $O(\log n)$ *on the EREW PRAM with* $O(n/\log n)$ *processors.*

In line with the statement above we can assume that the sum of n values can be calculated in time $O(\log n)$ on $O(n/\log n)$-processor EREW PRAMs. We can also calculate minimal or maximal values of a sequence based on the following fact.

Fact 4.2. *The minimal and the maximal value of input sequence* $(x_1, x_2, \ldots, x_n)$ *can be determined in time* $O(\log n)$ *on the EREW PRAM with* $O(n/\log n)$ *processors.*

The next fact makes it possible to calculate a function in constant time $O(1)$ on the PRAM.

Fact 4.3. *The value of* $y = (y_1, y_2, \ldots, y_n)$ *where* $y_i = f(x_i)$, $x = (x_1, x_2, \ldots, x_n)$ *can be calculated on the CREW PRAM with* n *processors in time* $O(c) = O(1)$, *where* c *is a time needed to calculate the single value of* $y_i = f(x_i)$.

We need n processors to do this. One can also formulate the following fact for PRAM with a fewer number of processors.

Fact 4.4. *The problem formulated in the previous fact can be calculated in time* $O(\log n)$ *on* $O(n \log n)$ *processors.*

If we do not possess such a big number of processors we can use the following fact to keep the same cost:

Fact 4.5. *If the algorithm* A *works on* p*-processor PRAM in time* t, *then for each* $p' < p$ *there exists an algorithm* A' *for the same problem which works on* p'*-processor PRAM in time* $O(pt/p')$.

The facts mentioned above give a theoretical tool for the single-walk parallel algorithm analysis. The PRAM model gives a good approximation of the fine-grained concurrent computing systems behavior, such as GPUs.

4.3. Calculations for single-walk parallelization

The goal of this method is to speed up the process of a neighborhood graph passing through parallelization of the most time-consuming operations – calculations of the cost function of parallelization of the process of neighbors generating. In the case of parallelization of cost function calculations, the speeding up of computations can be obtained by keeping identical – as in sequence algorithm – trajectory of passing by a neighborhood graph. In the other case, i.e., decomposition of the neighborhood generating process into parallel processes, there occurs a situation in which a parallel algorithm, checking concurrently a greater number of neighbors than a sequential algorithm does (usually using a mechanism of reducing the

size of the neighborhood) will be moving along a better trajectory, determining a more advantageous path of passing by the neighborhood graph and obtaining better results of computations (solutions with better cost function values).

Parallelization of single-walk algorithms has to be done with fine-grained granularity because of the frequent communication and synchronization. The first application based on this model appears in the context of parallelization of simulated annealing and genetic algorithm parallel metaheuristics. Although parallel decomposition of the neighborhood does not always lead to the computations time reduction, it is frequently applied to increase the neighborhood size considered. This type of a parallel tabu search algorithm was proposed by Fiechter [106] for the traveling salesman problem. A synchronic tabu search was also researched by Porto and Ribeiro [209, 210, 211]. In paper [212], Porto, Kitajima and Ribeiro present parallel tabu search algorithms based on a *master-slave* model with dynamic load balancing of processors. Bożejko [25] proposed a parallel scatter search metaheuristic with single-walk parallelization of the goal function calculation. Bożejko, Smutnicki and Uchroński [35] propose a single-walk parallel goal function calculation in metaheuristics with the use of 128-processor nVidia Tesla GPU.

Aarts and Verhoeven [1, 260] distinguished two subclasses in the class of single trajectory parallel search algorithms. A *single-step* class includes algorithms in which a neighborhood is searched by concurrently running parallel processes, but only a single neighbor is chosen as a result. In a *multiple-step* class a sequence of following moves in a neighborhood graph is determined and concurrently searched.

4.4. Huge neighborhoods

In the case of traditional algorithms the neighborhood is generated by single transformations (moves). Let k and l ($k \neq l$) be a pair of positions in a permutation: $\pi = (\pi(1), \pi(2), \ \ldots\ , \pi(k-1), \pi(k), \pi(k+1), \ \ldots\ , \pi(l-1), \pi(l), \pi(l+1), \ \ldots\ , \pi(n))$. Of the many types of moves considered in the literature, the following two are the most common:

1. Insert move i_l^k consists in removing the job $\pi(k)$ from the position k and inserting it in position l. Thus the move i_l^k generates a new permutation $i_l^k(\pi) = \pi_l^k$ in the following way: if $k < l$, then
   $\pi_l^k = (\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(l-1), \pi(l), \pi(k), \pi(l+1), \ldots, \pi(n))$
   else
   $\pi_l^k = (\pi(1), \ldots, \pi(l-1), \pi(k), \pi(l), \pi(l+1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(n))$.
   Each of n elements can be inserted to any of n places, so insert type move generates the neighborhood of $n(n-1)$ elements.

2. Swap move s_l^k, in which the jobs $\pi(k)$ and $\pi(l)$ are swapped among some positions k and l. The move s_l^k generates permutation $s_l^k(\pi) = \pi_l^k = (\pi(1), \pi(2), \ldots, \pi(k-1), \pi(l), \pi(k+1), \ldots, \pi(l-1), \pi(k), \pi(l+1), \ldots, \pi(n))$. Each of n elements can be swapped with any of the other $n-1$ elements, so this move generates the neighborhood of $n(n-1)/2$ (without repetitions).

The insert move can be executed in time $O(n)$, the swap move, in constant time $O(1)$ in the classic, linear representation of permutation. Local search methods typically determine a solution x^{i+1} from the neighborhood $N(x^i)$ with the minimal goal function value, i.e.,

$$F(x^{i+1}) = \min_{x \in N(x^i)} F(x). \tag{4.1}$$

The aim of increasing the size of $N(x^i)$ is to search 'faster' big areas of the solution space. The question is how big $N(x^i)$ can be to search it efficiently enough. Several kinds of this type huge neighborhoods are discussed in the literature (see Bożejko and Wodecki [46]), especially in the TSP context. The most promising for scheduling problem seems to be that proposed by Congram et al. [81], known as dynasearch neighborhood. Applying this method it is possible to revise this non-polynomial size neighborhood (calculate the minimal element) in polynomial time. Methods of dynamical programming require, however, a lot of time and memory, so they can be applied only in limited range of large problems.

Here we present parallel algorithm in which the fundamental element is a parallel method of generating and revising the huge neighborhood. We prove that it is possible to show some new properties which indicate that such a method is cost-optimal with efficiency $O(1)$. We apply an appropriate algorithm to solve the single machine total weighted tardiness problem.

Let us consider two swap moves s_j^i and s_l^k. These moves are said to be *independent* if

$$\max\{i, j\} < \min\{l,\ k\} \ \text{ or } \ \min\{i, j\} > \max\{l,\ k\}. \tag{4.2}$$

The *huge swap neighborhood* consists of all permutations that can be obtained from π by a series of pairwise independent swap moves. The size of the neighborhood is $2^{n-1} - 1$ (see Congram et al. [81]).

We define a partial sequence in the state (j, π), for $j = 1, 2, \ldots, n$, if it can be obtained from the partial sequence $\pi(1), \pi(2), \ldots, \pi(j)$ by applying a series of independent swaps. Let π_j be a partial sequence with minimum total weighted tardiness for jobs $\pi(1), \pi(2), \ldots, \pi(j)$ among partial sequences in state (j, π). Further, let $F(\pi_j)$ be the total weighted tardiness for jobs $\pi(1), \pi(2), \ldots, \pi(j)$ in π_j, i.e.,

$$F(\pi_j) = \min\{F(\beta) :\ \beta \in (j, \pi)\}. \tag{4.3}$$

Optimality of $F(\pi_j)$ understood as the 'optimal substructure' property leads to the $F(\pi_n)$ optimality, that is, the best move of the whole huge neighborhood will be determined.

The partial sequence π_j must be obtained from a partial sequence π_i that has the minimum objective value from all partial sequences in the first previous state (i, π), where $0 \leq i < j$, by appending the job $\pi(j)$ if $i = j - 1$, or by the first appending job $\pi(j)$ and then interchanging jobs $\pi(i+1)$ and $\pi(j)$ if $0 \leq i < j-1$. These two possibilities are considered in detail below.

Case A. $i = j - 1$. In this case, the job $\pi(j)$ is not involved in any $\pi(j)$ swaps, and $\pi(j)$ simply appends to a partial sequence π_{j-1}; hence $\pi_j = (\pi_{j-1}, \pi_j)$. Accordingly,

$$F(\pi_j) = F(\pi_{j-1}) + w_{\pi(j)}(C_{\pi(j)} - d_{\pi(j)})^+, \tag{4.4}$$

where, for any real x, $(x)^+ = \max\{0,\ x\}$.

Case B. $0 \leq i < j - 1$. Here, jobs $\pi(j)$ and $\pi(i+1)$ are swapped; that is why π_j can be written as $\pi_j = (\pi_i, \pi(j), \pi(j+2), .., \pi(j-1), \pi(i+1))$, and the total weighted tardiness $F(\pi_j)$ is readily computed as

$$\begin{aligned} F(\pi_j) = F(\pi_i) &+ w_{\pi(j)}(C_{\pi(i)} + p_{\pi(j)} - d_{\pi(j)})^+ + \\ &+ \sum_{k=i+2}^{j-1} w_{\pi(k)}(C_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)} - d_{\pi(j)})^+ + \\ &+ w_{\pi(i+1)}(C_{\pi(j)} - d_{\pi(i+1)})^+. \end{aligned} \tag{4.5}$$

These values can be determined recursively. For any $j = 2, 3, \ldots, n$,

$$F(\pi_j) = \min \begin{cases} F(\pi_{j-1}) + K_{\pi(j),\pi(j)}, \\ \min\limits_{0 \leq i \leq j-2} \{F(\pi_i) + K_{\pi(j),\pi(i)} + K_{\pi(i+1),\pi(j)} + \\ + \sum\limits_{k=i+2}^{j-1} w_{\pi(k)}(P_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)} - d_{\pi(k)})^+, \end{cases} \tag{4.6}$$

where the initialization is $F(\pi_0) = 0$, $F(\pi_1) = w_{\pi(1)}(p_{\pi(1)} - d_{\pi(1)})^+$ and a temporary variable $K_{\pi(a),\pi(b)} = w_{\pi(a)}(C_{\pi(b)} - d_{\pi(a)})^+$. The optimal solution is $F(\pi_n)$ and this algorithm runs in time $T_{seq} = O(n^3)$ – there are two loops to be executed inside equation (4.6) (a minimum of at most $n-1$ elements and a sum of at most $n-2$ elements) and this formula has to be calculated $n-1$ times, for each j.

4.5. Huge neighborhood searching method

In this section, we present application of the exponential-size neighborhood searching method for some NP-hard single machine scheduling problems. Beginning from a sequential approach (in the range of a neighborhood searching), we show how to design an efficient parallel huge neighborhood searching approach. Such a methodology has not yet been proposed in the literature.

Problem $1|r_i|\sum w_iT_i$

In this problem, a starting time C_i of a job $i \in \mathcal{J}$ is not less than the release date r_i. For any permutation (sequence) of jobs $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, a time of finishing the job $\pi(i) \in \mathcal{J}$

$$C_{\pi(i))} = C_{\pi(i-1)} + (r_{\pi(i)} - C_{\pi(i-1)})^+ + p_{\pi(i)}, \tag{4.7}$$

where $C_{\pi(0)} = 0$. Recurrent formula (4.6) takes the form

$$F(\pi_j) = \min \begin{cases} F(\pi_{j-1}) + w_{\pi(j)}(C_{\pi(j)} - d_{\pi(j)})^+, \\ \min\limits_{1 \le i \le j-1} G_{\text{calc}}(\pi_i), \end{cases} \tag{4.8}$$

where the initialization is $F(\pi_0) = 0$ and $F(\pi_1) = w_{\pi(1)}(C_{\pi(1)} - d_{\pi(1)})^+$. The value of the function $G_{\text{calc}}(\pi_i)$ can be computed from the procedure presented in Figure 4.1.

```
Algorithm 2.  G_calc(π,j)
 ret = F(j-1);
 tmp = max{P(j-1), r_π(i)};
 ret+= max{(tmp + p_π(i) - d_π(i)),0}*w_π(i);
 tmp+=p_π(i);
 for(v = j+1; v <= i-1; v++)
   tmp=max{tmp,r_π(v)};
   ret+=max{(tmp + p_π(v) - d_π(v)),0}*w_π(v);
   tmp+=p_π(v);
 tmp=max{tmp, r_π(j)};
 ret+=max{(tmp + p_π(j) - d_π(j)), 0}*w_π(j);
 return ret;
```

Fig. 4.1. G_{calc} function.

Computational complexity of this function is $O(j^2)$. Therefore, the algorithm of determining the value of $F(\pi_n)$ from formula (4.6) possesses the complexity $O(n^3)$, as mentioned in Section 4.4.

Problem $1||\sum w_iU_i$

For a given sequence π we can compute for the job $\pi(i)$ its completion time $C_{\pi(i)}$ and the unit penalty $U_{\pi(i)}$. $U_{\pi(i)} = 1$ if $C_{\pi(i)} > d_{\pi(i)}$ and $U_{\pi(i)} = 0$ otherwise. In another form of notation $U_{\pi(i)} = \mathrm{sgn}((C_{\pi(i)} - d_{\pi(i)})^+)$.

For $j < i$ we can compute similarly as in the original (i.e., for $1||\sum w_iT_i$) method

$$F(\pi_j) = \min \begin{cases} F(\pi_{j-1}) + U_{\pi(j)}w_{\pi(j)}, \\ \min\limits_{0\le i\le j-2}\{F(\pi_{j-1}) + V_{\pi(j-1),\pi(i)}w_{\pi(i)} + W_{\pi(i),\pi(j)}w_{\pi(j)} + \\ + \sum\limits_{k=i+2}^{j-1} \mathrm{sgn}((C_{\pi(k)} - p_{\pi(j)} + p_{\pi(i)} - d_{\pi(k)})^+)w_{\pi(k)}, \end{cases} \tag{4.9}$$

where temporary variables $V_{\pi(a),\pi(b)} = \mathrm{sgn}((C_{\pi(a)}+p_{\pi(b)}-d_{\pi(b)})^+)$ and $W_{\pi(a),\pi(b)} = \mathrm{sgn}((C_{\pi(a)} - d_{\pi(b)})^+)$ are used. Computational complexity of the search over the whole neighborhood is $O(n^3)$. Based on (4.8) and (4.9) it is easy to formulate recurrence relationship for the $1|r_i|\sum w_iU_i$ problem.

Problem $1||\sum(w_iE_i + u_iT_i)$

In this problem, by e_i and d_i we mean the expected earliest and latest moments of completing a job $i \in \mathcal{J}$. If the scheduling of jobs is established and C_i is the moment of finishing a job i, then we call $E_i = [e_i - C_i]^+$ the *earliness* and $T_i = [C_i - d_i]^+$ the *tardiness*. The expression $u_iE_i + w_iT$ is the *cost* of executing a job, where u_i and w_i ($i \in \mathcal{J}$) are nonnegative coefficients of a goal function. The problem consists in minimizing the sum of the costs of jobs, that is, the function $\sum_{i=1}^{n}(u_iE_i + w_iT_i)$.

To obtain an adequate recurrence relationship it is necessary to exchange the cost of job w_iT_i for $u_iE_i + w_iT_i$ in formula (4.6).

Problem $1||\sum w_iC_i$

For the problem of minimizing the costs of jobs completion the recurrence formula (4.6) takes the form

$$F(\pi_j) = \min \begin{cases} F(\pi_{j-1}) + w_{\pi(j)}C_{\pi(j)}, \\ \min\limits_{0\le i\le j-2}\{w_{\pi(j)}(C_{\pi(i)} + p_{\pi(j)}) + w_{\pi(i+1)}C_{\pi(j)} + \\ + F(\pi_i)\} + \sum\limits_{k=i+2}^{j-1} w_{\pi(k)}(C_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)}). \end{cases} \tag{4.10}$$

The algorithm of determining optimal value of $F(\pi_n)$ has computational complexity $O(n^3)$.

4.6. Parallel huge neighborhood searching method

We shall make use of the facts for the PRAM parallel computer model described in Section 4.2.

Theorem 4.1. *The best element of the huge neighborhood can be determined in time* $O(n \log^2 n)$ *on the PRAM with* $O\left(\frac{n^2}{\log^2 n}\right)$ *processors.*

Proof. Let us notice that all times of job finishing C_j, $j = 1, 2, \ldots, n$ can be computed as prefix sums (see Fact 4.1, Section 4.2) in time $O(\log n)$ using $O(n/\log n)$ processors. Next, to compute the value of $F(\pi_j)$ in equation (4.6) it is necessary to determine the sum of at most n values of

$$\sum_{k=i+2}^{j-1} w_{\pi(k)}(P_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)} - d_{\pi(k)})^+ \tag{4.11}$$

and next the minimal element of (at most) n values

$$\min_{0 \le i \le j-2} \{F(\pi_i) + w_{\pi(j)}(C_{\pi(i)} + p_{\pi(j)} - d_{\pi(j)})^+ + \sum_{k=i+2}^{j-1} w_{\pi(k)}(P_{\pi(k)} +$$

$$+ p_{\pi(j)} - p_{\pi(i+1)} - d_{\pi(k)})^+ + w_{\pi(i+1)}(C_{\pi(j)} - d_{\pi(i+1)})^+\}, \tag{4.12}$$

with computed sums inside. Operations of addition, due to their small and, first of all, constant cost, independent of the n number, can be executed in constant time $O(1)$, so we can omit them in our discussion. To determine the above-mentioned minimal element we need PRAM with $O(\frac{n}{\log n})$ processors (see Fact 4.2, Section 4.2), each one of which has to execute computations of the sums (mentioned at the beginning) also with $O(\frac{n}{\log n})$ processors, so in total we need

$$p = O\left(\frac{n}{\log n}\right) \cdot O\left(\frac{n}{\log n}\right) = O\left(\frac{n^2}{\log^2 n}\right) \tag{4.13}$$

processors of the PRAM and the time

$$T_{par}(p) = O(\log n) \cdot O(\log n) = O(\log^2 n). \tag{4.14}$$

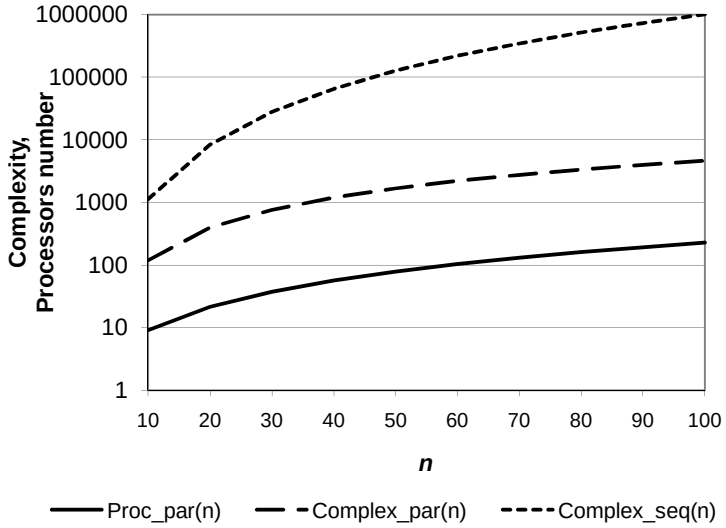


Fig. 4.2. Comparison (on the logarithmic scale) of complexity functions.

At the end, it is necessary to compute the minimum of two calculated values of (4.6), which we can do in constant time $O(1)$ on one processor. Therefore, to determine all values $F(\pi_j)$, $j = 2, 3, \ldots, n$ we need time

$$T_{par}(p) = n \cdot O(\log^2 n) = O(n \log^2 n) \tag{4.15}$$

and

$$p = O\left(\frac{n^2}{\log^2 n}\right) \tag{4.16}$$

processors. ■

The conclusion of this theorem is as follows: such a method has a speedup

$$S(p) = \frac{T_{seq}}{T_{par}(p)} = O\left(\frac{n^3}{n \log^2 n}\right) = O\left(\frac{n^2}{\log^2 n}\right) \tag{4.17}$$

in the order of the number of processors used, hence the algorithm connected with this theorem is cost-optimal with the efficiency $O(1)$. The speedup obtained has asymptotically maximal possible value.

Parallelization provides us with two possible benefits: shortening computation time or examining more solutions in the same time, because Fact 4.5 from Section 4.2 allows searching huge neighborhood, e.g. with 4-processor PRAM machine in time $O(n^3/4)$, so 4 times faster than a sequential algorithm does. It is also possible to use all $O\left(\frac{n^2}{\log^2 n}\right)$ processors while keeping cost optimality and obtaining maximal speedup of computation process. The aforementioned number

of processors is not potentially too big and can be encountered in real parallel systems. A comparison of the speed of functions increasing (complexity of sequential and parallel method, number of processors) is presented in Figure 4.2, where Complex_seq$(n) = n^3$, Complex_par$(n) = \frac{n}{\log^2 n}$, Proc_par$(n) = \frac{n^2}{\log^2 n}$.

4.7. Remarks and conclusions

The methods described in this chapter give an effective methodology of cost-optimal single-walk parallelization of the huge neighborhood searching process. Thanks to it there is a possibility to take advantage of full computational power of parallel mainframe computers equipped with shared memory as well as GPGPU to find good solutions of hard scheduling problems.

The neighborhood generated by series of swap moves, which has an exponential size, is explored in polynomial time $T_{seq} = O(n^3)$. The proposed cost-optimal parallelization makes it possible to speed up the calculations obtaining the parallel runtime $T_{par} = O(n \log^2 n)$ and the speedup $S(p) = O\left(\frac{n^2}{\log^2 n}\right)$.

The method proposed here can be applied on GPGPU environment making use of a big number of processors working as SIMD machine and connected by the fast shared memory. In a hybrid CPU-GPU implementation the huge neighborhood generation function discussed can be coded on GPU as the most computational complex element of the whole single-walk parallel algorithm.

Summing up, in this chapter we proposed the new methodology of transferring the best huge neighborhood search technologies in the local search methods into the parallel computing environment. These exponential-size neighborhoods are searched in an effective way – not only in polynomial time, as sequential method does, but also by a cost-optimal parallel method.

Chapter 5

Job shop scheduling

The goal of this chapter is to propose a methodology of the effective cost function determination for the job shop scheduling problem in parallel computing environment. Parallel Random Access Machine (PRAM) model is applied for the theoretical analysis of algorithm efficiency. The methods need a fine-grained parallelization, therefore the approach proposed is especially devoted to parallel computing systems with fast shared memory (e.g. GPGPU, *General-Purpose computing on Graphics Processing Units*).

5.1. Introduction

There are only a few papers dealing with parallel algorithms for the job shop scheduling problem. Bożejko et al. [34] proposed a single-walk parallelization of the simulated annealing metaheuristic for the job shop problem. Steinhöfel et al. [237] described the method of parallel cost function determination in time $O(\log^2 o)$ on $O(o^3)$ processors, where o is the number of all operations. Bożejko [25] considered a method of parallel cost function calculation for the flow shop problem, which constitutes a special case of the job shop problem. Here we shall propose a more efficient version of the algorithm developed by Steinhöfel et al., which works in time $O(\log^2 o)$ on $O(o^3/\log o)$ processors. Besides, we show a cost-optimal parallelization which takes a time $O(d)$, where d is the number of layers in the topological sorted graph representing a solution. Finally, we prove that this method has a constant $O(1)$ time complexity if we know a value of the upper bound of the cost function value.

5.2. Sequential determination of the cost function

Taking into consideration constraints (3.47)–(3.49) presented in Section 3.5 it is possible to determine the time moments of job completion C_j, $j \in \mathcal{O}$ and job beginning S_j, $j \in \mathcal{O}$ in time $O(o)$ on the sequential machine using the recurrent formula

$$S_j = \max\{S_i + p_i, S_k + p_k\}, j \in \mathcal{O}. \tag{5.1}$$

where an operation i is a direct technological predecessor of the operation $j \in \mathcal{O}$ and an operation k is a directed machine predecessor of the operation $j \in \mathcal{O}$. The determination procedure of $S_j, j \in \mathcal{O}$ from the recurrent formula (5.1) should be initiated by an assignment $S_j = 0$ for those operations j which do not possess any technological or machine predecessors. Next, in each iteration an operation j has to be chosen for which:

1. the execution beginning moment S_j has not been determined yet, and
2. these moments were determined for all its direct technological and machine predecessors; for such an operation j the execution beginning moment can be determined from (5.1).

It is easy to observe that the order of determining S_j times corresponds to the index of the vertex of the graph $G(\pi)$ connected with an operation j after the topological sorting of this graph. The method mentioned above is in fact a simplistic sequential topological sort algorithm without indexing of operations (vertices of the graph). If we add to this algorithm an element of indexing vertices, for which we calculate S_j value, we obtain a sequence which is the topological order of vertices of the graph $G(\pi)$. Now, we define *layers* of the graph collecting vertices (i.e., operations) for which we can calculate S_j in parallel, as we have calculated starting times for all machine and technological predecessors of operations in the layer (see Figure 5.1).

Definition 5.1. *The layer of the graph $G(\pi)$ is a subsequence of the sequence of vertices ordered by the topological sort algorithm, such that there are no arcs between vertices of this subsequence.*

We will need this definition in the next paragraph.

5.3. Parallel determination of the cost function

Two kinds of methods for parallel cost function determination are proposed. The first class of methods is based on matrix multiplication method and it enables

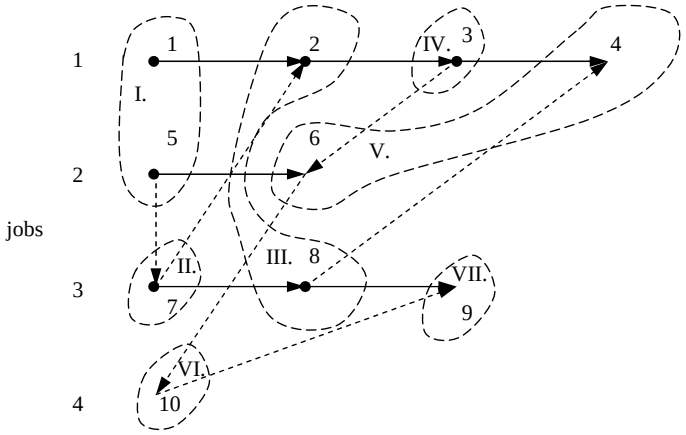


Fig. 5.1. A sample of conjunctive graph for the job shop problem with $d = 7$ layers.

obtaining a good parallel runtime. The second class of methods uses the partitioning of a graph into layers. Algorithms based on this class are cost-optimal. Both approaches shown in this chapter are new, original results.

5.3.1. Methods based on matrix multiplication

We propose an original method using $O\left(\frac{o^3}{\log o}\right)$-processor CREW PRAM with the computational complexity $O(\log^2 o)$. This algorithm is $O(\log o)$ times more efficient than the algorithm proposed in the paper of Steinhöfel et al. [237] and it can be used not only for $J||C_{\max}$ but also for $J||\sum f_i$ problems as well as for $J||f_{\max}$ defined in Section 3.5.

Theorem 5.1. *For a fixed feasible operations order π for the $J||\sum f_i$ or $J||f_{\max}$ problem, the value of the cost function can be determined in time $O(\log^2 o)$ on $O\left(\frac{o^3}{\log o}\right)$-processor CREW PRAMs.*

Proof. For the graph $G^*(\pi) = (O^*, U^* \cup E)$ defined in Section 3.5 for a job shop problem we introduce the matrix of distances $A = [a_{u,v}]$ with the size $o \times o$, where $a_{u,v}$ is the length of the longest path between vertices u and v. We initiate values $a_{u,v}$ in the following way

$$a_{u,v} = \begin{cases} p_u & \text{if} \quad (u,v) \in U^* \cup E(\pi), \\ 0 & \text{if} \quad (u,v) \notin U^* \cup E(\pi). \end{cases} \tag{5.2}$$

The matrix A will be used for calculation of the longest paths in the graph $G^*(\pi)$. Initial values of the matrix A can be determined in time $O(1)$ using $O(o^2)$ processors, because this requires o^2 independent assignment instructions, each one for every pair (u,v), $u,v = 1,2,\ldots,o$.

The problem of determining cost function value for the $J||\sum f_i$ or $J||f_{\max}$ job shop problems requires finding lengths of the longest paths from the vertex $0 \in U^*$ to vertices $l_1, l_2, \ldots, l_n$ (which corresponds to determination of the following values of job execution finishing times: C_{l_1}, C_{l_2}, ..., C_{l_n}), where n defines the number of jobs, as was defined in Section 3.5. To determine the length of paths it is enough to execute $\lceil \log(o) \rceil$ parallel steps, because in each step $k = 1, 2, \ldots, \lceil \log(o) \rceil$ the algorithm described below updates lengths of the longest paths between vertices with the distance (in the sense of the number of vertices) of at most $1, 2, 4, 8, \ldots, 2^{\log(o)}$. After having executed the $\lceil \log(o) \rceil$ steps the matrix A possesses information about the length of paths between vertices with the distance (in the sense of the number of vertices) of $2^{\log(o)} = o$, that is, between *all* the vertices, because the number of vertices on the longest (in the sense of the number of vertices) path in the graph $G^*(\pi)$ must not be greater than o ($G^*(\pi)$ is an acyclic digraph). For technical needs of the algorithm, an additional three-dimensional table $T = [t_{u,w,v}]$ of the size $o \times o \times o$ is defined. It is used for a transitive closure calculation of $G^*(\pi)$. The algorithm requires execution of the following identical steps $\lceil \log o \rceil$ times:

1. updating $t_{u,w,v}$ for all triples (u, w, v) due to the formula $t_{u,w,v} = a_{u,w} + a_{w,v}$,
2. updating $a_{u,v}$ for all pairs (u, v) on the basis of the equation $a_{u,v} = \max\{a_{u,v}, \max_{1 \le w \le o} t_{u,w,v}\}$.

Step 1 executed on o^3 processors can take the time $O(1)$. On $\lceil o^3 / \log o \rceil$ processors the calculations have to be made $\lceil \log o \rceil$ times, so the computational complexity of this step is $O(\log o)$.

Step 2 consists in determining a maximum of $o + 1$ values, which can be done on $O(o/\log o)$ processors in time $O(\log o)$. As such a maximum should be determined for o^2 pairs (u, v) and these calculations are independent and have to be repeated $\lceil \log o \rceil$ times, therefore using $p = O(o^3/\log o)$ processors the whole algorithm has a computational complexity

$$T_{par}(p) = \lceil \log o \rceil O(\log o) = O(\log^2 o). \tag{5.3}$$

Finally, for the $J||\sum f_i$ problem, all the $f_j(C_{l_j})$, where $C_{l_j} = a_{0,l_j}$, $j \in \mathcal{J}$, should be summarized. These values can be taken from table A. Summation takes the time $O(\log n)$ using $O(n/\log n)$-processor CREW PRAMs keeping computational complexity $O(\log^2 o)$ and the number of processors $O(o^3/\log o)$ for the whole method described, because the number of jobs n is smaller or equals the number of operations o.

Similarly, for the $J||f_{\max}$ problem it is necessary to determine maximum of all values $f_j(C_{l_j})$, $j \in \mathcal{J}$. This step has also computational complexity $O(\log n)$ using

$O(n/\log n)$-processor CREW PRAM machine keeping computational complexity $O(\log^2 o)$ and the number of processors $O(o^3/\log o)$ for the entire method. ■

Corollary 5.1. For the method based on Theorem 5.1 the speedup and the efficiency are

$$S(p) = O\left(\frac{o}{\log^2 o}\right), \quad \eta(p) = O\left(\frac{1}{o^2 \log o}\right). \tag{5.4}$$

Efficiency of the method formulated below quickly decreases with increasing the size of the problem. Therefore, it is not cost-optimal, however computational complexity $O(\log^2 o)$ gives a significant time profit compared to a sequential computational complexity $O(o)$. Additionally, there is no big constant hidden in the notation O. A comparison of the speed of functions increasing for $f(o) = o$ and $f(o) = \log^2 o$ is given in Table 5.1.

Table 5.1. Speed of increasing $f(o) = o$ and $f(o) = \lceil \log^2 o \rceil$ functions.

o	$\lceil \log^2 o \rceil$
10	12
100	45
1,000	100
10,000	177
100,000	276
1,000,000	398
10,000,000	541

Table 5.2 presents times of $C_{\max}$ calculations due to the matrix multiplication based method from Theorem 5.1. The 32-processor nVidia GeForce 9500 GT card (GPU) with CUDA support was used for calculation. The maximum for each pair (u, v) of vertices was calculated in time $O(o)$ using a single processor, because the number of processes $\frac{o^3}{\log o}$ was too big for the hardware used in the experiment. Therefore, the whole parallel algorithm has the computational complexity $O(o \log^2 o)$ instead of $O(\log^2 o)$. Computational experiments shown in Table 5.2 and in Figure 5.2 fully confirm theoretical results.

It is well known that the maximum of n values can be determined on the CRCW (concurrent read, concurrent write) PRAM of a 'common' type (the value is written to the cell of memory if all processes want to write the same value) in constant time $O(1)$ using $O(n^2)$ processors (see Storer [239]). Using this property we can formulate a faster version of the previous method for CRCW PRAMs.

Table 5.2. Times of $C_{\max}$ calculations due to the method from Theorem 5.1 on GPU.

$n \times m$	o	t_p^{av}	$t_p^{\min}$	$t_p^{\max}$	$o \log^2 o$
5×5	25	0.24	0.17	0.61	0.054
10×5	50	0.25	0.2	0.61	0.159
20×5	100	0.32	0.24	0.66	0.441
10×10	100	0.28	0.22	0.61	0.441
20×10	200	0.49	0.41	0.85	1.169
10×20	200	0.66	0.39	4.86	1.169
50×5	250	0.65	0.58	1.06	1.586
20×20	400	1.13	1.02	1.61	2.989
100×5	500	1.94	1.54	8.16	4.019
50×10	500	1.94	1.4	8.1	4.019
10×50	500	2.13	1.48	6.5	4.019
100×10	1,000	4.97	3.58	8.86	9.932
50×20	1,000	4.11	3.46	9.16	9.932
20×50	1,000	4.34	3.41	8.34	9.932
100×20	2,000	14.47	11.96	16.47	24.050
50×50	2,500	21.49	17.6	23.22	31.853
100×50	5,000	71.77	67.85	80.1	75.494

Theorem 5.2. *For a fixed feasible operations order π for the $J||\sum f_i$ or $J||f_{\max}$ problem, the value of the cost function can be determined in time $O(\log o)$ on $O(o^4)$-processor CRCW PRAMs of a 'common' type.*

Proof. We apply identical procedure to that of the previous theorem so that the maximum in Step 2 for each of o^2 pairs (u, v) is determined in constant time $O(1)$ making use of the group of $O(o^2)$ processors connected with the pair (u, v). ∎

It is also possible to formulate the same theorem for the 'sum' type of CRCW PRAMs when the sum of values is written to the cell of memory when multiple processors want to write to the same cell.

Theorem 5.3. *For a fixed feasible operations order π for the $J||\sum f_i$ or $J||f_{\max}$ problem, the value of the cost function can be determined in time $O(\log o \log \Lambda)$ on $O(o^3/\log o)$-processor CREW PRAMs, where Λ is an upper bound of the cost function value.*

Proof. The proof is similar to that of Theorem 5.1. It is enough to repeat the main loop $\lceil \log \Lambda \rceil$ times instead of $\lceil \log o \rceil$ times, because the maximal number of

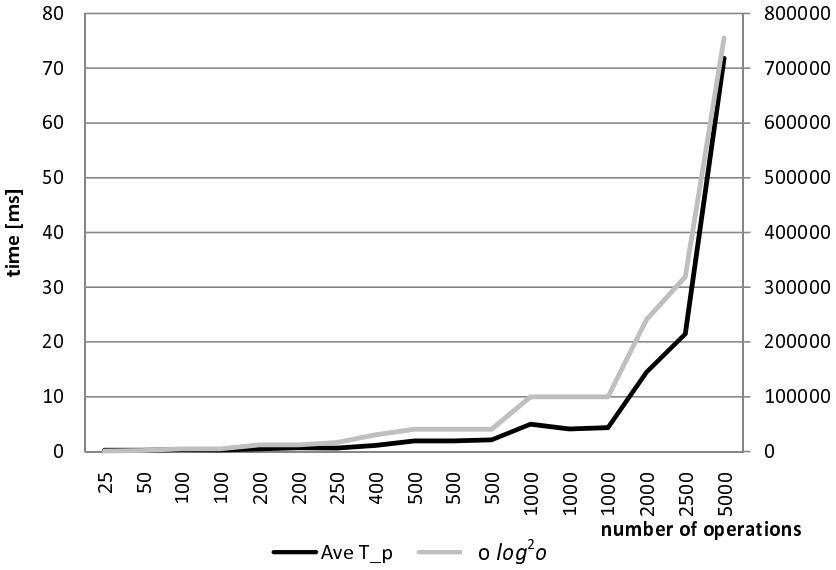


Fig. 5.2. Comparison of execution times of the matrix multiplication based procedure on a 32-processor GPU.

vertices on the critical path is not greater than $\Lambda/p_{\min}$, where $p_{\min}$ is the shortest operation execution time among all operations from the set $\mathcal{O}$. Values of $p_{\min}$ are integers, minimal value equals 1, thus the maximal number of vertices on the critical path is Λ. To determine the length of the critical path it is enough to execute $\lceil \log \Lambda \rceil$ parallel steps (compare Theorem 5.1) which decreases computational complexity of the whole method to $O(\log o \log \Lambda)$ keeping the number of processors $O(o^3/\log o)$. ■

Corollary 5.2. For the method based on Theorem 5.3 we have the speedup and the efficiency

$$S(p) = O\left(\frac{o}{\log o \log \Lambda}\right), \quad \eta(p) = \frac{s_{A_p,M}(p)}{p} = O\left(\frac{1}{o^2 \log \Lambda}\right). \tag{5.5}$$

When the value of the upper bound Λ of all the values of the cost function is known, then the value $\log \Lambda$ is constant and it has no influence upon the computational complexity.

According to studies on popular benchmark instances for the job shop problem (Fisher and Thomson [107], Lawrence [163], Yamada and Nakano [272], Storer et al. [240]) there is a conjecture in the paper of Steinhöfel et al. [237] that for the makespan of the optimal solutions λ_{opt} there exists a uniform upper bound

$\lambda_{opt} \leq p_{\max}(n+m)$, where $p_{\max}$ is the maximal processing time of all operations, n is the number of jobs and m is the number of machines.

5.3.2. Methods based on partitioning into layers

The main problem in obtaining a good speedup value of the methods mentioned above is the fact that a computational complexity of the sequential method of determining makespan value for the job shop problem is $O(o)$. It is, however, difficult to parallelize it because of its recurrent nature. Now we show another approach to determine cost function value, which is more time-consuming, but cost-optimal. First, we need to determine the number of layers d of the graph $G(\pi)$. A sample of layer determination for the conjunctive graph from Figure 3.5 is shown in Figure 5.1.

Theorem 5.4. *For a fixed feasible operations order π for the $J||C_{\max}$ problem, the number of layers from Definition 5.1 of the graph $G(\pi)$ can be calculated in time $O(\log^2 o)$ on the CREW PRAMs with $O\left(\frac{o^3}{\log o}\right)$ processors.*

Proof. Here we use the graph $G^*(\pi)$ with additional vertex 0. Let $B = [b_i j]$ be an incidence matrix for the graph $G^*(\pi)$, i.e., $b_i j = 1$ if there is an arc i, j in the graph $G^*(\pi)$, otherwise $b_i j = 0$, $i, j = 1, 2, \ldots, o$. The proof is given in three steps.

1. Let us calculate longest paths (in the sense of the number of vertices) in $G^*(\pi)$. We can use the algorithm from the proof of Theorem 5.1 with the incidence matrix B instead of the matrix A. We need the time $O(\log^2 o)$ and CREW PRAMs with $O(o^3/\log o)$ processors.

2. We sort distances from the vertex 0 to each vertex in an increasing order. Their indexes after having been sorted correspond to the topological order of vertices. This takes the time $O(\log o)$ and CREW PRAMs with $o+1 = O(o)$ processors, using Cole's parallel merge sort algorithm [80]. We obtain a sequence $Topo[i]$, $i = 0, 1, 2, \ldots, o$.

3. Let us assign each element of the sorted sequence to one processor, without the last one. If the next value of the sequence (distance from 0) $Topo[i+1]$, $i = 0, 1, \ldots, o-1$ is the same as $Topo[i]$ considered by the processor i, we assign $c[i] \leftarrow 1$, and $c[i] \leftarrow 0$ if $Topo[i+1] \neq Topo[i]$. This step requires the time $O(1)$ and o processors. Next, we add all values $c[i]$, $i = 0, 1, \ldots, o-1$. To make this step we need the time $O(\log o)$ and CREW PRAMs with $O(o)$ processors. We get $d = 1 + \sum_{i=0}^{o-1} c[i]$ because there is an additional layer connected with exactly one vertex 0.

The most time- and processor-consuming is Step 1. We need the time $O(\log^2 o)$ and the number of processors $O\left(\frac{o^3}{\log o}\right)$ of the CREW PRAMs. ∎

The result obtained can also be implemented on a CRCW machine in time $O(\log o)$ and the number of processors $O(o^4)$ similarly as described in the proof of Theorem 5.2.

Theorem 5.5. *For a fixed feasible operations order π for the $J||C_{\max}$ problem, the value of cost function can be determined in time $O(d)$ on $O(o/d)$-processor CREW PRAMs, where d is the number of layers of the graph $G(\pi)$.*

Proof. Let Γ_k, $k = 1, 2, \ldots, d$, be the number of calculations of the operations finishing moment C_i, $i = 1, 2, \ldots, o$ in the k-th layer. Certainly $\sum_{i=1}^{d} \Gamma_i = o$. Let p be the number of processors used. The time of computations in a single layer k after having divided calculations into $\lceil \frac{\Gamma_i}{p} \rceil$ groups, each group containing (at most) p elements, is $\lceil \frac{\Gamma_i}{p} \rceil$ (the last group cannot be full). Therefore, the total computation time in all d layers equals $\sum_{i=1}^{d} \lceil \frac{\Gamma_i}{p} \rceil \leq \sum_{i=1}^{d} (\frac{\Gamma_i}{p} + 1) = \frac{o}{p} + d$. To obtain the time of computations $O(d)$we should use $p = O(\frac{o}{d})$ processors. ∎

This theorem provides a cost-optimal method of parallel calculation of the cost function value for the job shop problem with the makespan criterion. For example, if we discuss a classic permutational flow shop problem with n jobs and m machines, which is a special case of the job shop problem, we can observe that $d = n + m - 1$ which is the length (as the number of vertices) of the longest path in the graph $G(\pi)$, then, we get the cost-optimal method which works in time $O(n+m)$ on $O(nm/(n+m))$-processor CREW PRAM machine for the flow shop scheduling problem. The sequence of calculations is presented in Figure 5.3.

Practical aspects of the cost function value determination. Step 1 consists in determining the longest paths (in the sense of vertex number) in the graph. We are interested in the lengths of paths from the vertex 0 to each other vertex. In Step 2 we should sort the obtained lengths. We make use of a two-row table and we sort it with reference to the second row together with the first row

$$\begin{bmatrix} 1 & 2 & 3 & 4 & \ldots & o \\ C_1 & C_2 & C_3 & C_4 & \ldots & C_o \end{bmatrix},$$

which can be obtained using $O(o)$ processors in time $O(1)$ (each processor writes its own number i and C_i), i.e., for the sample from Figure 5.1

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 3 & 6 & 5 & 1 & 5 & 2 & 3 & 7 & 6 \end{bmatrix}.$$

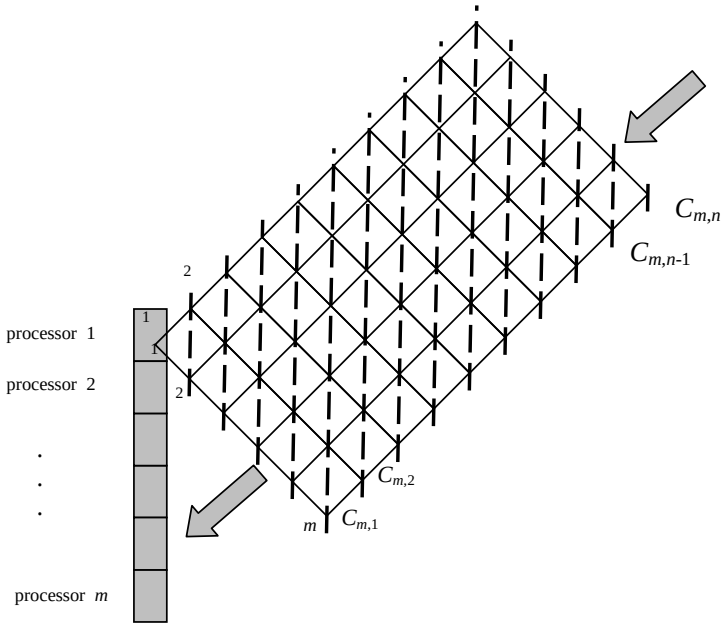


Fig. 5.3. A layer-based sequence of C_{ij} calculations for the *flow shop* – a special case of the job shop problem.

Afterwards we obtain the sorted second row, however there are numbers of corresponding vertices in the first row

$$\begin{bmatrix} 1 & 5 & 7 & 2 & 8 & 3 & 4 & 6 & 10 & 9 \\ 1 & 1 & 2 & 3 & 3 & 4 & 5 & 5 & 6 & 7 \end{bmatrix}.$$

This two-row table will be called $Topo[1..o][1..2]$. To avoid concurrent readings we can multiply this table to the second, identical, table $Topo2[1..o][1..2]$ using $O(o)$ processors in constant time $O(1)$. Each processor $i = 1, 2, \ldots, o-1$ compares a value $Topo[i][1]$ with $Topo2[i+1][1]$. If $Topo[i][1] < Topo2[i+1][1]$ then a processor generates (writes to a variable c) 1, otherwise it generates 0. To determine the number of layers d it is enough to get the last $Topo$ value: $d = Topo[o][1]$ because it corresponds to the longest path (understood as the number of vertices) from the vertex 0 (table $Topo$ is sorted). In our sample the number of layers $d = 7$. For the parallel algorithm to determine the cost function value it is necessary to know what is the index of the first vertex in the layer, how many vertices belong to each layer and what are their numbers. Such a table $first\_in[1..o]$ of the first vertices in each layer can be created in time $O(1)$ as follows: each processor which generates 1 (i.e., its $c = 1$) writes to the table $first\_in[Topo[i][1]] := i$. In this way we obtain the d-elementary table $first\_in$ from which, together with the table $Topo$, we can get all the information of layers.

However, in general, if we have no knowledge of the number of layers d, we can employ the following estimation.

Theorem 5.6. *For a fixed feasible operations order π for the $J||C_{\max}$ problem, the value of cost function can be determined in time $O(\Lambda)$ on $O(o)$-processor CREW PRAMs, where Λ is an upper bound of the cost function value.*

Proof. Let us observe that the number of layers d corresponds to the number of vertices on the longest (in the sense of the number of vertices) path in $G^*(\pi)$. This number d can be bounded by $C_{\max}/p_{\min}$ ($p_{\min}$ is a minimal processing time of all operations), where $C_{\max}$ is the length (as a sum of weights) of the longest path in $G^*(\pi)$. The proof is 'a contrario'.

Let us assume that there is a path with more than $C_{\max}/p_{\min}$ vertices. Therefore, its length (as a sum of weights) would be greater than $C_{\max}$, which is impossible because $C_{\max}$ is the length of the longest path in $G^*(\pi)$. We can use the upper bound of the cost function Λ instead of $C_{\max}$, as well as a minimal $p_{\min}$ value which is 1, because $C_{\max}/p_{\min} < \Lambda/1$ and we are looking for the upper bound. The other part of the proof is similar to the proof of Theorem 5.5, as regards the time $d \leq \Lambda$ estimation and the number of processors $\frac{o}{d} \leq o$. ∎

From the above theorem a surprising conclusion can be drawn, namely: if we can determine the upper bound of the cost function value Λ, the calculations take constant time $O(\Lambda) = O(1)$. The trivial upper bound Λ of the makespan is the sum of the processing times of all operations. Although the algorithm for determining the d value has computational complexity $O(\log^2)$, it can be executed only once, at the very beginning. Next, one can calculate only how the d value is changing after having executed an *insert* or *swap* move, and this can be done in constant time $O(1)$.

5.4. Remarks and conclusions

In this chapter, there were designed new methods of parallel goal function value calculation for a given job execution sequence in the job shop problem. Considering the computational complexity $O(o)$ for the sequential case, the new parallel methods have been proposed with significantly lower computational complexity $O(\log^2 o)$, $O(\log o \log \Lambda)$, $O(d)$ and $O(\Lambda)$, where o is the number of operations, Λ is an upper bound of the cost function value and d is the number of layers created during the work of a topological sort algorithm. For the first time there was proposed a method with complexity $O(\Lambda) = O(1)$, if the Λ value can be estimated. In particular, the first of the above algorithms, with complexity $O(\log^2 o)$, is more effective than is known from the literature [237]. What is more, the results

obtained remain valid for $f_{\max}$ criterion as well as for f_{sum} criterion. The proposed methodology of the single-walk parallelization is based on using the parallel path determination in graphs as well as genuine, dedicated methods, and it can be easily extended to flexible scheduling problems such as the job shop problem with parallel machines.

Chapter 6

Hybrid scheduling

The aim of this chapter is to show how to determine the neighborhood and how to search it in the parallel environment, this being illustrated by an example of the hybrid scheduling, more precisely a flexible job shop problem. We present a parallel single-walk approach in this respect. A theoretical analysis based on PRAM model of parallel computing has been made. We propose a cost-optimal method of neighborhood generation parallelization.

6.1. Solution method

There is an exponential number of possible job-to-machine assignments in relation to the number of operations. Each feasible assignment generates an NP-hard problem (job shop) whose solution consists in determining an optimal job processing order on machines. One has to solve an exponential number of NP-hard problems to solve the flexible job shop problem. Therefore, we shall apply approximate algorithms consisting in executing the following two steps:

Step 1: Job-to-machine assignment determination;
Step 2: Solving a job shop problem for the assignment determined in Step 1.

We use the tabu search algorithm in Step 1. The neighborhood of the current solution (assignment) is generated by jobs moving between machines of the same type. The best element of this neighborhood generates a job shop problem which is solved in Step 2. For comparison, we also present an algorithm which uses population-based method in Step 1, without the jobs moving between machines of the same type.

6.2. Machine workload

The problem of 'good' (suboptimal) operation-to-machine determination is considered in this section. Each operation is assigned to one nest only. It is necessary to make a partition of operations assigned to machines in each nest. The method of partitioning has an influence on all jobs completion time, that is the value of the job shop problem solution. Generally, we are looking for such a partition whose cost function value (of the corresponding job shop problem) is minimal.

Let $\Theta = (\mathcal{Q}, \pi(\mathcal{Q})) \in \Phi^\circ$ be a feasible solution of FJSP where $\mathcal{Q} = (\mathcal{Q}^1, \mathcal{Q}^2, \ldots, \mathcal{Q}^m)$ is the machine workload, ϱ_i is the number of operations executed on machine M_i (i.e., $\varrho_i = |\mathcal{Q}^i|$) and

$$\pi(\mathcal{Q}) = (\pi_1(\mathcal{Q}), \pi_2(\mathcal{Q}), \ldots, \pi_m(\mathcal{Q})) \tag{6.1}$$

is a concatenation of m permutations. A permutation $\pi_i(\mathcal{Q})$ determines a sequence of operations from the set $\mathcal{Q}^i$ which have to be processed on machine M_i $(i = 1, 2, \ldots, m)$.

In the further part of this section, in the case in which it does not evoke ambiguity, we omit the assignment of operations $\mathcal{Q}$ which occurs as a permutation parameter. Thus, the concatenation $\pi(\mathcal{Q}) = (\pi_1(\mathcal{Q}), \pi_2(\mathcal{Q}), \ldots, \pi_m(\mathcal{Q}))$ will be presented as $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$.

By $t^i_j(k, l)$ we denote a *transfer* type move (a *t-move*, for short) which consists in moving an operation from the position k in the permutation π_i (i.e., the operation $\pi_i(k)$) to the position l in the permutation π_j (moving operations from positions l, $l+1$, ... one position to the right). The execution of the move $t^i_j(k, l)$ generates from $\Theta = (\mathcal{Q}, \pi) \in \Phi^\circ$ (by Φ° we denote a set of feasible solutions of the FJSP, see Section 3.6) a new solution $\Theta' = (\mathcal{Q}', \pi')$ such that

$$\pi'_v = \pi_v, \quad v \neq i, j, \quad v = 1, 2, \ldots, m \tag{6.2}$$

and

$$\pi'_i = (\pi_i(1), \pi_i(2), \ldots, \pi_i(\boldsymbol{k-1}), \pi_i(\boldsymbol{k+1}), \ldots, \pi_i(\varrho_i - 1), \tag{6.3}$$

$$\pi'_j = (\pi_j(1), \pi_j(2), \ldots, \pi_j(\boldsymbol{l-1}), \pi_i(\boldsymbol{k}), \pi_j(\boldsymbol{l}), \ldots, \pi_j(\varrho_j + 1). \tag{6.4}$$

The execution of this move causes a movement of the operation $\pi_i(k)$ from the set $\mathcal{Q}^i$ (i.e., from machine M_i) to the set $\mathcal{Q}^j$ (i.e., to machine M_j). Therefore

$$\mathcal{Q}'^v = \mathcal{Q}^v, \quad v \neq i, j, \quad v = 1, 2, \ldots, m \tag{6.5}$$

and

$$\mathcal{Q}'^i = \mathcal{Q}^i \setminus \{\pi_i(k)\}, \quad \mathcal{Q}'^j = \mathcal{Q}^j \cup \{\pi_i(k)\}. \tag{6.6}$$

The graph $G(\Theta')$ generated by a *t-move* can have a cycle and then the solution $\Theta' = (\mathcal{Q}', \pi')$ is not feasible.

Remark 6.1. *An upper bound of the number of t-moves is* $O(qm^2o^2)$.

For data from Example 3.1, an upper bound of the number of *t-moves* equals 26244.

Example 6.1. The second critical path described in Example 3.4 $(s, O_1, O_2, O_3, O_6, O_9, c)$ includes a block of three elementary operations $B^1 = (O_3, O_6, O_9)$ executed in the nest $\mathcal{M}^3$ on the machine M_5. There is another machine M_6 in this nest to which no job is assigned. Therefore, for the assignment $\mathcal{Q}$ described in Example 3.2, the operations set (assigned to the machine M_5) $\mathcal{Q}^5 = \{O_3, O_6, O_9\}$ and to the machine M_6, $\mathcal{Q}^6 = \emptyset$. We are moving the operation O_6 from the block B^1 onto the machine M_5 (thus from the set Q^5) to the set of operations which are executed on the machine M_6 (therefore to the set $\mathcal{Q}^6$). In this way, we generate a new job-to-machine assignment $\mathcal{Q}'$ from the assignment $\mathcal{Q}$. In the new assignment $\mathcal{Q}'$ there are $\mathcal{Q}'^5 = \mathcal{Q}^5 \setminus \{O_6\} = \{O_3, O_9\}$, $\mathcal{Q}'^6 = \mathcal{Q}^6 \cup \{O_6\} = \{O_6\}$, with other sets in both assignments being the same. ■

The execution of *t-move* causes an operation to transfer from one machine to another, i.e., a new machine workload in the nest. Therefore, it is possible to obtain any machine workload from any solution (machine workload) by executing *t-moves*. If τ is a *t-move*, we denote by $\tau(\Theta)$ a solution generated from Θ by executing a move τ (see (6.2)–(6.6)). For a fixed feasible solution Θ, let $\mathcal{T}(\Theta)$ be a set of all *t-moves*. A *neighborhood* Θ is a set

$$\mathcal{N}(\Theta) = \{\tau(\Theta) \in \Phi^\circ : \tau \in \mathcal{T}\}, \tag{6.7}$$

where Φ° is a set of feasible solutions. The feasibility $\tau(\Theta)$ corresponds to the acyclicity of the graph $G(\tau(\Theta))$.

It was mentioned at the beginning of this chapter that the solution of the FJSP consists of two steps. The first, determination of machine workload and the second, determination of processing order of operations, i.e., a job shop problem solving. Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution of the FJSP. The new machine workload $\mathcal{Q}'$ will be generated from the workload $\mathcal{Q}$ as follows:

- determine a neighborhood $\mathcal{N}(\Theta)$,
- select from the neighborhood a solution $\Theta' = (\mathcal{Q}', \pi')$ with the lowest goal function value – the new machine workload $\mathcal{Q}'$.

The number of all *t-moves* can be huge, so we omit some of them and consider only those which can offer an improvement of the goal function value. Moreover, we do not determine an exact goal function value of solutions generated by

t-moves, but approximate them only. As computational experiments proved, this causes a significant algorithm work acceleration with little results aggravation. In the further part, we precisely describe methods of eliminating superfluous moves from the neighborhood (6.7) as well as methods of estimating a goal function value.

6.2.1. Neighborhood determination

Execution of a *t-move* can lead to a non-feasible solution, i.e., a graph connected with this solution can have a cycle. Therefore, checking feasibility equals checking if a graph has a cycle. The corresponding algorithm has a computational complexity $O(o)$ where o is the number of all operations. Further on we prove theorems which make it possible to check feasibility of solutions (i.e., acyclicity of corresponding graphs) generated by *t-moves* in constant time.

Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution where $\mathcal{Q} = (\mathcal{Q}^1, \mathcal{Q}^2, \ldots, \mathcal{Q}^m)$ is the machine workload and $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ is a concatenation of m permutations. A permutation π_i determines a processing order of operations from the set $\mathcal{Q}^i$ on the machine M_i $(i = 1, 2, \ldots, m)$.

We consider two machines M_i and M_j from the same nest. A permutation π_i determines a processing order of operations from the set $\mathcal{Q}^i$ on the machine M_i and π_j – a processing order of operations from the set $\mathcal{Q}^j$ on the machine M_j. For an operation $\pi_i(k) \in \mathcal{Q}^i$ we define two parameters connected with paths in the graph $G(\Theta)$.

The first parameter is

$$\eta_j(k) = \begin{cases} 1, \text{ if there is no path } C(\pi_j(v), \pi_i(k)) \; \forall\, v = 1, 2, \ldots, \varrho_j, \\ 1 + \max\limits_{1 \le v \le \varrho_j} \{\text{there is a path } C(\pi_j(v), \pi_i(k))\}, \text{ otherwise.} \end{cases} \tag{6.8}$$

Thus, there does not exist any path to the operation (vertex) $\pi_i(k)$ from any of the operations placed in the permutation π_j in positions $\eta_j(k)$, $\eta_j(k)+1, \ldots, \varrho_j$ (where $\varrho_j = |\mathcal{Q}^j|$) in the graph $G(\Theta)$. This situation is shown in Figure 6.1.

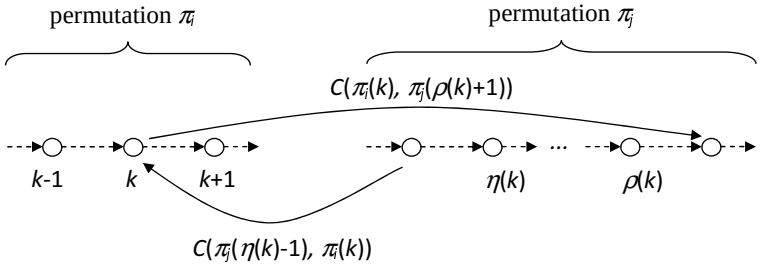


Fig. 6.1. Visualization of parameters $\eta_j(k)$ and $\rho_j(k)$ for an operation $\pi_i(k)$.

The second parameter is

$$\rho_j(k) = \begin{cases} 1+\varrho_j, \text{ if there is no path } C(\pi_j(v),\pi_i(k))\ \forall\, v=1,2,\ldots,\varrho_j, \\ 1+\min\limits_{\eta_j(k)\le v\le\varrho_j}\{\text{there is a path } C(\pi_i(k),\pi_j(v))\}, \text{ otherwise.} \end{cases} \tag{6.9}$$

From the definition formulated above it follows that in the graph there is no path from a vertex $\pi_i(k)$ to any operation placed in positions $\eta_j(k), \eta_j(k)+1, \ldots, \rho_j(k)$ in the permutation π_j (see Figure 6.1). Now we prove two theorems characterizing a *t-move* whose execution generates an unfeasible solution. These theorems constitute a constructional base for very efficient neighborhoods. The structure of assumptions allows an easy implementation in the parallel computing environment, such as GPUs.

Theorem 6.1. *Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution for the* FJSP *and let π_i, π_j be permutations of operations executed on machines M_i, M_j. If machines M_i, M_j belong to the same nest then executing a t-move $t^i_j(k,l)$ ($\pi_i(k) \in \mathcal{Q}^i$, $l = 1,2,\ldots,$ $\eta_j(k)-1$) generates a solution which is not feasible.*

Proof. Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution and let $G(\Theta)$ be a corresponding graph. The permutation π_i determines operations executing order on machines M_i and π_j – an order on machine M_j. We consider a *t-move* $t^i_j(k,l)$ consisting in an operation $\pi_i(k)$ transfer from a machine M_i to a position l ($1 \le l \le \eta_j(k)-1$) in the permutation π_j, i.e., to the machine M_j. This move generates a new solution $\Theta' = (\mathcal{Q}', \pi')$ (see (6.2–6.6) and a corresponding graph $G(\Theta')$. Now, we prove that this graph includes a cycle.

From definition (6.8) of the parameter $\eta_j(k)$ it follows that in the graph $G(\Theta)$ there exists a path from the vertex $\pi_j(\eta_j(k)-1)$ to $\pi_i(k)$, i.e., the path $C(\pi_j(\eta_j(k)-1), \pi_i(k))$, as shown in Figure 6.2.

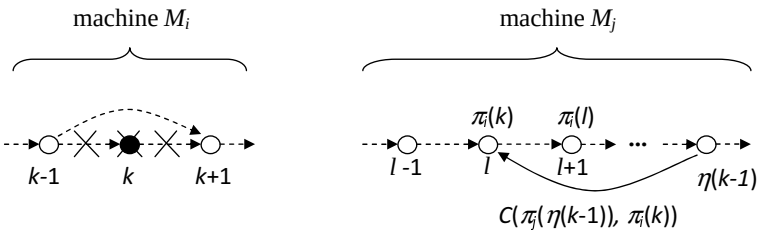


Fig. 6.2. Directed graph $\Theta' = G(t^i_j(k,l)(\Theta)) = G(\mathcal{Q}', \pi')$.

Moreover, there is also a path $C(\pi_j(l)) = (\pi_i(k), \pi_j(\eta_j(k)-1))$ in this graph. Executing a move $t^i_j(k,l)$ causes an insertion of the operation $\pi_i(k)$ in position l to the permutation π_j. As a result, an arc $(\pi_i(k), \pi_j(l))$, among others, is inserted

to the graph $G(\Theta')$. This creates a cycle

$$((\pi_i(k),\pi_j(l)),C(\pi_j(l),\pi_j(\eta_j(k)-1)),C(\pi_j(\eta_j(k)-1),\pi_i(k))),$$

which completes the proof of the theorem. ■

A similar theorem can be proved for the $\rho_j(k)$ parameter.

Theorem 6.2. *Let $\Theta=(\mathcal{Q},\pi)$ be a feasible solution for the* FJSP *and π_i,π_j be permutations of operations executing on machines M_i,M_j. If machines M_i,M_j belong to the same nest then executing a t-move $t^i_j(k,l)$ (where $\pi_j(k)\in\mathcal{Q}^i$, $l=\rho_j(k)+1,\rho_j(k)+2,\ldots,\varrho_j$) generates a solution which is not feasible.*

Proof. Similarly as in the proof of Theorem 6.1 one can show that after having executed a move $t^i_j(k,l)$ $(l=\rho_j(k)+1,\rho_j(k)+2,\ldots,\varrho_j)$ there appears a cycle $(C(\pi_i(k),\pi_j(\rho_j(k)-l)),C(\pi_j(\rho_j(k)-l),\pi_j(l-1)),(\pi_j(l-1),\pi_i(k))$ in the generated graph $G(\Theta')$. ■

Let us denote by $\mathcal{T}^{noacc}$ a set of the *t-moves* from $\mathcal{T}(\Theta)$ which fulfill assumptions of Theorems 6.1 or 6.2. Therefore, the moves generate non-feasible solutions from Θ.

Theorem 6.3. *Let $\Theta=(\mathcal{Q},\pi)$ be a feasible solution of the* FJSP *and π_i,π_j be permutations of operations executed on machines M_i,M_j. If machines M_i,M_j belong to the same nest then executing a t-move $t^i_j(k,l)$ $(l=\eta_j(k),\ \eta_j(k)+1,\ldots,\rho_j(k))$ generates a feasible solution.*

Proof. The proof of this theorem follows directly from the definition of parameters $\eta_j(k)$, $\rho_j(k)$ $(\pi_i(k)\in\mathcal{Q}^i)$ and Theorems 6.1 and 6.2. ■

Property 6.1. *For each operation $\pi_i(k)$ executed on machine M_i there exists a position l in the permutation π_j (i.e., on machine M_j from the same nest) such that executing a move $t^i_j(k,l)$ generates a feasible solution from a solution Θ.*

Proof. It is enough to observe that if Θ is a feasible solution hence for an operation $\pi_i(k)$ there is $\rho_j(k)\geq\eta_j(k)$. ■

Now we prove a theorem which constitutes a base for eliminating some *t-moves* during the process of neighborhood generation. Its function is similar to that of Theorem 3.4. Let us assume that for an assignment $\mathcal{Q}$ the concatenation π is the optimal operations schedule on machines, $G(\Theta)$ is a graph connected with solution $\Theta=(\mathcal{Q},\pi)$ and $C_{\max}(\Theta)$ is a cost function value, i.e., the critical path length in the graph $G(\Theta)$.

Theorem 6.4. *Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution for the* FJSP *and let $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ be a sequence of critical path blocks in the graph $G(\Theta)$. If $\Theta' = (\mathcal{Q}', \pi')$ is a feasible solution which was generated from Θ by machine workload changing in a nest and $C_{\max}(\Theta') < C_{\max}(\Theta)$ therefore in the Θ' at least one operation from some block was moved to a different machine (in the same nest).*

Proof. Let $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ be a sequence of critical path blocks in the graph $G(\Theta)$. Each block is an operation sequence

$$B^i = (\pi(a^i), \pi(a^i + 1), \ldots, \pi(b^i)), \tag{6.10}$$

for $i = 1, 2, \ldots, r$ where $1 \leq a^1 \leq b^1 < a^2 \leq b^2 <, \ldots, < a^k \leq b^k$. For a notion simplification we assume (in the proof of this theorem) that each operation from a critical path belongs to some block. Thus, a block can consist of a single operation. By

$$\mathcal{Y}^i(\pi) = \{\pi(a^i), \pi(a^i + 1), \ldots, \pi(b^i)\}, \tag{6.11}$$

we denote a set of jobs from the block B^i. The critical path $C(s, c)$ in the graph $G(\Theta)$ includes all vertices (operations) of a set $\bigcup_{i=1}^{r} \mathcal{Y}^i$ and its length $L(s, c) = C_{\max}(\Theta) = \sum_{i=1}^{r} \sum_{v \in \mathcal{Y}^i} p_v$.

Let $\Theta' = (\mathcal{Q}', \pi')$ be a feasible solution such that $C_{\max}(\Theta') < C_{\max}(\Theta)$. Let us assume that no operations from any block $B^1, B^2, \ldots, B^r$ in the workload $\mathcal{Q}'$ are moved to another machine from the same nest. Thus

$$\mathcal{Y}^i(\pi) = \mathcal{Y}^i(\pi'), \quad i = 1, 2, \ldots, r. \tag{6.12}$$

Therefore, a sequence of jobs $(\pi(a^i), \pi(a^i+1), \ldots, \pi(b^i))$ in the permutation π and $(\pi'(a^i), \pi'(a^i + 1), \ldots, \pi'(b^i))$ in π' are permutations of the same job subsequence $\mathcal{Y}^i = \{\pi(a^i), \pi(a^i+1), \ldots, \pi(b^i)\}$. We consider a path $C'(s, c)$ in the graph $G(\Theta')$. Vertices of this path belong to a set $\mathcal{A} = \bigcup_i^r \mathcal{Y}^i(\pi')$. The length of this path $L'(s, c) = \sum_{v \in \mathcal{A}} p_v$ so it equals the length $L(s, c) = C_{\max}(\Theta)$ of the critical path $C(s, c)$ in the graph $G(\Theta)$. Therefore $C_{\max}(\Theta') \geq C_{\max}(\Theta)$, which contradicts the assumption. ∎

Let Θ be a feasible solution, $\mathcal{B}$ – a sequence of critical path blocks in the graph $G(\Theta)$ and $\mathcal{T}$ – a set of *t-moves* defined for the Θ. We denote by $\mathcal{T}^{out}(\Theta)$ a set of those moves from $\mathcal{T}(\Theta)$ which consider operations not belonging to any block. Directly from Theorem 6.4 there follows a property which constitutes a base for eliminating superfluous moves.

Property 6.2. *If a feasible solution Θ' is generated from Θ by executing a t-move belonging to the set $\mathcal{T}^{out}(\Theta)$ then*

$$C_{\max}(\Theta') \geq C_{\max}(\Theta). \tag{6.13}$$

Proof. The proof results directly from Theorem 6.4. ∎

Therefore, executing a *t-move* that consists in moving an operation not lying on the critical path to another machine does not generate a solution with lower cost function value.

Theorem 6.5. *Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution for the* FJSP*. If B^u is a block on the machine M_i and B^v is a block on M_j and both machines belong to the same nest then a transfer type move consisting in moving an operation from an internal block B^u to the internal block B^v does not generate a solution with lower cost function value.*

Proof. Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution, $G(\Theta)$ – a graph connected with it and $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ – block of the critical path sequence. We assume that

$$B^u = (\pi(a^u), \pi(a^u + 1), \ldots, \pi(b^u)), \quad B^v = (\pi(a^v), \pi(a^v + 1), \ldots, \pi(b^v)),$$

are blocks $(1 \leq u < v \leq r)$ on machines M_i and M_j, respectively. We consider a *t-move* $t^i_j(k, l)$ where $a^u < k < b^u$ and $a^v < l < b^v$ moving the operation from the block B^u to the block B^v. This move generates from Θ a new solution $\Theta' = (\mathcal{Q}', \pi')$. The machine workload $\mathcal{Q}'$ as well as the permutation π' are defined in (6.2)–(6.6). The critical path $C(s, c)$ in the graph $G(\Theta)$ can be partitioned as follows

$$C(s, c) = (C(s, \pi(a^u)), C(\pi(a^u), \pi(b^u)),$$
$$C(\pi(b^u), \pi(a^v)), C(\pi(a^v), \pi(b^v)), C(\pi(b^v), c)). \tag{6.14}$$

This path is shown in Figure 6.3.

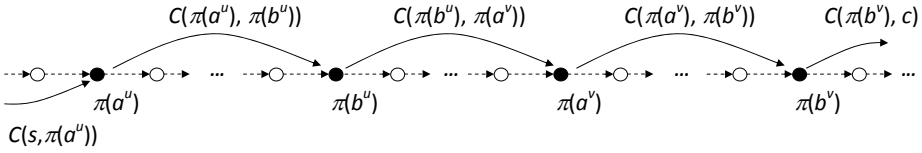


Fig. 6.3. Critical path in the graph $G(\mathcal{Q}, \pi)$.

There is a path

$$C'(s, c) = (C'(s, \pi'(a^u)), C'(\pi'(a^u), \pi'(b^u)),$$
$$C'(\pi'(b^u), \pi'(a^v)), C'(\pi'(a^v), \pi'(b^v)), C'(\pi'(b^v), c)) \tag{6.15}$$

in the graph $G(\Theta')$. It is easy to observe that the following paths are the same

$$C'(s, \pi'(a^u)) = C(s, \pi(a^u)), \quad C'(\pi'(b^u), \pi'(a^v)) = C(\pi(b^u), \pi(a^v)) \tag{6.16}$$

and

$$C'(\pi'(b^v), s) = C(\pi(b^v), s), \tag{6.17}$$

so their lengths are also equal.

We consider paths $C(\pi(a^u), \pi(b^u))$ and $C(\pi(a^v), \pi(b^v))$ in the graph $G(\Theta)$, and $C'(\pi(a^u), \pi(b^u))$ and $C'(\pi(a^v), \pi(b^v))$ in the graph $G(\Theta')$. Because the path $C'(\pi'(a^u), \pi'(b^u))$ includes all vertices of the path $C(\pi(a^u), \pi(b^u))$ excluding the vertex $\pi(k)$ which was moved by the *t-move* from the block B^u, therefore the path length is

$$L'(\pi'(a^u), \pi'(b^u)) = L(\pi(a^u), \pi(b^u)) - p_{\pi(k)}. \tag{6.18}$$

Similarly, considering paths $C(\pi(a^v), \pi(b^v))$ and $C'(\pi(a^v), \pi'(b^v))$ one can show that

$$L'(\pi'(a^v), \pi'(b^v)) = L(\pi(a^v), \pi(b^v)) + p_{\pi(k)}. \tag{6.19}$$

From this it follows that $L'(c, s) = L(c, s)$.

Summing up, the length of some path $C'(c, s)$ in the graph $G(\Theta')$ equals $C(s, t) = C_{\max}(\Theta)$. Accordingly, $C_{\max}(\Theta') \geq C_{\max}(\Theta)$, which completes the proof of the theorem. ■

Therefore, to generate a better solution by executing a *t-move* one should move the first or the last operation of the block before the first or after the last operation of another block. Theorems 6.1 and 6.2 proved in this section concern feasibility of solutions generated by *t-moves*. If paths between any pair of vertices in the graph are known, then this check is executed in constant time. Moreover, Theorems 6.4 and 6.5 defined the so-called blocks elimination criteria. That is why they make it possible to omit in the generation procedure those moves which do not generate better solutions than the current ones. From the set of moves $\mathcal{T}(\Theta)$ generating the neighborhood of the solution Θ we omit all moves which fulfill assumptions of Theorems 6.1, 6.2, 6.4 and 6.5. Therefore, *t-moves* from

$$\mathcal{T}^{acc}(\Theta) = \mathcal{T}(\Theta) \setminus (\mathcal{T}^{noacc} \cup \mathcal{T}^{out}) \tag{6.20}$$

will be used for creation of neighborhood Θ, where $\mathcal{T}^{noacc}$ is the set of moves generating non-feasible solutions (Theorems 6.1 and 6.2) and $\mathcal{T}^{out}$ is the set of moves generating solutions with the cost function value lower than $C_{\max}(\Theta)$ (Theorems 6.4 and 6.5).

If $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ is a block from the critical path sequence in the graph $G(\Theta)$ then the set $\mathcal{T}^{acc}(\Theta)$ includes moves which transfer the first (or the last) operation of each block from the machine M_i to another machine (from the same nest). If $\pi(v)$ is the first (or the last) operation of a block and M_j is the machine from the same nest then the set $\mathcal{T}^{acc}(\Theta)$ includes moves which transfer $\pi(v)$ to

the following positions: $\eta_j(v), \eta_j(v+1), \ldots, \rho_j(v)$. Such a neighborhood has a big size, so we have limited moves which transfer the first (or the last) operation $\pi(v)$ of a block *only* in the position $\eta_j(v)$ or $\rho_j(v)$. So, ultimately

$$\mathcal{T}^{subm}(\Theta) = \{t_j^i(v,w) \in \mathcal{T}^{acc} : \ v \in \{a^k, b^k\},$$

$$w \in \{\eta_j(v), \rho_j(v)\}, \ k = 1, 2, \ldots, r. \tag{6.21}$$

The neighborhood Θ constitutes the set of feasible solutions

$$\mathcal{N}(\Theta) = \{\tau(\Theta) : \ \tau \in \mathcal{T}^{subm}(\Theta)\}. \tag{6.22}$$

This neighborhood has the size $O(r \cdot m)$, where r is the number of the critical path blocks.

In practice, we should select the best element of the neighborhood (e.g. inside a metaheuristic). Making use of parallel computing environment we can follow one of the approaches below.

- We get as many processors as there are blocks r. Next, in the loop, each processor checks the cost of the operation insertion to another machine of the same type, concurrently calculating the minimal value of such an insertion.
- We get as many processors as there are pairs of machines of the same type. We calculate the minimal value in logarithmic time using a tree scheme of parallel calculations.

The first approach leads to the cost-optimal method. However, the second approach, using much greater number of processors, leads us to obtaining shorter computing time.

6.2.2. Methods of the cost function value estimation

Each solution $\Theta = (\mathcal{Q}, \pi)$ is a pair whose first element is a set sequence – machine workloads. A new assignment will be determined by choosing an element with the lowest cost function value from the neighborhood (6.22). This requires a critical path to be determined for each element of the neighborhood. To speed this procedure up, we will compute the lower bounds of the cost function value as a choosing criterion. In this section, we will present methods of determining lower bounds.

Let $\Theta = (\mathcal{Q}, \pi)$ be a feasible solution where $\mathcal{Q} = (\mathcal{Q}^1, \mathcal{Q}^2, \ldots, \mathcal{Q}^m)$ constitutes machine workload and $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ is a concatenation of m permutations. Further, let $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ be a sequence of critical path blocks in the graph $G(\Theta)$. We consider two machines M_i and M_j belonging to the same nest.

On machine M_i there are executed operations from the set Q^i in the order $\pi_i = (\pi_i(1), \pi_i(2), \ldots, \pi_i(\varrho_i))$ and on machine M_j operations from the set Q^j in the order $\pi_j = (\pi_j(1), \pi_j(2), \ldots, \pi_j(\varrho_j))$. Let us assume that the block

$$B^k = (\pi_i(a^k), \pi_i(a^k + 1), \ldots, \pi_i(b^k - 1), \pi_i(b^k)), \tag{6.23}$$

includes operations executed on machine M_i. For simplicity, we omit index k which denotes the block number. Therefore, $\pi_i(a)$ is the first and $\pi_i(b)$ is the last operation of the block B^k.

According to the strategy of searching neighborhood $\mathcal{N}(\Theta)$ we are looking for such a move $\tau \in \mathcal{T}^{subm}(\Theta)$ which generates a graph $G(\tau(\Theta))$ – a feasible solution with possibly the lowest estimation of the critical path length (i.e., cost function value). For moves from $\mathcal{T}^{subm}(\Theta)$ which transfer the first operation $\pi_i(a^k)$ of the block B^k to position $\eta_j(a^k)$ or $\rho_j(a^k)$ in the permutation π_j we introduce the notions

$$\Delta^{a^k}_{\mathrm{x}(a^k)} = \max\{L_1^{\mathrm{x}(a^k)}, L_2^{\mathrm{x}(a^k)}, L_3^{\mathrm{x}(a^k)}, L_4^{\mathrm{x}(a^k)}\}, \;\; \mathrm{x}(a^k) \in \{\eta_j(a^k),\ \rho_j(a^k)\}, \tag{6.24}$$

where

$$L_1^{\mathrm{x}(a^k)} = L(s, \pi_i(a^k - 1)) - L(s, \pi_i(a^k)), \tag{6.25}$$

$$L_2^{\mathrm{x}(a^k)} = L(s, \pi_i(a^k + 1)) - L(s, \pi_i(a^k)) - p_{\pi_i(a^k+1)}, \tag{6.26}$$

$$\begin{aligned} L_3^{\mathrm{x}(a^k)} = {} & L(s, \pi_j(1)) + \sum_{h=2}^{\varrho_j - 1} p_{\pi_i(h)} + p_{\pi_i(a^k)} + L(\pi_j(\varrho_j), c) + \\ & - L(s, \pi_i(a^k)) - \sum_{h=a^k+1}^{b^k-1} p_{\pi_i(h)} - L(\pi_i(b^k), c), \end{aligned} \tag{6.27}$$

$$\begin{aligned} L_4^{\mathrm{x}(a^k)} = {} & \sum_{h=\mathrm{x}(a^k)+1}^{\varrho_j - 1} p_{\pi_j(h)} + L(\pi_j(\varrho_j), c) + \\ & - \sum_{h=a^k+1}^{b^k-1} p_{\pi_i(h)} - L(\pi_i(b^k), c). \end{aligned} \tag{6.28}$$

Similarly, for moves from $\mathcal{T}^{subm}(\Theta)$ which move the last operation $\pi_i(b^k)$ of the block B^k to position $\eta_j(a^k)$ or $\rho_j(a^k)$ in the permutation π_j we introduce the notions

$$\Delta^{b^k}_{\mathrm{y}(b^k)} = \max\{L_1^{\mathrm{y}(b^k)}, L_2^{\mathrm{y}(b^k)}, L_3^{\mathrm{y}(b^k)}, L_4^{\mathrm{y}(b^k)}\}, \;\; \mathrm{y}(b^k) \in \{\eta_j(b^k),\ \varrho_j(b^k)\}, \tag{6.29}$$

where

$$L_1^{\mathrm{y}(b^k)} = L(\pi_i(b^k-1),c) - p_{\pi(b^k-1)} - L(\pi_i(b^k),c), \tag{6.30}$$

$$L_2^{\mathrm{y}(b^k)} = L(\pi_i(b^k+1),c) - L(\pi_i(b^k),c), \tag{6.31}$$

$$L_3^{\mathrm{y}(b^k)} = L(s,\pi_j(1)) + \sum_{h=2}^{\varrho_j-1} p_{\pi_j(h)} + p_{\pi_i(b^k)} + L(\pi_j(\varrho_j),c) + \\ - L(s,\pi_i(a^k)) - \sum_{h=a^k+1}^{b^k-1} p_{\pi_i(h)} - L(\pi_i(b^k),c), \tag{6.32}$$

$$L_4^{\mathrm{y}(b^k)} = L(s,\beta(1)) + \sum_{h=2}^{\mathrm{y}(b^k)-1} p_{\pi_j(h)} - L(s,\pi_i(a^k)) + \\ - \sum_{h=a^k+1}^{b^k-1} p_{\pi_i(h)}. \tag{6.33}$$

Now, we will prove theorems which allow us to estimate a cost function value for a solution generated from Θ by shifting the first operation $\pi_i(a)$ from the block B^k (executing a *t-move*) to position $\eta_j(a)$ or $\rho_j(a)$ on machine M_j.

Theorem 6.6. *If the solution $\Theta' = (\mathcal{Q}', \pi')$ is generated from the $\Theta = (\mathcal{Q}, \pi)$ by executing the move $t_j^i(a^k, \mathrm{x}(a^k)) \in \mathcal{T}^{subm}(\Theta)$, $\mathrm{x}(a^k) \in \{\eta_j(a^k), \rho_j(a^k)\}$ then*

$$L'(s,c) \geq L(s,c) + \Delta_{\mathrm{x}(a^k)}^{a^k}. \tag{6.34}$$

Proof. Let $\pi_i = (\pi_i(1), \pi_i(2), \ldots, \pi_i(\varrho_i))$ and $\pi_j = (\pi_j(1), \pi_j(2), \ldots, \pi_j(\varrho_j))$ be permutation of operations executing on machines M_i and M_j, respectively. The operations sequence $B^k = (\pi_i(a^k), \pi_i(a^k+1), \ldots, \pi_i(b^k))$ $(1 \leq a^k \leq b^k \leq \varrho_i)$ is a block on the machine M_i, i.e., B^k is a subsequence π_i.

For the notion simplification we assume that $a = a^k$, $b = b^k$ and $\alpha = \pi_i = (\alpha(1), \ldots, \alpha(a), \ldots, \alpha(b), \ldots, \alpha(u))$, $\beta = \pi_j = (\beta(1), \ldots, \beta(w))$ where $u = \varrho_i$, $w = \varrho_j$.

The graph $G(\Theta)$ is acyclic, so there exists a critical path $C(s,c)$ with the length $L(s,c)$. For each vertex $v \in \mathcal{O}$ there is

$$C(s,c) = (C(s,v), C(v,c)) \tag{6.35}$$

and

$$L(s,c) = L(s,v) + C(v,c) - p_v. \tag{6.36}$$

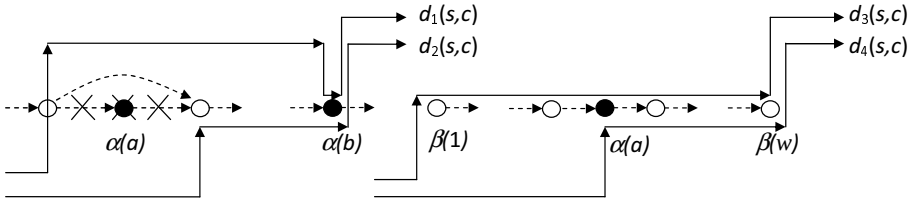


Fig. 6.4. Paths in the graph $G(\mathcal{Q}', \pi')$ generated from $G(\mathcal{Q}, \pi)$ by a move $t_j^i(a, \mathrm{x}(a))$.

Vertices of the critical path can be partitioned into subsequences

$$C(s,c) = (C(s,\alpha(a)), C(\alpha(a),\alpha(b)), C(\alpha(b),c))) \tag{6.37}$$

We consider the following paths: $d_1(s,c)$, $d_2(s,c)$, $d_3(s,c)$ and $d_4(s,c)$ from the vertex s to c in the graph $G(\Theta')$. They are shown as arrows in Figure 6.4.

$$d_1(s,c) = (C'(s,\alpha(a-1)), C'(\alpha(a-1),\alpha(b)), C'(\alpha(b),c)), \tag{6.38}$$
$$d_2(s,c) = (C'(s,\alpha(a+1)), C'(\alpha(a+1),\alpha(b)), C'(\alpha(b),c)), \tag{6.39}$$
$$d_3(s,c) = (C'(s,\beta(1), C'(\beta(1),\beta(w)), C'(\beta(w),c)), \tag{6.40}$$
$$d_4(s,c) = (C'(s,\alpha(a)), C'(\alpha(a) = \beta(\mathrm{x}(a)),\beta(w)), C'(\beta(w),c). \tag{6.41}$$

Taking advantage of the fact that

$$C'(s,\alpha(a-1)) = C(s,\alpha(a-1)) \text{ and } C'(\alpha(b),c)) = C(\alpha(b),c)) \tag{6.42}$$

we obtain

$$d_1(s,c) = (C(s,\alpha(a-1)), C'(\alpha(a-1),\alpha(b)), C(\alpha(b),c)) \tag{6.43}$$

and similarly

$$d_2(s,c) = (C(s,\alpha(a+1)), C'(\alpha(a+1),\alpha(b)), C(\alpha(b),c)), \tag{6.44}$$
$$d_3(s,c) = (C(s,\beta(1), C'(\beta(1),\beta(w)), C(\beta(w),c)), \tag{6.45}$$
$$d_4(s,c) = (C(s,\alpha(a)), C'(\alpha(a) = \beta(\mathrm{x}(a)),\beta(w)), C(\beta(w),c). \tag{6.46}$$

Therefore the length of these paths (in the graph $G(\Theta')$) can be defined by length of some paths in the graph $G(\Theta)$. These are as follows

$$l^1(s,c) = L(s,\alpha(a-1)) + \sum_{h=a+1}^{b-1} p_{\alpha(h)} + L(\alpha(b),c), \tag{6.47}$$

$$l^2(s,c) = L(s,\alpha(a+1)) + \sum_{h=a+2}^{b-1} p_{\alpha(h)} + L(\alpha(b),c), \tag{6.48}$$

$$l^3(s,c) = L(s,\beta(1)) + \sum_{h=2}^{w-1} p_{\beta(h)} + p_{\alpha(a)} + L(\beta(w),c), \tag{6.49}$$

$$l^4(s,c) = L(s,\alpha(a) + \sum_{h=\mathrm{x}(a)+1}^{w-1} p_{\beta(h)} + L(\beta(w),c). \tag{6.50}$$

As the graph $G(\Theta')$ is acyclic, there exists a critical path $C'(s,c)$ whose length must not be shorter than the length of any other paths from vertex s to c in $G(\Theta')$. Therefore

$$L'(s,c) \geq l^1(s,c), \tag{6.51}$$

$$L'(s,c) \geq l^2(s,c), \tag{6.52}$$

$$L'(s,c) \geq l^3(s,c), \tag{6.53}$$

$$L'(s,c) \geq l^4(s,c). \tag{6.54}$$

From this and using (6.37) we obtain

$$\begin{aligned}
L'(s,c) \geq\ & \max\{l^1(s,c),\ l^2(s,c),\ l^3(s,c),\ l^4(s,c)\} = \max\{L(s,\alpha(a-1)) \\
& + \sum_{h=a+1}^{b-1} p_{\alpha(h)} + L(\alpha(b),c),\ L(s,\alpha(a+1)) + \sum_{h=a+2}^{b-1} p_{\alpha(h)} \\
& + L(\alpha(b),c)),\ L(s,\beta(1)) + \sum_{h=2}^{w-1} p_{\alpha(h)} + p_{\alpha(a)} + L(\beta(w),c), \\
& L(s,\alpha(a) + \sum_{h=\mathrm{x}(a)}^{w-1} p_{\alpha(h)} + L(\beta(w),c)\} \\
=\ & \max\{L(s,c) + L(s,\alpha(a-1)) - L(s,\alpha(a)), L(s,c) + L(s,\alpha(a+1)) + \\
& - L(s,\alpha(a)) - p_{\alpha(a+1)}, L(s,c) + L(s,\beta(1)) + \sum_{h=2}^{w-1} p_{\beta(h)} + p_{\alpha(a)} + \\
& + L(\beta(w),c) - L(s,\pi_i(a^k)) - \sum_{h=a^k+1}^{b^k-1} p_{\pi_i(h)} - L(\pi_i(b^k),c), \\
& L(s,c) + \sum_{h=\mathrm{x}(a)+1}^{w-1} p_{\beta(h)} + L(\beta(w),c) - \sum_{h=a+1}^{b-1} p_{\alpha(h)} - L(\alpha(b),c)\} \\
=\ & L(s,c) + \max\{L_1^{\mathrm{x}(a^k)},\ L_2^{\mathrm{x}(a^k)},\ L_3^{\mathrm{x}(a^k)},\ L_4^{\mathrm{x}(a^k)}\} \\
=\ & L(s,c) + \Delta^a_{\mathrm{x}(a)},
\end{aligned} \tag{6.55}$$

which completes the proof of the theorem. ■

The next theorem is related with moving the last operation $\pi(b^k)$ from the block B^k to machine M_j.

Theorem 6.7. *If the solution $\Theta' = (\mathcal{Q}', \pi')$ is generated from the $\Theta = (\mathcal{Q}, \pi)$ by executing the move $t_j^i(b^k, \mathrm{y}(b^k)) \in \mathcal{T}^{subm}$, $\mathrm{y}(b^k) \in \{\eta_j(b^k),\ \rho_j(b^k)\}$ then*

$$L'(s,c) \geq L(s,c) + \Delta^{b^k}_{\mathrm{y}(b^k)}. \tag{6.56}$$

Proof. Similarly, as in the proof of Theorem 6.6 we assume that $\pi_i = (\pi_i(1), \pi_i(2), \ldots, \pi_i(\varrho_i))$ and $\pi_j = (\pi_j(1), \pi_j(2), \ldots, \pi_j(\varrho_j))$ are permutations of operations executed on machine M_i and M_j, and $B^k = (\pi_i(a^k),\ \pi_i(a^k+1), \ldots, \pi_i(b^k))$ $(1 \leq a^k \leq b^k \leq \varrho_i)$ is the block on the machine M_i.

Further, for simplification purposes we assume that $\alpha = \pi_i = (\alpha(1),\ \alpha(2),\ \ldots,\ \alpha(u))$, $\beta = \pi_j = (\beta(1), \ldots, \beta(w))$ where $u = \varrho_i$, $w = \varrho_j$, and the block $B^k = (\pi_i(a), \pi_i(a+1), \ldots, \pi_i(b))$.

We consider the following paths

$$d_1(s,c) = (C'(s,\alpha(a)), C'(\alpha(a), \alpha(b-)), C'(\alpha(b-1), c)), \tag{6.57}$$
$$d_2(s,c) = (C'(s,\alpha(a)), C'(\alpha(a), \alpha(b+1)), C'(\alpha(b+1), c)), \tag{6.58}$$
$$d_3(s,c) = (C'(s,\beta(1), C'(\beta(1), \beta(w)), C'(\beta(w), c)), \tag{6.59}$$
$$d_4(s,c) = (C'(s,\beta(1)), C'(\beta(1), \beta(\mathrm{y}(b)) = \alpha(b)), C'(\beta(\mathrm{y}(b)), c) \tag{6.60}$$

in the acyclic graph $G(\Theta')$ generated by the move $t_j^i(b^k, \mathrm{y}(b^k))$. The lengths of these paths are as follows

$$l^1(s,c) = L(s,\alpha(a)) + \sum_{h=a+1}^{b-2} p_{\alpha(h)} + L(\alpha(b-1), c), \tag{6.61}$$

$$l^2(s,c) = L(s,\alpha(a)) + \sum_{h=a+1}^{b-1} p_{\alpha(h)} + L(\alpha(b+), c), \tag{6.62}$$

$$l^3(s,c) = L(s,\beta(1)) + \sum_{h=2}^{w-1} p_{\beta(h)} + L(\beta(w), c), \tag{6.63}$$

$$l^4(s,c) = L(s,\beta(1) + \sum_{h=2}^{\mathrm{y}(b)-1} p_{\beta(h)} + L(\beta(b), c). \tag{6.64}$$

Because the critical path $C'(s,c)$ in the graph $G(\Theta')$ is the longest one from vertex s to c, its length equals

$$L'(s,c) \geq \max\{l^1(s,c),\ l^2(s,c),\ l^3(s,c),\ l^4(s,c)\}. \tag{6.65}$$

After having executed transformations similar to those in the proof of Theorem 6.6 we obtain

$$L'(s,c) \geq L(s,c) + \Delta^{b}_{\mathrm{y}(b)}, \tag{6.66}$$

which completes the proof of the theorem. ∎

Remark 6.2. *Values l^k, $k = 1,2,3,4$, can be determined sequentially in time $O(n) = O(o)$. These calculations can be done in parallel in time $O(\log n) = O(\log o)$ using $O\left(\frac{n}{\log n}\right) = O\left(\frac{o}{\log o}\right)$-processor CREW PRAM.*

Moving the operation $\pi(a^k)$ to the position $\eta_j(a^k)$ or $\rho_j(a^k)$ the graph is generated in which the lower bound of the length of the critical path from vertex s to c is the value of the expression $L(s,c) + \Delta^{a^k}_{\eta_j(a^k)}$ (or $L(s,c) + \Delta^{a^k}_{\rho_j(a^k)}$). That is why the expression $\Delta^{a^k}_{\mathrm{x}(a^k)}$, $\mathrm{x}(a^k) \in \{\eta_j(a^k),\ \rho_j(a^k)\}$ can be used to determine the operation (i.e., an element from the neighborhood) that will be moved.

Similarly, $L(s,c)+\Delta^{b^k}_{\eta_j(b^k)}$ (or $L(s,c)+\Delta^{b^k}_{\rho_j(b^k)}$) is a lower bound of the critical path length in the graph generated by moving an operation $\pi(b^k)$ to positions $\eta_j(b^k)$ or $\rho_j(b^k)$ and the expression $\Delta^{b^k}_{\mathrm{y}(b^k)}$, $\mathrm{y}(b^k) \in \{\eta_j(b^k),\ \rho_j(b^k)\}$ can be employed to select an element from the neighborhood.

We choose the operation $\pi(v) \in O$ such that

$$\Delta^{v}_{\chi(v)} = \min_{1\leq k\leq r} \min\{\Delta^{z}_{\mu(z)} :\ z \in \{a^k,\ b^k\},\ \mu(z) \in \{\eta_j(z),\ \rho_j(z)\}\} \tag{6.67}$$

The minimal value $\Delta^{v}_{\chi(v)}$ is connected with the best *t-move* which consists in moving the first or the last operation from some block to another machine. From Theorems 6.6 and 6.7 it follows that if $\Delta^{v}_{\chi(v)} > 0$, then the critical path length $L'(s,c) > L(s,c)$ in the generated graph $G(\Theta')$.

Summing up, for the solution $\Theta = (\mathcal{Q}, \pi)$ (fixed machine workload $\mathcal{Q}$) we propose the following method of the new assignment $\mathcal{Q}'$ determination. In the graph $G(\Theta)$ we determine the critical path $C(s,c)$ (if there are more than one, we choose any of them) and we calculate its length $L(s,c) = C_{\max}(\Theta)$. Next, we determine the partition of the path into blocks $\mathcal{B} = (B^1, B^2, \ldots, B^r)$ and in accordance with (6.21) the set of moves $\mathcal{T}^{subm}(\Theta)$. Using (6.67) we determine $\Delta^{v}_{\chi(v)}$ and choose the best *t-move* $t^i_j(v, \chi(v))$. This move generates a solution (the new machine workload) from the neighborhood $\mathcal{N}(\Theta)$ with the lowest value of the lower bound of the cost function.

6.2.3. Machine workload rearrangement

The algorithm proposed here searches the neighborhood generated by *t-moves* transferring the first and the last operations of each block from the critical path

to another machine. The computational complexity of the `NewPar` algorithm is $O(o^3)$ because of the complexity of creating a *t-move* neighborhood (Step 3). Determination of the longest paths (Step 1) can be done using Floyd's algorithm in time $O(o^3)$, or applying the recursive method based on topological sorting in time $O(o)$, maintaining the complexity $O(o^3)$ of the whole sequential algorithm. An outline of the algorithm is presented in Figures 6.5 and 6.6.

6.2.4. Parallel determination of the workload

Now, we will show a parallel version of the `NewPar` algorithm designed to be executed on $O(o^2)$-processor CREW PRAM in time $O(o)$. An outline of the algorithm is presented in Figures 6.7 and 6.8.

Step 1 consists in: (1) sequential determination of the graph $G(\Theta) = (\mathcal{V}, \mathcal{R} \cup \mathcal{E}(\Theta))$ connected with the solution Θ and (2) parallel determination of the longest paths for all pairs of vertices in this graph, which can be done using parallel Floyd's algorithm. Because the graph has at most o vertices, parallel all-pairs the longest paths determination algorithm works in time $\Theta(o)$ using o^2-processor CREW PRAM (see [124]). It is possible to determine the longest paths faster (using a greater number of processors), but in this case this is useless because Step 3 (neighborhood determination) has linear computational complexity $O(o)$.

Step 2 (block determination) can be executed in constant time $O(o)$ using as many processors as there are vertices on the longest path – at most o. Let us assign each processor to one vertex v lying on the critical path. It is enough to check by each processor if the machine number assigned to its vertex $\lambda(v)$ is the same as the machine number $\lambda(u)$ assigned to the next vertex u lying on the critical path. If it is not the same it means that the next block begins in u (see Section 3.6.2). Such a comparison can be made in time $O(1)$ using $O(o)$-processor CREW PRAM.

Step 3 (neighborhood determination) consists of two loops: external and internal one, which can be executed independently in parallel. Inside them each processor needs to determine feasible positions $\eta_j(a^k)$, $\rho_j(a^k)$, $\eta_j(b^k)$ and $\rho_j(b^k)$, which can be done in linear time $O(o)$. Afterwards, values $\Delta^{a^k}_{\eta_j(a^k)}$, $\Delta^{a^k}_{\rho_j(a^k)}$, $\Delta^{b^k}_{\eta_j(b^k)}$ and $\Delta^{b^k}_{\rho_j(b^k)}$ have to be calculated, which also needs time $O(o)$ (the sum of at most o elements has to be determined; see Theorem 6.7). The entire Step 3 requires $O(o^2)$ processors (to execute two loops in parallel) to be made in time $O(o)$.

Step 4 (the best *t-move* move determination) consists in choosing one move from $O(4r \cdot m_{\max})$ moves, where $m_{\max} = \max_{1 \le i \le q} m_i$ is the maximal number of machines in a nest. Because $O(4r \cdot m_{\max}) = O(rm) = O(o^2)$ therefore we need to use $O(o^2)$ processors to determine the minimal element of $O(o^2)$ elements in time $O(\log o^2) = O(2 \log o) = O(\log o)$. In fact, it is enough to use less processors,

Algorithm 3. NewPar

Input: $\Theta = (\mathcal{Q}, \pi)$ - a feasible solution of the FJSP;

Output: $\Theta' = (\mathcal{Q}', \pi')$ - a feasible solution generated by the *t-move*;

Step 1: {*Graph creation*}

Determine a graph with weighted vertices
$G(\Theta) = (\mathcal{V}, \mathcal{R} \cup \mathcal{E}(\Theta))$ connected with the solution Θ;
Determine the longest paths lengths between vertices
of the graph $G(\Theta)$;

Step 2: {*Blocks determination*}

Determine the critical path in $G(\Theta)$
(i.e., vertices sequence $C(s, c)$);
Determine blocks sequence $\mathcal{B} = (B^1, B^2, \ldots, B^r)$
of the critical path $C(s, c)$);

Step 3: {*Neighborhood determination*}

for k:=1 **to** r **do** {*consecutive blocks consideration*}

if (block operations $B^k = (\pi(a^k), \pi(a^k + 1), \ldots, \pi(b^k))$
are executed on the machine M_v from the nest $\mathcal{M}^u$)

then

for $i := t_{u-1} + 1$ **to** $t_{u-1} + m_u$ **do**
{*machines of the nest* $\mathcal{M}^u$}
if $i \neq v$ **then**
begin
determine feasible positions $\eta_j(a^k)$ and $\rho_j(a^k)$
for the operation a^k on the machine M_i and calculate
the expression value $\Delta^{a^k}_{\eta_j(a^k)}$ and $\Delta^{a^k}_{\rho_j(a^k)}$;
determine feasible positions $\eta_j(b^k)$ and $\rho_j(b^k)$
for the operation b^k on the machine M_i and calculate
the expression value $\Delta^{b^k}_{\eta_j(b^k)}$ and $\Delta^{b^k}_{\rho_j(b^k)}$;
end;

Step 4: {*The best move determination*}

Determine the value
$\Delta^v_{\chi(v)} = \min_{1 \leq k \leq r} \min\{\Delta^z_{\mu(z)} : z \in \{a^k, b^k\},$
$\mu(z) \in \{\eta_j(z), \rho_j(z)\}\}$
corresponding to the best *t-move* $t^v_{\chi(v)}$ consisting in
moving the first or the last operation, respectively,
from some block to another machine from the same nest;

Fig. 6.5. Outline of the sequential NewPar algorithm, Part 1.

```
  Step 5: {The new assignment determination}
      Determine the new machine workload Q'
      corresponding to the solution Θ' generated by the t-move t^v_χ(v)
      (determined by (1)–(5));
end.
```

Fig. 6.6. Outline of the sequential NewPar algorithm, Part 2.

```
 Algorithm 4.  ParallelNewPar
Input: Θ = (Q, π) - a feasible solution of the FJSP;
Output: Θ' = (Q', π') - a feasible solution generated by the t-move;
  Step 1: {Graph creation}
      if proc_id = 1 then
        Determine a graph with weighted vertices
            G(Θ) = (V, R ∪ E(Θ))
        connected with the solution Θ;
      parfor proc_id = 1..o^2 do
        Parallel determine the longest paths lengths between vertices
        of the graph G(Θ);
      end parfor;
  Step 2: {Blocks determination}
      if proc_id = 1 then
        Determine the critical path in G(Θ)
            (i.e., vertices sequence C(s, c));
      parfor proc_id = 1..o do
        Parallel determine blocks sequence B = (B^1, B^2, ..., B^r)
        of the critical path C(s, c));
      end parfor;
  Step 3: {Neighborhood determination}
      parfor k:=1 to r do      {consecutive blocks consideration}
        if (block operations B^k = (π(a^k), π(a^k + 1), ..., π(b^k))
            are executed on the machine M_v from the nest M^u)
        then                                {machines of the nest M^u}
```

Fig. 6.7. Outline of the ParallelNewPar algorithm, Part 1.

namely $O(\frac{o^2}{2\log o})$ instead of $O(o^2)$ to maintain the same computational complexity $O(\log o)$; though it is not necessary because the other elements of the whole algorithm have linear complexity $O(o)$.

```
            parfor i := t_{u-1} + 1 to t_{u-1} + m_u do
                if i ≠ v then
                begin
                  determine feasible positions η_j(a^k) and ρ_j(a^k)
                  for the operation a^k on the machine M_i and
                  calculate expressions value Δ^{a^k}_{η_j(a^k)} and Δ^{a^k}_{ρ_j(a^k)};
                  determine feasible positions η_j(b^k) and ρ_j(b^k)
                  for the operation b^k on the machine M_i and
                  calculate expressions value Δ^{b^k}_{η_j(b^k)} and Δ^{b^k}_{ρ_j(b^k)};
                end;
  Step 4: {The best move determination}
      parfor proc_id = 1..o^2 do
        Parallel Determine the minimal value
        Δ^v_{χ(v)} = min_{1≤k≤r} min{Δ^z_{μ(z)} : z ∈ {a^k, b^k},
            μ(z) ∈ {η_j(z), ρ_j(z)}}
        connected with the best t-move t^v_{χ(v)} consisting in
        moving the first or the last operation, respectively,
        from some block to another machine from the same nest;
  Step 5: {The new assignment determination}
      if proc_id = 1 then
        Determine the new machine workload Q'
        connected with the solution Θ' generated by the t-move t^v_{χ(v)}
end.
```

Fig. 6.8. Outline of the ParallelNewPar algorithm, Part 2.

Step 5 consisting in executing the *t-move* selected in the previous step can be made by the single (master) processor in constant time $O(1)$. Thus computational complexity of the whole parallel algorithm is $O(o)$. The algorithm needs to be executed on $O(o^2)$-processor CREW PRAM and it is cost-optimal with the cost $O(o^3)$. A general scheme of the `ParallelNewPar` algorithm execution on GPU for the CUDA programming environment is shown in Figure 6.9 as the case of heterogeneous programming model (i.e., with using both CPU and GPUs).

6.3. Remarks and conclusions

A single-walk parallel approach to the flexible job shop scheduling has been presented in this chapter. We show the new integrated approach to the neighborhood structure design and to its searching methodology from the point of view of the

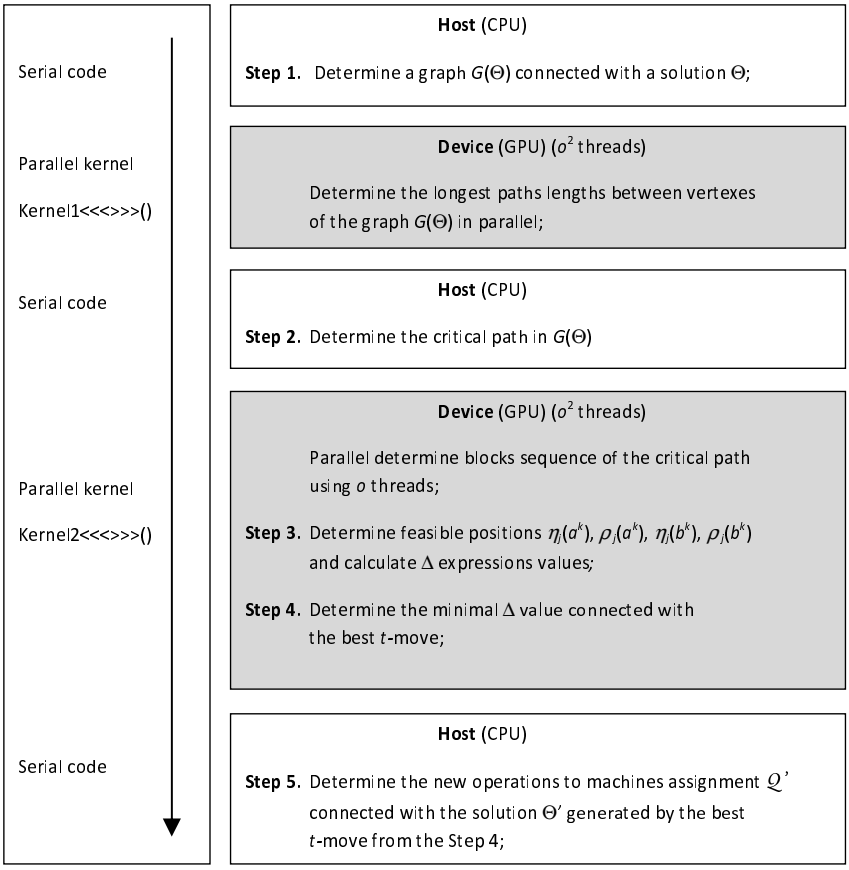


Fig. 6.9. The general scheme of the `ParallelNewPar` algorithm execution on the host (CPU) and the computational device (GPU) for the CUDA environment.

efficient multi-thread computing environment usage. The methodology is illustrated by a special case of hybrid job shop scheduling problem. We propose the new machine workload rearrangement technique used to concurrent generation of the operations on machine schedules. Additionally, critical and sub-critical paths lengths estimation allows us to shorten computations time by using lower bound of the goal function instead of its exact value during neighborhood searching.

A theoretical analysis based on PRAM model of parallel computing was also made. We proposed a cost-optimal method of the neighborhood generation parallelization for the CREW PRAM parallel computing model. The workload parallel determination algorithm decreases the computations time from $O(o^3)$ (of the sequential approach) to $O(o)$ time, using $O(o^2)$ processors. Applying PRAM computing model makes it possible to convert the proposed methods to GPU environment easily.

Chapter 7

Theoretical properties of a single-walk parallel GA

This chapter aims at presenting theoretical properties which can be used to approximate the theoretical speedup of parallel genetic algorithms. The most frequent parallelization method employed in a genetic algorithm implements a master-slave model by distributing the most computationally exhausting elements of the algorithm (usually evaluation of the fitness function, i.e., cost function calculation) among a number of processors (slaves). This master-slave parallelization is regarded as easy in programming, which makes it popular with practitioners. Additionally, if the master processor stores the population (and slave processors are used only as computational units for fitness function evaluation of individuals), it explores the solution space in exactly the same manner as sequential genetic algorithm. We can thus say that we analyze the single-walk parallel genetic algorithm.

We present two approaches in this chapter. The first one, in Section 7.1, follows from Cantú-Paz [72] and we discuss it briefly. The second one, described in Section 7.2, constitutes a new idea of the broadcasting time approximation for the master-slave parallel genetic algorithm.

7.1. Sequential broadcasting

A parallel genetic algorithm based on the master-slave model consists of two major modules: (1) communication module, performed chiefly by the master processor which broadcasts a part of population among slave processors, and (2) computing modules, executed both on master and slaves, in which evaluation of the fitness function is performed. We use notation taken from Cantú-Paz [72]. Let T_c be the time used to send a portion of data between two processors, and let T_f denote

the time required to evaluate one individual. Each of the processors, i.e., both master and slaves, evaluates a fraction of the population in time $\frac{nT_f}{p}$, where p is the number of processors and n is the population size. Next we assume in this section that the master broadcasts the data to slave processors sequentially, as Figure 7.1 shows. We omit the time consumed by genetic operators as well as by the mutation (it is usually much shorter than the time of the fitness function evaluation). We also assume that the part of data assigned to each processor (i.e., the number of individuals evaluated) is the same both for each slave processor, and for the master processor.

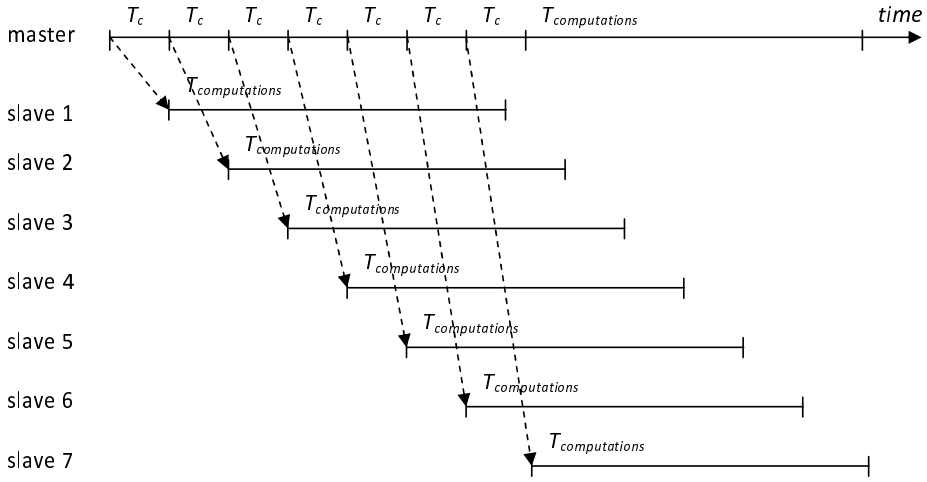


Fig. 7.1. Sequential broadcasting in the master-slave parallel genetic algorithm.

For a sequential model of broadcasting, the parallel running time is given by the equation

$$T_p = pT_c + \frac{nT_f}{p}. \tag{7.1}$$

Let us check for which p the T_p is minimal. We denote this p by p_1^*. Calculating $\frac{\partial T_p}{\partial p} = 0$ we get

$$T_c - \frac{nT_f}{p^2} = 0, \tag{7.2}$$

$$p = p_1^* = \sqrt{\frac{nT_f}{T_c}}, \tag{7.3}$$

which provides us with an optimal number of processors p_1^* minimizing the value of the parallel running time T_p. Calculating the maximum value of the theoretical

speedup S_p we obtain

$$S_p = \frac{T_s}{T_p} = \frac{nT_f}{pT_c + \frac{nT_f}{p}}. \tag{7.4}$$

Substituting the optimal number of processors p_1^* we have

$$S_{p_1^*} = \frac{nT_f}{p_1^* T_c + \frac{nT_f}{p_1^*}} = \frac{nT_f}{\sqrt{\frac{nT_f}{T_c}}T_c + \frac{nT_f}{\sqrt{\frac{nT_f}{T_c}}}} =$$

$$= \frac{nT_f}{\sqrt{nT_f T_c} + \sqrt{\frac{(nT_f)^2}{\frac{nT_f}{T_c}}}} = \frac{\sqrt{(nT_f)^2}}{2\sqrt{nT_f T_c}} = \frac{1}{2}\sqrt{\frac{nT_f}{T_c}} = \frac{1}{2}p_1^*, \tag{7.5}$$

which gives us a maximal possible speedup for this model of the single-walk master-slave parallel genetic algorithm.

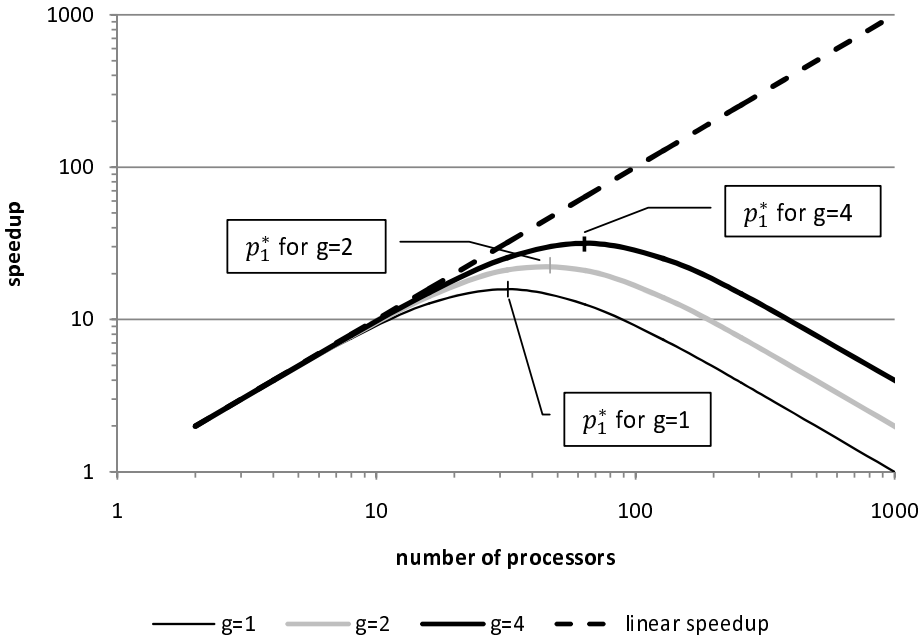


Fig. 7.2. Theoretical speedups for the *sequential broadcasting* in the master-slave parallel genetic algorithm.

Figure 7.2 shows possible theoretical speedups for a given ratio $g = \frac{T_f}{T_c}$. The speedup is plotted for $g = 1, 2, 4$ showing that linearity of the speedup increases with parameter g. In practice, T_f is much greater than T_c. In such a situation, the

parallel algorithm can achieve near-linear speedup for the number of processors from the range $[1, p_1^*]$. For the number of processors greater than p_1^* speedup quickly decreases.

7.2. Tree-based broadcasting

Now, we propose a faster model of communication for the master-slave parallel genetic algorithm. The broadcasting process is based on tree communication scheme, which offers the possibility of obtaining logarithmic complexity of the broadcasting process. This broadcasting scheme needs cooperation of all processors during the communication process. A scheme of the master-slave parallel genetic algorithm based on this communication model is shown in Figure 7.3.

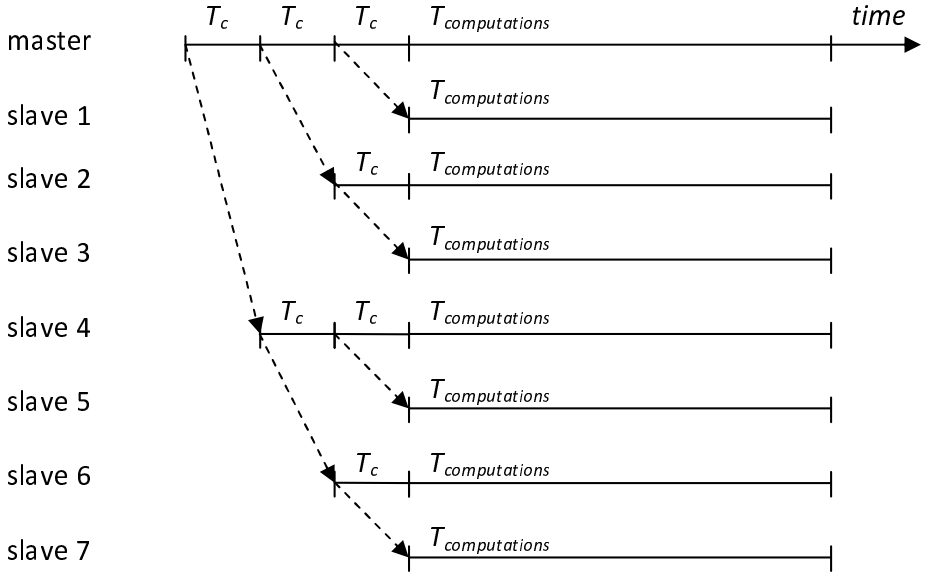


Fig. 7.3. Tree-based broadcasting in the master-slave parallel genetic algorithm.

For the tree-based communication model the parallel running time T_p is estimated by

$$T_p = T_c \log_2 p + \frac{nT_f}{p}. \tag{7.6}$$

In the case of using more processors, the parallel computing time $\left(\frac{nT_f}{p}\right)$ decreases, whereas the time of communication $(T_c \log p)$ increases. We are looking for such a number of processors p (let us call it p_2^*) for which T_p is minimal. Calculating

$\frac{\partial T_p}{\partial p} = 0$ we obtain

$$\frac{T_c}{p \ln 2} - \frac{nT_f}{p^2} = 0 \tag{7.7}$$

and then

$$p = p_2^* = \frac{nT_f \ln 2}{T_c}, \tag{7.8}$$

which provides us with an optimal number of processors p_2^* which minimizes the value of the parallel running time T_p for this model of broadcasting. Calculating the maximum value of the theoretical speedup S_p we have

$$S_p = \frac{T_s}{T_p} = \frac{nT_f}{T_c \log_2 p + \frac{nT_f}{p}}. \tag{7.9}$$

Substituting the optimal number of processors p_2^* we obtain

$$S_{p_2^*} = \frac{nT_f}{T_c \log_2 p_2^* + \frac{nT_f}{p_2^*}} = \frac{nT_f}{\frac{T_c}{\ln 2} \ln \frac{nT_f \ln 2}{T_c} + \frac{nT_f}{\frac{nT_f \ln 2}{T_c}}} =$$

$$= \frac{nT_f \ln 2}{T_c \left(1 + \ln \frac{nT_f \ln 2}{T_c}\right)} = \frac{p_2^*}{1 + \ln p_2^*}. \tag{7.10}$$

This equation provides us with a maximal possible speedup for the tree-based model of broadcasting for the single-walk master-slave parallel genetic algorithm.

Figure 7.4 shows possible theoretical speedups for a given ratio $g = \frac{T_f}{T_c}$, $g = 1, 2, 4$. As with the sequential communication plotted in Figure 7.2, linearity of the speedup increases with an increase of the parameter g. The parallel algorithm achieves the near-linear speedup for the number of processors from the range $[1, p_2^*]$. For the number of processors greater than p_2^* speedup keeps on increasing.

7.3. Remarks and conclusions

In this chapter, we discussed some theoretical properties of a metaheuristic which can be used to solve scheduling optimization problems. The tree-based broadcasting model seems to be more efficient than the sequential broadcasting model from the theoretical point of view. In practice, it is possible to make an additional improvement of the algorithm efficiency by fulfilling the idle time of some processors during the communication phase – if the process is executed in the cycle, one generation of the parallel genetic algorithm after another, we can remove the synchronicity constraint. In such a case master processor can execute its communication phase during a communication phase of the previous generation.

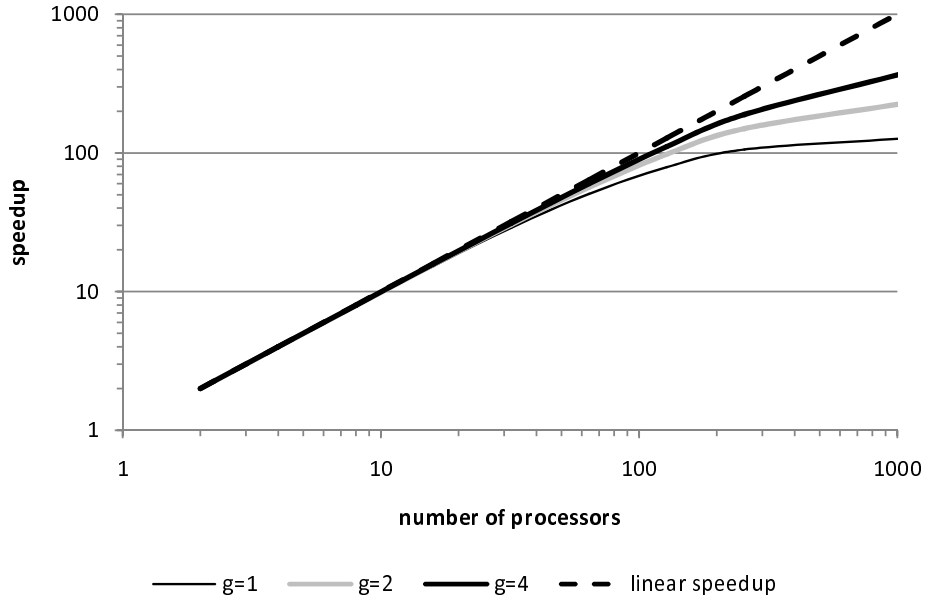


Fig. 7.4. Theoretical speedups for the *tree-based broadcasting* in the master-slave parallel genetic algorithm.

The proposed speedup estimation considered the parallel genetic algorithm based on the master-slave model of parallelism. The analyzed approaches give a theoretical approximation of the optimal number of processors necessary to obtain the highest speedup. Additionally, it is possible to determine theoretical upper bounds for the speedups obtained for the master-slave model of the parallel genetic algorithm with a single population kept by the master processor. The results shown in this chapter can be easily adopted to any other parallel algorithm in which calculation and communication processes appear one after another, i.e., as in distributed single-walk scatter search method and the majority of population-based approaches.

Part III

MULTIPLE-WALK PARALLELIZATION

Chapter 8

Parallel memetic approach

This chapter seeks to present a parallel memetic approach using as an example a single machine total weighted earliness-tardiness ($TWET$) problem described in Section 3.3.4. We additionally assume that the problem considered has *no idle* constraint ($TWET$-*no-idle* problem), which means that the machine works without stops. There are many service systems (especially in reservation systems, electronic commerce, in tasks synchronized directly with the Internet), where each task has to be executed in some fixed range of time. Violating the term is disadvantageous and causes additional penalties. Therefore, it is necessary to establish an optimal sequence of tasks (which minimizes penalties) and their starting times. This amounts to some job scheduling problems with earliness and tardiness. As tasks are usually received in the distributed system (in the web), that is why to solve the problem presented we propose a parallel memetic algorithm based on Lamarck's evolution and the island model of migration in which part of a population is replaced with adequate local minima. The property of partitioning a permutation into subsequences (blocks) was used in an algorithm of determining local minima. This method decreases the size of a neighborhood to about 50% (in a local optimization algorithm), improving a solution's values and significantly speeding up computations.

8.1. Introduction

Implementations of algorithms which are based on multithread multiple-walk searching of the solution space are usually coarse-grained application, i.e., they require sparse communication and synchronization. Algorithms of this type are easy to apply in distributed calculation systems, as clusters which express beneficial efficiency-to-price ratio. Apart from speeding up the calculations, it is possible to improve the quality of results obtained. Search processes can be either independent or cooperative.

8.1.1. Independent searching threads

In this category we can distinguish two basic approaches:

- Researching of the solution space by using multiple trajectories, which begin from different starting solutions (or different starting populations in the case of using population-based approaches). Searching threads can use either the same or different strategies, i.e., the same or different local search algorithms, the same or different parameters (tabu list length, population size, etc.). Trajectories can cross each other in one or more places of the neighborhood graph.

- Parallel researching of subgraphs of a neighborhood graph obtained by decomposing the problem into a few subproblems (for example, fixing some variables). Subgraphs of the neighborhood graph are searched concurrently without crossing search trajectories. We obtain the partitioning of the neighborhood graph into disjoint subgraphs.

The first parallel implementation of the tabu search method based on multiple-walk searching of the solution space was proposed by Taillard for the quadratic assignment problem (QAP) [244] and the job shop problem [245]. The multiple-walk parallelization strategy based on independent searching threads is easy in implementation and one can obtain good values of the speedup under condition of proper decomposition of the solution space into searching threads (and their trajectories). If the decomposition is done improperly, a parallel algorithm can multiply search through the same regions of the solution space, i.e., we obtain redundance of searching.

8.1.2. Cooperative searching threads

This model constitutes the most general and promising type of solution space searching strategy by using parallel metaheuristics, however it requires knowledge of solving problem specificity. 'Cooperative' means here the interchange of information – experience of searching history up to now. Specific information, which is characteristic of the problem and the method (e.g. the best solution found so far, elite solutions, the frequency of moves, tabu lists, backtrack-jump list, subpopulations and their sizes, etc.) has to be exchanged or broadcasted.

Information shared by search processes can be stored as global variables kept in the *shared memory* or as records in the local memory of the dedicated central processor which communicates with all other processors providing them with requested data. In a model in which processes cooperate with each other and information gathered when moving along a trajectory is used to improve other trajectories, one can expect not only convergence of such a parallel algorithm,

but also finding at the same time a better solution than the parallel algorithm without communication. In such a case we can say that cooperative concurrent algorithms constitute a new class of algorithms indeed.

The first heuristic algorithm of this type was asynchronous parallel tabu search algorithm proposed by Crainic, Toulouse and Gendreau [85]. Packages such as ASA [141] and ParSA [155] offer ready implementations of parallel simulated annealing algorithms based on cooperative searching threads. The interaction strategy is also very efficient in implementation of parallel genetic algorithms (in the sense of solutions obtained). There are plenty of ready libraries such as PGAPack [13] and POOGAL [68]. The majority of cooperative implementations of parallel genetic algorithm are based on the *migration island model*. Each process has its own subpopulation exchanging from time to time a number of individuals (usually the best – elite) with other processes (Bubak and Sowa [68], Crainic and Toulouse [84]). Bubak and Sowa [68] used the migration island model to implement a parallel genetic algorithm for the traveling salesman problem on HP/Convex Exemplar SPP1600 with 16 processors and on heterogenous clusters: Hewlett-Packard (D-370/2 and 712/60) and IBM (RS6000/520 and RS6000/320). Bożejko [26] proposed a parallel path-relinking metaheuristic based on the parallel scatter search algorithm.

8.2. Memetic algorithm

All operations in a coevolutionary memetic algorithm (selection, crossover, local optimization and succession) are executed locally, on some subsets of the current population called *islands*. It is a strongly decentralized model of an evolutionary algorithm. There are independent evolution processes on each of the islands, and communication takes place sporadically. Exchanging individuals between islands secures diversity of populations and prevents fast imitating of an individual with a local minimum as its goal function. On each island a hybrid algorithm is applied, in which an evolutionary algorithm is used to determine the starting solutions for the local search algorithm. An outline of the standard memetic algorithm is presented in Figure 8.1.

8.3. Parallel memetic algorithm

The parallel algorithms based on the island model divide the population into a few subpopulations. Each of them is assigned to a different processor which performs a sequential memetic algorithm based on its own subpopulation. The crossover involves only individuals within the same population. Occasionally, the processor exchanges individuals through a migration operator. The main

Algorithm 5. Memetic algorithm
Number of iteration k :=0; $P_0 \leftarrow$ initial population;
repeat
$P'_k \leftarrow$Selection(P_k); {*Selection of parents*}
$P''_k \leftarrow$Crossover(P'_k); {*Generating an offspring*}
$P''_k \leftarrow$Mutation(P''_k);
$A \leftarrow$RandomSubSet(P''_k); {*Subpopulation*}
$P''_k \leftarrow P''_k \cup$LocalMinimumSet($A$); {*Local optimization*}
$P_{k+1} \leftarrow$Succession(P_k, P''_k) {*A new population*}
$k := k + 1$;
until some termination condition is satisfied;

Fig. 8.1. Outline of the memetic algorithm.

determinants of this model are: (1) size of the subpopulations, (2) topology of the connection network, (3) number of individuals to be exchanged, (4) frequency of exchanging. The island model of parallel memetic algorithm is characterized by a significant reduction of the communication time, compared to the global model (with distributed computations of the fitness function only). As shared memory is not required, this model is also more flexible.

Below, a parallel memetic algorithm is proposed. The algorithm is based on the island model of parallelism (see Bożejko and Wodecki [50]). We have adapted the MSXF (Multi-Step Crossover Fusion) operator which is used to extend the process of searching for better solutions of the problem. Originally, an MSXF has been described by Reeves and Yamada [215]. Its idea is based on local search, starting from one of the parent solutions, to find a new good solution, where the other parent is used as a reference point. Here we propose to use block properties defined in Section 3.3.4 to make the search process more effective – prevent changes inside the block (which are unprofitable from the point of view of the fitness function). Such a proceeding is consistent with an idea of not making unprofitable changes between memes. In this way we design an MSXF+B (MSXF with blocks) operator.

The neighborhood $\mathcal{N}(\pi)$ of the permutation (individual) π is defined as a set of new permutations that can be obtained from π by exactly one adjacent pairwise exchange operator which exchanges the positions of two adjacent jobs of a problem solution connected with permutation π. The distance measure $d(\pi,\sigma)$ is defined as a number of adjacent pairwise exchanges needed to transform permutation π into permutation σ. Such a measure is known as Kendall's τ measure (see Diaconis [99]). An outline of the procedure is presented in Figure 8.2.

Algorithm 6. Multi-Step Crossover Fusion with Blocks
Let π_1, π_2 be parent solutions. Set $x = q = \pi_1$;
repeat
  Determine blocks in the solution π.
  Determine restricted neighborhood $\mathcal{N}(x)$ according to blocks;
  For each member $y_i \in \mathcal{N}(x)$, calculate $d(y_i, \pi_2)$;
  Sort $y_i \in \mathcal{N}(x)$ in ascending order of $d(y_i, \pi_2)$;
  repeat
    Select y_i from $\mathcal{N}(x)$ with a probability inversely
      proportional to the index i;
      Calculate $F(y_i)$;
    Accept y_i with probability 1 if $F(y_i) \leq F(x)$, and with
      probability
        $$P_T(y_i) = exp((F(x) - F(y_i)) \ / \ T)$$
      otherwise (T is temperature);
    Change the index of y_i from i to n and the indices of
      y_k, $k = i{+}1, \ldots, n$ from k to $k{-}1$;
  until y_i is accepted;
  $x \leftarrow y_i$;
  if $F(x) < F(q)$ **then**
    $q \leftarrow x$;
until *some termination condition is satisfied*;
q is the offspring.

Fig. 8.2. Outline of the Multi-Step Crossover Fusion with Blocks procedure.

In the implementation proposed here the Multi-Step Crossover Fusion with Blocks (MSXF+B) is an inter-island (i.e., inter-subpopulation) crossover operator which constructs a new individual by making use of the best individuals of different islands connected with subpopulations on different processors. The condition of termination consisted in exceeding 100 iterations by the MSXF+B function. An outline of the whole parallel memetic algorithm is presented in Figure 8.3.

The learning phase of the proposed algorithm uses the path-relinking conception which makes it more efficient than a standard descent search used as the learning phase in the sequential memetic algorithm. It provides a good genetic diversification of the population together with a high quality of each individual.

Frequency of communication between processors (MSXF+B operator and migration) is very important for the parallel algorithm performance. This must not take place very often because of the relatively long time of communication be-

```
Algorithm 7. Parallel memetic algorithm
parfor j = 1, 2, ..., p  { p is the processors number}
  i ← 0;
  P_j ← random subpopulation connected with processor j;
  p_j ← number of individuals in j subpopulation;
  repeat
    Selection(P_j, P'_j);
    Crossover(P'_j, P''_j);
    Mutation(P''_j);
    if (k mod R = 0) then
      {every R iteration}
      r := random(1, p);
      MSXF+B(P'_j(1), P_r(1));
      {P_r(1) is the best individual of subpopulation of processor r}
    end if;
    P_j ← P''_j; i ← i + 1;
    if there is no improvement of the average fitness F then
      {Partial restart}
      r ← random(1,p);
      Remove α = 90 percentage of individuals
        in subpopulation P_j.;
      Replenish P_j by random individuals;
    end if;
    if (k mod S = 0) then
      {Migration}
      r ← random(1,p);
      Remove β = 20 percentage of individuals
        in subpopulation P_j;
      Replenish P_j by the best individuals
        from subpopulation P_r taken from processor r;
    end if;
  until Stop_ Condition;
end parfor
```

Fig. 8.3. Outline of the parallel memetic algorithm.

tween processors, compared with the time of communication inside the program of one processor. In this implementation the processor gets new individuals rather rarely, every $R = 20$ (MSXF+B operator) or every $S = 35$ (migration) iterations.

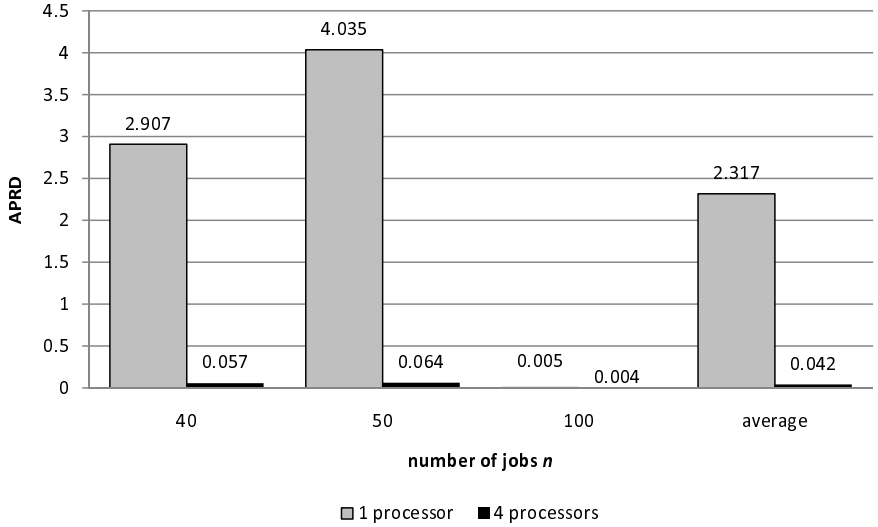


Fig. 8.4. Average percentage relative deviations (APRD) for the sequence and parallel memetic algorithms.

8.4. Computer simulations

The algorithm was implemented in the Ada95 language and ran on the SGI Altix 3700 Bx2 supercomputer installed in WCNS [266] under the Novell SUSE Linux Enterprise Server operating system. Tests were based on 125 instances with 40, 50 and 100 jobs taken from the OR-Library [202]. The results were compared to the best known ones, also taken from [202].

The computational results are presented in Figure 8.4 and in Table A.1 in Appendix A (supplementary tables). The number of iterations is given as a sum of iterations on processors, being permanently set to 800. For example, 4-processor implementations make 200 iterations on each of the 4 processors, so we can obtain comparable costs of computations. As we can observe, the parallel versions of the algorithm achieve much better results of the average and maximal relative deviation from the optimal (or the best known) solutions, working (parallel) in a shorter time. Due to the small cost of communication the speedup parameter of the parallel algorithms is almost linear.

8.5. Remarks and conclusions

The Lamarck evolution theory as well as memetic approach not only significantly extend traditional GA, but offer more effective approach, too. It is well known that the classic GA has a week search intensification phase – genetic operators

as well as a mutation mainly diversify the search process. Additionally, in the memetic approach it is possible to make use of specific problem properties such as the new MSXF+B operator with block properties. Embedding special properties of the problem inside GA is usually difficult. Further benefits are obtained by using an island model with inter-island operator for the parallel asynchronous coevolution.

As we observe MA is also able to improve convergence time compared to GA. Compared to a sequential algorithm, in turn, the parallelization of MA shortens the computing time and improves the quality of solutions obtained. The proposed methodology of memetic algorithms parallelization can be applied to solve concurrently all scheduling problems with block properties, such as flow shop and job shop problems with makespan criterion, single machine scheduling problems, etc., for which a solution is represented as a permutation.

Chapter 9

Parallel population-based approach

In this chapter, we propose parallelization of the new original population-based method using the idea of concurrent local minima searching. It follows the method introduced in papers of Bożejko and Wodecki [27, 52]. The parallelization methodology is illustrated by a single machine scheduling problem with sequence-dependent setup times, defined in Section 3.3.3.

9.1. Population-based metaheuristic

We present a method belonging to the population-based approaches which consists in determining and searching for the local minima. This (heuristic) method is devised on the following observation. If there are the same elements in some positions in several solutions, which are local minima, then these elements can be in the same position in the optimal solution. Because we propose this method for solving problems in which a solution is a permutation that is why in the next part of the chapter we identify these two notions.

The basic idea is to start with an initial population (any subset of the solution space). Next, for each element of the population, a local optimization algorithm is applied (e.g. descending search algorithm or a metaheuristic) to determine a local minimum. In this way we obtain a set of permutations – local minima. If there is an element which is in the same position in several permutations, then it is fixed in this position in the permutation and other positions and elements of permutations are still free. A new population (a set of permutations) is generated by drawing free elements in free positions (because there are fixed elements in fixed positions). After having determined a set of local minima (for the new population) we can increase the number of fixed elements. To prevent the algorithm from

finishing its work after having executed a number of iterations (when all positions are fixed and there is nothing left to draw) in each iteration 'the oldest' fixed elements are set free. The method proposed is especially helpful in solving large-size instances of very difficult discrete optimization problems with irregular goal functions. A similar parallel population-based method was proposed by Bożejko and Wodecki for the routing problem in work [36].

To solve the problem considered we propose a population-based algorithm which examines local minima of the cost function. To determine the local minimum a local search algorithm is used. We apply the following notation:

π^* : suboptimal permutation determined by the algorithm,
η : number of elements in the population,
P^i : population in the iteration i of the algorithm, $P^i = \{\pi_1, \pi_2, \ldots, \pi_\eta\}$,
$LocalOpt(\pi)$: local optimization algorithm to determine local minimum, where π is a starting solution,
LM^i : a set of local minima in iteration i, $LM^i = \{\hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_\eta\}$, $\hat{\pi}_j = LocalOpt(\pi_j)$, $\pi_j \in P^i$, $j = 1, 2, \ldots, \eta$.
FS^i : a set of fixed elements and position in permutations of the population P^i,
$FixSet(LM^i, FS^i)$: a procedure which determines a set of fixed elements and positions in the next iteration of evolutionary algorithm, $FS^{i+1} = FixSet(LM^i, FS^i)$,
$NewPopul(FS^i)$: a procedure which generates a new population in the next iteration of algorithm, $P^{i+1} = NewPopul(FS^i)$.

In any permutation $\pi \in P^i$ positions and elements which belong to the set FS^i (in iteration i) we call *fixed*, whereas other elements and positions we call *free*. The algorithm work begins with creating an initial population P^0 (and it can be created randomly). We set a suboptimal solution π^* as the best element of the population P^0. A new population of iteration $i+1$ (a set P^{i+1}) is generated as follows: for the current population P^{i+1} a set of local minima LM^i is determined (for each element $\pi \in P^i$ executing procedure $LocalOpt(\pi)$). Elements which are in the same positions in local minima are established (procedure $FixSet(LM^i, FS^i)$), and a set of fixed elements and positions FS^{i+1} is generated. Each permutation of a new population P^{i+1} includes fixed elements (in fixed positions) from the set FS^{i+1}. Free elements are randomly drawn in the remaining free positions of the permutation. If permutation $\beta \in LM^i$ exists and $F(\beta) < F(\pi^*)$, then the permutation π^* is set to β. The algorithm finishes its work after having generated a fixed number of generations.

A general structure of the population-based metaheuristic algorithm for the permutation optimization problem is given in Figure 9.1.

Algorithm 8. Population-Based Metaheuristic
Initialization:
a random creation of an initial population $P^0 \leftarrow \{\pi_1, \pi_2, \ldots, \pi_\eta\}$;
$\pi^* \leftarrow$ the best element of the population P^0;
a set of fixed elements $FS^0 \leftarrow \emptyset$; $i \leftarrow 0$;
repeat
Determine a set of local minima $LM^i \leftarrow \{\hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_\eta\}$, where
$\hat{\pi}_j \leftarrow LocalOpt(\pi_j)$, $\pi_j \in P^i$;
for $j \leftarrow 1$ **to** η **do**
if $F(\hat{\pi}_j) < F(\pi^*)$ **then** $\pi^* \leftarrow \hat{\pi}_j$;
Determine a set $FS^{i+1} \leftarrow FixSet(LM^i, FS^i)$ **and**
generate a new population $P^{i+1} \leftarrow NewPop(FS^i)$;
$i \leftarrow i + 1$;
until not Stop Criterion.

Fig. 9.1. General structure of the population-based metaheuristic.

Procedures *LocalOpt*, *FixSet* and *NewPopul* are described in further parts of this chapter. According to the memetic algorithm presented in Chapter 8, the population-based approach proposed here has no genetic operators. The information about the most common elements of local minima is collected and used for creation of new individuals, instead.

9.1.1. A set of fixed elements and positions

A set FS^i (in iteration i) includes quadruples (a, l, α, φ), where a is an element of a set N ($a \in N$), l is a position in permutation ($1 \leq l \leq n$) and α, φ are attributes of a pair (a, l). The parameter α means '*adaptation*' and decides on inserting an element to the set, while φ – '*age*' – decides on deleting it from the set. A $FixSet(LM^i, FS^i)$ procedure is invoked, in which the following operations are executed:

(a) changing the age of each element (φ parameter),
(b) deleting the oldest elements,
(c) inserting the new elements.

There are two functions of acceptance $\Gamma(i)$ and $\Xi(i)$ connected with the inserting and the deleting operations, respectively. Both of them can be determined experimentally.

9.1.2. Element age modification

In each iteration of the algorithm the age of each element which belongs to FS^i is increased by 1, that is

$$\forall(a, l, \alpha, \varphi) \in FS^{i+1}, \tag{9.1}$$

$$FS^{i+1} \leftarrow FS^i \backslash \{(a, l, \alpha, \varphi)\} \cup \{(a, l, \alpha, \varphi + 1)\}. \tag{9.2}$$

The age parameter makes it possible to delete an element from the set FS^i. Each fixed element is free after some number of iterations and can be fixed again in any free position.

9.1.3. Element insertion

Let $P^i = \{\pi_1, \pi_2, \ldots, \pi_\eta\}$ be a population of η elements in an iteration i. For each permutation $\pi_j \in P^i$, applying the local search algorithm (*LocalOpt*(π_j) procedure) a set of local minima $LM^i = \{\hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_\eta\}$ is determined. For any permutation

$$\hat{\pi}_j = (\hat{\pi}_j(1), \hat{\pi}_j(2), \ldots, \hat{\pi}_j(n)), \quad j = 1, 2, \ldots, \eta, \tag{9.3}$$

let

$$nr(a, l) = \left|\{\hat{\pi}_j \in LM^i : \quad \hat{\pi}_j(l) = a\}\right|, \tag{9.4}$$

which is a number of permutations from the set LM^i, wherein element a is in the position l. Let $\Xi(i)$ be defined as a fixed level of acceptance ($0 < \Xi(i) < 1$) connected with the iteration i (Ξ can also be constant). If $a \in N$ is a free element and

$$\alpha = \frac{nr(a, l)}{\eta} \geq \Xi(i), \tag{9.5}$$

then the element a is fixed in the position l; $\varphi = 1$ and the quadruple (a, l, α, φ) is inserted to the set of fixed element and positions, that is

$$FS^{i+1} \leftarrow FS^i \cup \{(a, l, \alpha, \varphi)\}. \tag{9.6}$$

9.1.4. Element deletion

To test many local minima each fixed element is released after some number of iterations have been executed. Let the deletion level function Γ be defined so that

$$\forall i, \;\; 0 < \Gamma(i) \leq 1. \tag{9.7}$$

Further, let

$$ES = \left\{(a, l, \alpha, \varphi) \in FS^{i+1} : \;\; \frac{\alpha}{\varphi} \leq \Gamma(i)\right\}. \tag{9.8}$$

It is a set of some elements and positions which are fixed in all permutations of the population P^i, for which $\frac{\alpha}{\varphi}$ (φ is an *age*) is under the deletion level $\Gamma(i)$ defined for an iteration i (Γ can be constant).

If $ES \neq \emptyset$, then elements of this set are deleted from FS^{i+1}, that is

$$FS^{i+1} \leftarrow FS^i \backslash ES, \tag{9.9}$$

otherwise (when $ES = \emptyset$), let $\delta = (a', l', \alpha', \varphi') \in FS^{i+1}$ be such that

$$\frac{\alpha'}{\varphi'} = \min\left\{\frac{\alpha}{\varphi} : \ \ (a, l, \alpha, \varphi) \in FS^{i+1}\right\}. \tag{9.10}$$

The element δ is deleted from the set FS^{i+1}, that is

$$FS^{i+1} \leftarrow FS^i \backslash \delta. \tag{9.11}$$

9.1.5. Auto-tuning of the acceptance level

The function Ξ is defined so that for each iteration i $0 < \Xi(i) \leq 1$. It is possible that none of the elements is acceptable to be fixed in an iteration. To prevent this situation an auto-tune procedure for Ξ value is proposed. After calculating the number of elements a in positions l (called $nr(a, l)$), in each iteration i, if

$$\max_{a,l \in \{1,2,\ldots,n\}} \frac{nr(a, l)}{K} < \Xi(i) \tag{9.12}$$

then, $\Xi(i)$ value is fixed as

$$\Xi(i) \leftarrow \max_{a,l \in \{1,2,\ldots,n\}} \frac{nr(a, l)}{K} - \epsilon, \tag{9.13}$$

where ϵ is a small constant, e.g. $\epsilon = 0.05$. In this way the value of $\Xi(i)$ is decreased. Similarly, it is possible to increase this value when it is too small (and too many elements are fixed in one iteration). The function $\Gamma(i)$ should be defined in such a way that each element of the set FS^i is deleted after a number of iterations have been executed.

9.1.6. A new population

If a quadruple $(a, l, \alpha, \varphi) \in FS^{i+1}$ then in each permutation of a new population P^{i+1} there is an element a in a position l. Therefore, to generate a new population P^{i+1} randomly drawn free elements are inserted in remaining free positions of the elements of population P^i. An outline of the procedure is given in Figure 9.2. The function *random* generates an element of the set W from the uniform distribution. The computational complexity of the algorithm is $O(\eta \cdot n)$.

```
Algorithm 9. New Population (NewPopul(FS_{i+1}))
P^{i+1} ← ∅;
Determine a set of free elements
FE ← {a ∈ N : ¬ ∃ (a, l, α, φ) ∈ FS^{i+1}}
and a set of free positions
FP ← {l : ¬ ∃ (a, l, α, φ) ∈ FS^{i+1}};
for j ← to η do   {inserting fixed elements}
  for every (a, l, α, φ) ∈ FS^{i+1} do
    π_j(l) ← a;
  end for;
  W ← FE;
  {inserting free elements}
  for s ← 1 to n do
    if s ∈ FP then π_j(s) ← w, where
      w ← random(W) and W ← W\{w};
  end for;
  P_{i+1} ← P_{i+1} ∪ {π_j}.
end for;
```

Fig. 9.2. Outline of the NewPopul procedure.

9.2. Parallel Population-Based Metaheuristic

For the parallel version of the *PBM* two models of parallelization have been proposed.

Single-thread model. This model executes multiple population-based metaheuristics which synchronize populations in each iteration, i.e., a common global table of fixed elements and positions is used for each processor. In each iteration the average number $nr(a, l)$ of permutations (for all subpopulations) in which there is an element a in a position l is computed. This model is called *cooperative*.

Multiple-thread model. In this model, processes execute independent algorithms (working on different subpopulations) with different parameters of fixing elements in positions. At the end, the best solution of each subpopulation is collected and the best solution of the whole algorithm is chosen. We will call this model *independent*.

A general structure of the parallel population-based metaheuristic using MPI library is given in Figures 9.3 and 9.4.

```
Algorithm 10. Parallel population-based metaheuristic
procedure ParPBM(int n, int benchm_opt, bool stops, bool communication)
  n – number of jobs to schedule;
  benchm_opt – value of the benchmark's near optimal solution, taken from [78];
  stops – if it is true, the algorithm stops after achieving benchm_opt;
  communication – if it is true the algorithm has got a common nr table;
parfor p ← 1..nrtasks do
  best_cost_p ← ∞;
  α_p ← 0.7;
  fixed_p ← 0;
  int nr_p[N_MAX][N_MAX];
  int φ_p[N_MAX]; for i ← 1..N_MAX do φ_p[i] ← 0; end for;
  perm P^p[K_MAX];
  for iter ← 1..R do
    for t ← 0..K − 1 do
      P^p[t] ← Random_Perm();
      int f ← Descent_Search(P^p[t]);
      if f < best_cost_p then best_cost_p ← f; end if;
      int f_min;
      if stops == true then
        MPI_Allreduce(&f, &f_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
        if f_min <= benchm_opt then return f_min;
      end if;
      for i, j ← 1..n do
        nr_p[i][j] ← 0;
      end for;
      for t ← 1..K − 1 do
        for i ← 1..fixed_p do
          nr_p[i][P^p[i]] + +;
        end for;
      end for;
```

Fig. 9.3. Parallel population-based metaheuristic, Part 1.

9.3. Computational experiments

Parallel population-based metaheuristic was implemented in C++ language with the MPI library and it was tested on the Silicon Graphics SGI Altix 3700 Bx2 with 128 Intel Itanium2 1.5 GHz processors and cache-coherent Non-Uniform Memory Access (CC-NUMA), craylinks NUMAflex4 in fat tree topology with the bandwidth 4.3 Gbps. Up to 16 processors of the supercomputer were used. Computational experiments were done to check the speed of convergence of the parallel algorithm in two proposed models of communication and to compare the results obtained with the benchmarks from the literature [78] and the latest results obtained for this single machine problem [79, 173, 172]).

```
      if communication == true then
        int new_count[N_MAX][N_MAX];
        MPI_Allreduce(nr_p, new_count, (n+1)*(n+1),
          MPI_INT, MPI_SUM, MPI_COMM_WORLD);
        for i, pos ← 1..n do
            nr_p[i][pos] ← new_count[i][pos]/nrtasks;
        end for;
      end if;
      { change α if it is too big or too small, i.e., no elements is fixed or too}
      { many are fixed}
      AutoTune(&pe);
      for pos, i ← 1..n do
        if nr_p[i][pos]/K > pe then
          fixed++;
          φ_p[i]++;
        end if;
      end for;
      for i ← 1..n do
        if φ_p[i] > MAX_AGE then
          φ_p[i] ← 0;
          fixed − −;
        end if;
      end for;
    end for; {t}
  end for; {iter}
  return f;
end parfor;
```

Fig. 9.4. Parallel population-based metaheuristic, Part 2.

In Tables A.5 and A.6 (Appendix A) results of computational experiments for the scheduling problem $1|s_{ij}|\sum w_i T_i$ are presented with the new upper bounds marked. As we can see in Tables A.5 and A.6 it was possible to find 65 new upper bounds of the optimal cost function for the 120 benchmark instances. The average percentage deviation from the solutions of Cicirello and Smith [78] was on the level of -12.08% and was better than in earlier approaches proposed for this problem (Cicirello and Smith [78] and upper bounds from Cicirello [79], Lin and Ying [173] and Liao and Juan [172]).

Two criteria of the algorithm termination were checked. The first one stops the algorithm after having achieved the benchmark value from [78] or exceeding $R = 10$ iterations. This criterion was helpful to determine the speedup of the parallel algorithm tested for two models: the independent model and the model with communication. Results of computations for this criterion of the algorithm termination are presented in Tables A.2, A.3 (convergence), Appendix A, and in Figures 9.5 and 9.6.

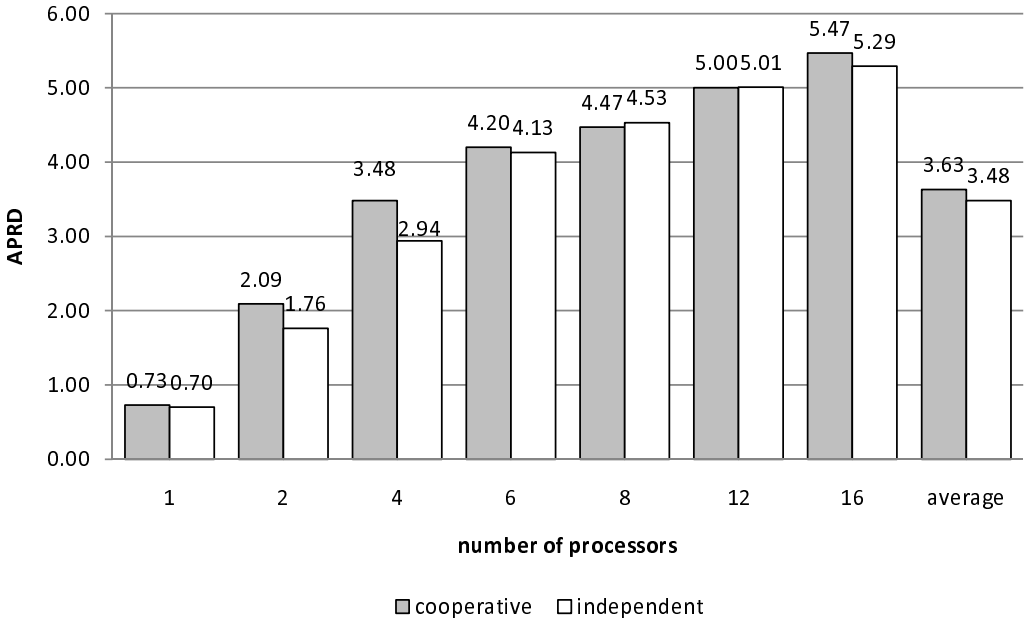


Fig. 9.5. Improvement of the reference solution of Cicirello [79] made by ParPBM algorithms (stop criterion: exceeding 10,000 sec.).

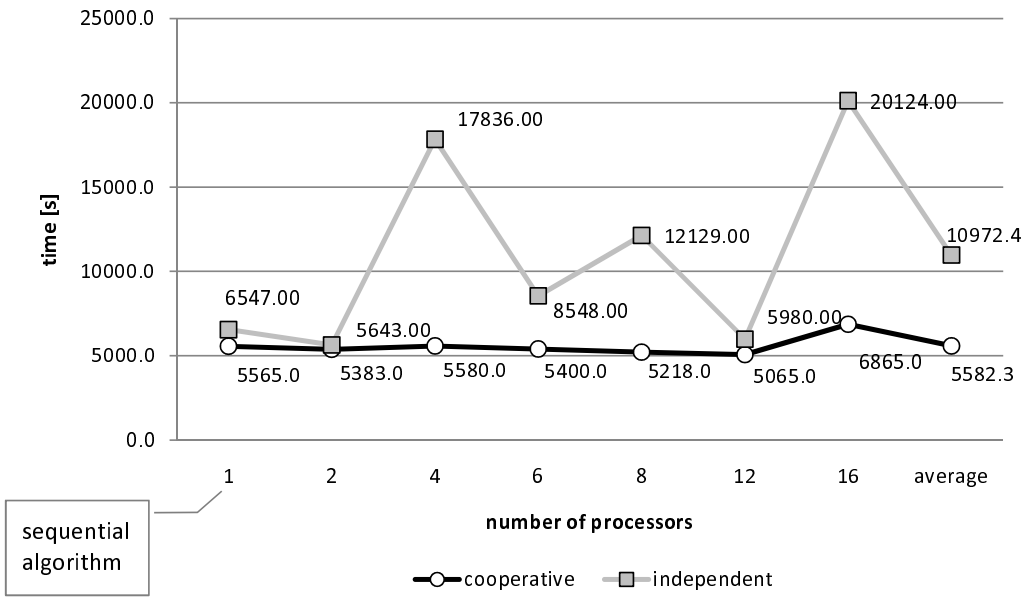


Fig. 9.6. Total time of ParPBM algorithms (stop criterion: APRD = −0.3%).

Table 9.1. Results of APRD (%) of the SA, GA and TS from Lin and Ying [173] compared to ParPBM approach.

Problem set	SA	GA	TS	ParPBM	$\sigma^{ParPBM*}$
1 to 10	20.00	22.83	19.12	21.46	3.47
11 to 20	20.89	27.60	18.46	36.94	33.62
21 to 30	30.39	30.93	29.18	26.28	31.85
31 to 40	6.86	6.42	5.81	17.71	3.59
41 to 50	5.21	5.65	5.33	5.01	2.02
51 to 60	5.29	5.65	4.44	7.96	6.11
61 to 70	7.25	6.56	7.25	7.27	3.80
71 to 80	15.39	15.02	16.32	14.28	5.80
81 to 90	0.66	0.56	0.56	4.91	0.40
91 to 100	−0.47	−0.50	−0.11	0.90	0.64
101 to 110	0.60	0.24	0.64	3.14	0.45
111 to 120	−0.23	−0.44	−0.23	0.61	1.01
average	**9.32**	**9.97**	**8.90**	**12.08**	**7.73**

* Standard deviation of the ParPBM results σ^{PHM} is determined over 10 runs.

As we can see in Tables A.5 and A.6 the cooperative model of the ParPBM has shorter real times of execution (t_{total}) than the independent one. Also the time consumed by all the processors (t_{cpu}) is shorter for the cooperative model of communication. It means that the cooperative model obtains faster the same solutions as the independent model does.

The second criterion of the algorithm termination determines the speed of the parallel algorithm convergence. Algorithms execute exactly $R = 10$ iterations. Results of computations for this criterion are presented in Table 9.1. Three t-Student tests of statistical significance show that the average value is better than in other approaches for the cooperative model of communication with the standard significance level $\alpha = 0.05$: $H_0 : m_1 = m_i, H_1 : m_1 > m_i, i = 2, 3, 4$, where m_1, m_2, m_3 and m_4 denote an (unknown) APRD of algorithms ParPBM, SA, GA and TS, respectively, for any set of test instances. Values of test statistics equal $Z_1 = 2.48, Z_2 = 1.73$ and $Z_3 = 2.93$, respectively. The critical set for $\alpha = 0.05$ is $[1.65, \infty)$ (from the normal distribution); all the values of test statistics do not belong to the critical set, so we reject H_0 and take the hypothesis H_1 which says that the APRD of the ParPBM algorithm is greater than APRDs of SA, GA and TS approaches, respectively.

9.4. Remarks and conclusions

We proposed the new approach to the permutation optimization problems grown on the parallel population-based technology, being the alternative and competitive tool for solving hard scheduling problems. The usage of the population with fixed features of local optima makes the performance of the method much better than the iterative improvement approaches, such as in tabu search, simulated annealing as well as classical genetic algorithms. This method can be implemented as a multithread master-slave application as well as a distributed algorithm in which a set of fixed elements and positions is determined independently or in cooperation. Due to its simplicity the proposed approach can be easily accommodated to solve any NP-hard discrete optimization problems such as vehicle routing or assignment problems.

Chapter 10

Parallel branch and bound approach

This chapter presents the parallel branch and bound (B&B) algorithm for the single machine total weighted tardiness problem defined in Section 3.3.2. Although this method is not a metaheuristic, it can be used as an approximate method by stopping calculations after a fixed period of time and getting the best solution up-to-now (i.e., the upper bound). In practice, such a method can be used as a heuristic for bigger instances of the problem if the algorithm is executed for a determined time period.

The method with cut tree is known as a *curtailed* B&B. Generally, the cut can be realized as the limit of the amount of computational resources for an algorithm:

a) limited processor work time,

b) limited memory,

c) limited depth of the search tree,

d) limited number of successors of a node during the tree generation process (so-called *beam search*, or *filtered beam search*).

In the algorithm proposed here, we have made use of the new properties of a permutation broken into blocks shown in Section 3.3.3. These properties are much stronger than elimination criteria (see Potts and Van Wassenhove [214], Rinnoy Kan et al. [222]) applied so far and they allow us to eliminate many branches of the solution tree. Parallel implementation of the algorithm enables us to reduce computational time significantly as well as solve larger problems. We have tested the algorithms on randomly generated instances (of up to 80 jobs) and benchmark instances taken from the OR-Library [22]. The solutions obtained have been compared with the results yielded by the best algorithms discussed in

the literature. The results show that the proposed algorithm solves the problem instances with high accuracy in a very short time.

10.1. Enumeration scheme

Each schedule of jobs can be represented by a permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ on the set of jobs $\mathcal{N}$. Let Φ_n denote the set of all such permutations. We will present the generation process of permutations from the set Φ_n as a *search tree* H. We create this tree as follows: from the root node (zero level), where no jobs have been scheduled, we branch to $2n$ different nodes on the first level; each node corresponds to a specific job being scheduled in the 1-st or n-th position. Each of these nodes leads to $2(n-1)$ new nodes on the second level, corresponding to one of the remaining $n-1$ jobs filling the first or the last position of the remaining range of $n-1$ free positions, etc., (see Figure 10.1).

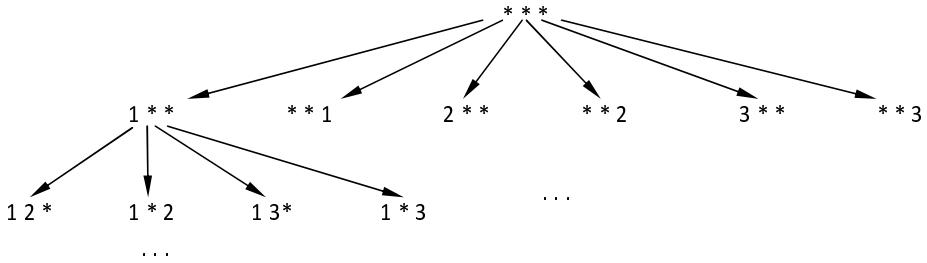


Fig. 10.1. A part of the H tree for $n = 3$ (an asterisk denotes a free job).

Each node π from the h-th level ($h = 0, 1, 2, \ldots, n$) in the tree H is characterized by the sets of *fixed jobs*

$$S^B(\pi) = (\pi(1), \ldots, \pi(s-1)), \quad S^E(\pi) = (\pi(t+1), \ldots, \pi(n)) \tag{10.1}$$

and *free jobs*

$$S(\pi) = (\pi(s), \ldots, \pi(t)), \tag{10.2}$$

where $1 \leq s \leq t+1 \leq n$, $\left|S^B(\pi)\right| = s-1$, $\left|S^E(\pi)\right| = n-t$, $|S(\pi)| = t-s+1$ and $\left|S^B(\pi)\right| + \left|S^E(\pi)\right| = h$. Therefore, a permutation π takes the form

$$\pi = \underbrace{(\pi(1), \ldots, \pi(s-1))}_{S^B(\pi)}, \underbrace{\pi(s), \ldots, \pi(t)}_{S(\pi)}, \underbrace{(\pi(t+1), \ldots, \pi(n))}_{S^E(\pi)}. \tag{10.3}$$

Producing from π a new permutation β (node on the $(h+1)$-th level of the tree) consists in fixing in the s-th or t-th position in β one of the free jobs from

the set $S(\pi)$, i.e., changing positions of the fixed job with the job which is in the s-th or t-th position in π and including it in the sets of fixed jobs $S^B(\pi)$ or $S^E(\pi)$. The remaining jobs in the same positions are in both permutations. Obviously, in each successor of permutation β the job fixed in the s-th or t-th position in β will still remain there.

We call the generation of a new permutation (new node in a solution tree) a *move* (insert move). Let k and l ($k \neq l,\ k < n$) be a pair of positions in a permutation

$$\pi = (\pi(1), \ldots, \boxed{\pi(k)}, \pi(k+1), \ldots, , \underline{\pi(l)}, \pi(l+1), \ldots, \pi(n)). \tag{10.4}$$

Thus, the move r_l^k generates a permutation π_l^k in the following way

$$\pi_l^k = (\ldots, \pi(k-1), \pi(k+1), \ldots, \underline{\pi(l)}, \boxed{\pi(k)}, \pi(l+1), \ldots), \tag{10.5}$$

if $k < l$, and

$$\pi_l^k = (\ldots, \pi(l-1), \boxed{\pi(k)}, \underline{\pi(l)}, \ldots, \pi(k-1), \pi(k+1), \ldots), \tag{10.6}$$

if $k > l$. A permutation (node) π_l^k ($1 \leq k, l \leq n$) is a root of a subtree in the solution tree H. This subtree contains all permutations which can be generated from π_l^k shifting jobs from the set of free jobs $S(\pi_l^k)$.

10.1.1. Lower bound

Let π be a node of the tree H on the h-th level as it was defined in (10.1) and (10.2). The lower bound $LB(S(\pi))$ of costs of all possible schedules generated from π can be defined as follows

$$LB(\pi) = F(S^B(\pi)) + F(S^E(\pi)) + LB(S(\pi)), \tag{10.7}$$

where

$$F(S^B(\pi)) = \sum_{i=1}^{s-1} f_{\pi(i)}\left(C_{\pi(i)}\right) \text{ and} \tag{10.8}$$

$$F(S^E(\pi)) = \sum_{i=t+1}^{n} f_{\pi(i)}\left(C_{\pi(i)}\right), \tag{10.9}$$

is the cost of executing fixed jobs and $LB(S(\pi))$ is the lower bound of executing free jobs. We will calculate $LB(S(\pi))$ applying two methods.

```
Algorithm 11.  LB^G
begin
    LB^G(S(π)) ← 0;  W ← S(π);  P ← Σ_{i=1}^{t} p_{π(i)};
    Execute |S(π)| times:
    if  there exists π(i) ∈ W such that d_{π(i)} ≥ P then
      W ← W \ {π(i)} and P ← P − p_{π(i)}
    else
      LB^G(S(π)) ← LB^G(S(π)) + min_{π(i)∈W} {f_{π(i)}(P)}
      and   P ← P − max_{π(i)∈W} {p_{π(i)}}
end.
```

Fig. 10.2. Outline of the lower bound from the greedy method (LB^G) algorithm.

A lower bound from the greedy method

The lower bound $LB^G(S(\pi))$ of execution costs of free jobs can be calculated as is shown in Figure 10.2. It is easy to prove that for any permutation γ of jobs from the set $S(\pi)$, $F(\gamma) \geq LB^G(S(\pi))$.

A lower bound from the assignment problem

Let

$$T(q) = \min\{P(Q) : Q \subset S(\pi)\},$$

where $q = |Q|$ for any $Q \subseteq N$, $\quad P(Q) = \sum_{\pi(i)\in Q} p_{\pi(i)}$. Next, we calculate

$$t_{ij} = P(S^B(\pi)) + p_{\pi(i)} + T\,(j-1) \text{ and} \tag{10.10}$$

$$c_{ij} = f_{\pi(i)}(t_{ij}), \quad i,j = 1,2,\ldots,k, \quad k = |S(\pi)|\,. \tag{10.11}$$

The lower bound $LB^{AP}(S(\pi))$ of execution cost of free jobs from the set $S(\pi)$ equals the optimal solution value for the following assignment problem

$$\sum_{i=1}^{k}\sum_{j=1}^{k} c_{ij}x_{ij} \to \min_{x} \tag{10.12}$$

$$x_{ij} \in \{0,1\}\,, \ \sum_{i=1}^{k} x_{ij} = 1, \ \sum_{j=1}^{k} x_{ij} = 1, \ \ i,j = 1,2,\ldots,k. \tag{10.13}$$

In the paper of Rinnoy Kan et al. [222] the lower bound for the TWTS problem is calculated in a similar way. Finally, we define the lower bound of the execution of free jobs as the maximum of values obtained from the greedy method and the assignment problem

$$LB(S(\pi)) = \max\{LB^G(S(\pi)),\ LB^{AP}(S(\pi))\}. \tag{10.14}$$

Therefore, if π^* is the best solution known so far and

$$LB(\pi) \geq F(\pi^*), \tag{10.15}$$

then permutation π (node from the tree H) can be eliminated.

10.1.2. Branching rule

Let π be a node of the tree H. Let us select a free job (the move) from the set $S(\pi) = \{\pi(s), \ldots, \pi(t)\}$. Fixing this job in position s or t will generate a new permutation which is a direct successor of π in the tree H. By $L(\pi) = \{r_s^i :\ i \in S(\pi)\}$ let us denote the set of candidates of moves which determine free jobs in position s and by $R(\pi) = \{r_t^i :\ i \in S(\pi)\}$ the set of candidates of moves which determine jobs in position t.

Let $r_s^l \in L(\pi)$, and take elements

$$\mathcal{I} = \{i \in \mathcal{N} :\ s \leq i < l \text{ and } C_{\pi(i)} > d_{\pi(i)}\} \tag{10.16}$$

and

$$\delta_s^l(\pi) = f_{\pi(l)}(C_{\pi(s-1)} + p_{\pi(l)}) - f_{\pi(l)}(C_{\pi(l)}) + p_{\pi(l)} \sum_{i \in \mathcal{I}} w_{\pi(i)}. \tag{10.17}$$

Next, for $r_t^l \in R(\pi)$

$$\delta_t^l(\pi) = f_{\pi(l)}(C_{\pi(t)}) - f_{\pi(l)}(C_{\pi(l)}) - p_{\pi(l)} \sum_{j \in \mathcal{J}} w_{\pi(j)}, \tag{10.18}$$

where $\mathcal{J} = \{j \in \mathcal{N} :\ l < j \leq t \text{ and } C_{\pi(j)} > d_{\pi(j)}\}$.

Theorem 10.1. *If β is a permutation generated from π by move $r_s^l \in L(\pi)$, then*

$$F(\beta) \geq F(\pi) + \delta_s^l(\pi), \tag{10.19}$$

and, if it is generated by move $r_t^l \in R(\pi)$, then

$$F(\beta) \geq F(\pi) + \delta_t^l(\pi). \tag{10.20}$$

Proof. Let a permutation π be a vertex on the level $h = s - 1 + n - t$ in the solution tree H. A set of free jobs in π, $S(\pi) = (\pi(s), \pi(s+1), \ldots, \pi(t-1), \pi(t))$. A move $r_s^l \in L(\pi)$ $(s \leq l \leq t)$ generates a new permutation (a vertex in the solution tree H) $\beta = r_s^l(\pi)$ such that

$$\beta(i) = \pi(i), \quad i = 1, 2, \ldots, s-1, l+1, \ldots, n, \tag{10.21}$$

$$\beta(s) = \pi(l) \text{ and } \beta(j+1) = \pi(j), \; j = s, s+1, \ldots, l-1. \tag{10.22}$$

Let

$$\mathcal{X} = \{i \in N : \; s \leq i < l, \; C_{\pi(i)} > d_{\pi(i)}\}, \tag{10.23}$$

$$\mathcal{Z} = \{i \in N : \; s \leq i < l, \; C_{\pi(i)} \leq d_{\pi(i)} \text{ and } C_{\pi(i)} + p_{\pi(l)} > d_{\pi(i)}\}. \tag{10.24}$$

For $s + 1 \leq k < l$ the element $\beta(k+1) = \pi(k)$. We consider three exhaustive cases: '$\pi(k) \in \mathcal{X}$', '$\pi(k) \in \mathcal{Z}$' and '$\pi(k) \notin (\mathcal{X} \cup \mathcal{Z})$'.

Case 1. '$\pi(k) \in \mathcal{X}$'. Then

$$f_{\beta(k+1)}(C_{\beta(k+1)}) = f_{\pi(k)}(C_{\pi(k)}) + w_{\pi(k)} \cdot p_{\pi(l)}. \tag{10.25}$$

Case 2. '$\pi(k) \in \mathcal{Z}$'. In this case

$$f_{\beta(k+1)}(C_{\beta(k+1)}) = f_{\pi(k)}(C_{\pi(k)}) + w_{\pi(k)} \cdot (C_{\pi(k)} + p_{\pi(l)} - d_{\pi(k)}). \tag{10.26}$$

Since $f_{\pi(k)}(C_{\pi(k)}) = 0$, so

$$f_{\beta(k+1)}(C_{\beta(k+1)}) =$$

$$= w_{\pi(k)} \cdot (C_{\pi(k)} + p_{\pi(l)} - d_{\pi(k)}) = f_{\pi(k)}(C_{\pi(k)} + p_{\pi(l)}). \tag{10.27}$$

Case 3. '$\pi(k) \notin (\mathcal{X} \cup \mathcal{Z})$'. Then

$$f_{\beta(k+1)}(C_{\beta(k+1)}) = f_{\pi(k)}(C_{\pi(k)}) = 0. \tag{10.28}$$

Since $\beta(s) = \pi(l)$, so

$$f_{\beta(s)}(C_{\beta(s)}) = f_{\pi(l)}(C_{\pi(s-1)} + p_{\pi(l)}). \tag{10.29}$$

The goal function value

$$F(\beta) = \sum_{i=1}^{s-1} f_{\beta(i)}(C_{\beta(i)}) + f_{\beta(s)}(C_{\beta(s)}) +$$

$$+ \sum_{i=s+1}^{l} f_{\beta(i)}(C_{\beta(i)}) + \sum_{i=l+1}^{n} f_{\beta(i)}(C_{\beta(i)}) = \sum_{i=1}^{s-1} f_{\pi(i)}(C_{\pi(i)}) +$$

$$+ f_{\pi(l)}(C_{\pi(s-1)} + p_{\pi(l)}) + \sum_{i=s}^{l-1} f_{\pi(i)}(C_{\pi(i)}) +$$

$$+ \sum_{\pi(i)\in\mathcal{I}} w_{\pi(i)} \cdot p_{\pi(l)} + \sum_{\pi(i)\in\mathcal{J}} f_{\pi(i)}(C_{\pi(i)} + p_{\pi(l)}) +$$

$$+ \sum_{i=l+1}^{n} f_{\pi(i)}(C_{\pi(i)}) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)}) + f_{\pi(l)}(C_{\pi(s-1)} +$$

$$+ p_{\pi(l)}) - f_{\pi(l)}(C_{\pi(l)}) + p_{\pi(l)} \cdot \left(\sum_{\pi(i)\in\mathcal{I}} w_{\pi(i)} \right) +$$

$$+ \sum_{\pi(i)\in\mathcal{J}} f_{\pi(i)}(C_{\pi(i)} + p_{\pi(l)}) = F(\pi) + \delta(r_s^l). \tag{10.30}$$

Similarly, we can prove that for a move $r_t^l \in R(\pi)$ $(s \leq l < t)$

$$F(\beta) = F(\pi) + \delta(r_t^l), \tag{10.31}$$

which ends the proof of the theorem. ∎

Therefore, expression $F(\pi) + \delta_s^l(\pi)$ or $F(\pi) + \delta_t^l(\pi)$ is a lower bound of a permutation weight generated from π by fixing the free job $\pi(l) \in S(\pi)$ in the s-th or t-th position. While the algorithm progresses we will choose jobs which after having been fixed will generate a permutation – a direct successor which has the smallest possible weight (i.e., which has the smallest $\delta_s^l(\pi)$, $r_s^l \in L(\pi)$ and $\delta_t^l(\pi)$, $r_t^l \in R(\pi)$).

10.2. Branch and bound algorithm

The starting point of the algorithm (the root of solution tree H) is a permutation π^0, the sets of fixed jobs $S^B(\pi^0) = S^E(\pi^0) = \emptyset$ $(s = 1, \quad t = n)$ and of free jobs $S(\pi^0) = N$. Let us assume $\pi^* \leftarrow \pi^0$ as the best solution and let the upper bound $UB = F(\pi^*)$. The tree level is $h = 0$. Let π be a permutation (node) on the h-th level of the tree H. The sets of fixed jobs $S^B(\pi) = (\pi(1), \pi(2), \ldots, \pi(s-1))$, $S^E(\pi) = (\pi(t+1), \pi(t+2), \ldots, \pi(n))$ and of free jobs $S = \{\pi(s), \ldots, \pi(t)\}$, where $h = s - 1 + n - t$. The quality of solutions calculated by the branch and bound algorithm depends on the starting point, too. For π^0 we set the best solution determined by one of the heuristic algorithms: SWPT, EDD, AU and COVERT, [213]. An outline of the branch and bound method is presented in Figure 10.3.

Algorithm 12. B&B

Step 1: {*Lower bound*}
if $LB(\pi) \geq UB$ then go to Step 5;

Step 2: {*Upper bound*}
if $F(\pi) < UB$ then $UB \leftarrow F(\pi)$, $\pi^* \leftarrow \pi$;

Step 3: {*Set of candidates*}
Determine set of candidate moves $L(\pi)$ and $R(\pi)$;

Step 4: {*Calculations*}
if $L(\pi) \cup R(\pi) = \emptyset$ then go to Step 5;
Select a move r_l^k, such that:
$\delta_l^k = \min\{\min_{r_s^i \in L(\pi)}\{\delta_s^i\}, \min_{r_t^i \in R(\pi)}\{\delta_t^i\}\}$;
Generate new permutation β (node in H) by
executing move r_l^k.
if $r_l^k \in L(\pi)$ then
  determine $\pi(k)$ on position s in β
and $s \leftarrow s + 1$
else (i.e., $r_l^k \in R(\pi)$)
  determine the job $\pi(k)$ on position t in β
and $t \leftarrow t - 1$;
Let $h \leftarrow h + 1$; $\pi \leftarrow \beta$;
go to Step 1;

Step 5: {*Backtrack*}
if π is the root of the tree
then exit; {π^* *is an optimal solution*}
if permutation π was generated from β
by move r_l^k then
  if$r_l^k \in L(\pi)$ then
  $s \leftarrow s - 1$ **and** $L(\pi) \leftarrow L(\pi) \backslash \{r_l^k\}$
  else (i.e., $r_l^k \in R(\pi)$)
  $t \leftarrow t + 1$ **and** $R(\pi) \leftarrow R(\pi) \backslash \{r_l^k\}$;
$h \leftarrow h - 1$;
go to Step 4;

Fig. 10.3. Outline of the Branch and Bound (B&B) method.

10.2.1. Parallel algorithm

The parallel algorithm was implemented for the SIMD model of parallel processors without shared memory. Each processor has its own local memory with a short time of access; the communication between processors is very slow (compared to

```
 Algorithm 13.  Parallel B&B
for (each processor p = 1, 2, ...)
  begin
    Heap : heap;   {local for each processor}
    while Heap ≠ ∅
      if LB(π) < UB then
      begin
        π ← Get(Heap);
        if F(π) < UB then
        begin
          UB ← F(π);   π* ← π;
           broadcast π* to other processors
        end;
        if L(π) ∪ R(π) = ∅ then Backtrack
        else select minimal move in L(π) ∪ R(π)
        and generate new permutation
        (node in tree H);
        Put(Heap, β)
      end
  end.
```

Fig. 10.4. Outline of the parallel B&B.

the local-memory access). A general scheme of the parallel algorithm is given in Figure 10.4.

The main idea of the parallel algorithm is to make a concurrent multiple-walk search process on the solution tree H. Each processor has a set of vertices to search and a local value of the upper bound UB. If every processor had the latest value of the best upper bound at any moment, the speedup (compared to the sequential algorithm) would be the greatest. But broadcasting the upper bound costs, i.e., the time of communication between processors is very long. That is why the frequency of communication between processors (broadcasting of the latest value of the upper bound) has to be low. In our implementation the processor is getting a new value of UB when it wants to broadcast its own π^*.

10.3. Computer simulations

The algorithm was implemented in Ada95 language and ran on the SGI Altix 3700 Bx2 supercomputer installed in Wrocław Centre of Networking and Supercomputing [266] under the Novell SUSE Linux Enterprise Server operating system.

Table 10.1. The number of iterations (over all processors) and the time of computing.

n	1 processor	time**	2 processors	4 processors
20	18 821	4	18 819	**18 814***
30	53 133	27	**53 123**	53 125
40	96 818	54	96 818	96 818
50	160 379	126	160 337	**160 315**
60	246 295	381	246 306	**246 290**
70	347 290	589	347 267	**347 260**
80	490 649	937	490 650	**490 535**

* Situations where parallel algorithm executes less iterations than the sequential one are marked with bold font.

** The average time of computing for single instance in seconds.

The tasks of Ada95 language were executed in parallel as system threads. Test problems were randomly generated (adapting the generation scheme proposed by [214]) and 125 instances were given for each size $n = 20, \ldots, 80$. For each job i, an integer processing time p_i was generated from the uniform distribution [1,100] and, for weighted tardiness problems, an integer weight w_i was generated from the uniform distribution [1,10]. Problem hardness is likely to depend on the relative range of due dates (RDD) and on the average tardiness factor (TF). Having computed and selected values of RDD and TF from the set 0.2, 0.4, 0.6, 0.8, 1.0, an integer due date d_i from the uniform distribution $[P(1 - TF - RDD/2)$, $P(1 - TF + RDD/2)]$ was generated for each job i. Five problems were generated for each of the 25 pairs of values of RDD and TF, yielding 125 problems for each value of n.

Table 10.1 shows average number of iterations performed by the algorithm for varying number of processors and average computing times. As we can notice a parallel algorithm performs on average a smaller number of iterations than the sequential algorithm. This effect can also be observed in Figure 10.5. In the parallel algorithm the number of iterations is computed as the sum of iterations for each processor, so the speedup we get may be almost-linear, or even superlinear, because of the rare communication between processors. Such speedup anomalies were seen in the context of branch and bound algorithms (Lai and Sahni [164], Mans and Roucairol [178]), as well as in the context of parallel tabu search and simulated annealing algorithms (Bożejko and Wodecki [65], Porto and Ribeiro [209], Wodecki and Bożejko [269]). Anomalies can appear due to wrong decisions

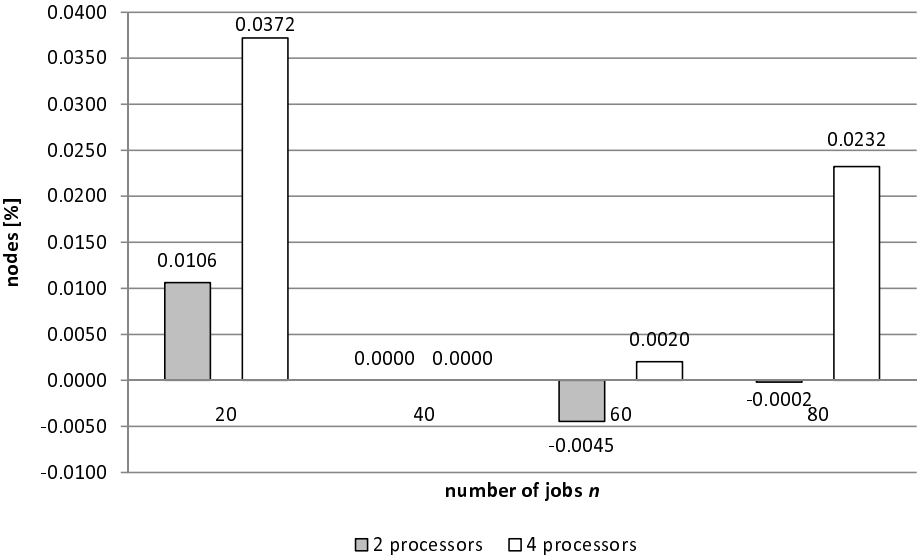


Fig. 10.5. Percentage improvement of the number of searched nodes of the parallel B&B compared to the sequential B&B algorithm.

made by sequential algorithms. As we can notice the number of iterations performed by algorithms grows exponentially, but much more slowly than it takes place in the known branch and bound algorithms for this problem, which cannot solve problems for $n > 50$. Our parallel algorithm can solve problems even for $n > 100$, for some instances, in 90 minutes.

10.4. Remarks and conclusions

This chapter provides us with a practical, sequential and parallel branch and bound algorithm for the single machine total weighted tardiness problem. Preliminary calculation allowed us to suppose that in the case of using parallel systems it will be possible to solve bigger instances (more than 50 jobs), especially if we use stronger lower bounds and additional elimination criteria. The proposed methodology can be applied to any discrete optimization problem in which a solution is represented as a permutation. Superlinear speedup effect, described also in the literature in the context of parallel B&B as a speedup anomaly, has been observed during computational experiments.

Chapter 11

Parallel simulated annealing

The aim of this chapter is to propose a new cooperative simulated annealing approach designed to solve hard discrete optimization problems. We present two simulated annealing algorithms (sequential and parallel) for the permutation flow shop sequencing problem. Two approaches to the simulated annealing method parallelization have been presented: (1) classic, multiple-walk, with parallel generation of trajectories, and (2) more expanded one, using additional backtrack jump, multimoves and temperature steering which makes it possible to intensify and diversify the searching process. The first approach is presented for the flow shop problems with the objective of minimizing the makespan, the second approach – with the sum of job completion times.

We propose a neighborhood applying block properties of jobs on a critical path and specific acceptance function. We also use the lower bound of cost function. By computer simulations conducted by Taillard [243] and other random problems, it is shown that the performance of the proposed algorithms is comparable with other random heuristic techniques discussed in the literature, but with much shorter computing time. The proposed methodology can be applied in any local search procedures.

11.1. Makespan criterion

We take into consideration the permutation flow shop scheduling problem defined in Section 3.4, denoted as $F||C_{\max}$ in the literature. The objective is to find a schedule that minimizes the completion time of the last job. The problem under consideration will be used as a case study to present the general methodology of the multithread simulated annealing method implementation as a multiple-walk parallelization.

11.1.1. Simulated annealing method

The general idea of the SA method was described in Section 2.1.2. Here we will extend the SA algorithm description with detailed implementable elements. Let us consider the notion for solutions of the flow shop problem as it was defined in Section 3.4. In each iteration of simulated annealing a random perturbation is made to the current solution $\pi \in \Phi_n$, giving rise to the set $\mathcal{N}(\pi)$ of *neighbors*. A neighbor $\beta \in \mathcal{N}(\pi)$ is accepted as the next configuration with probability function $\Psi_t(\pi, \beta)$. The $\Psi_t(\pi,\beta)$ is known as *acceptance function* and depends on control parameter t (*temperature*). Its value changes at suitably chosen intervals. In practice the function $\Psi_t(\pi,\beta)$ is chosen in such a way that solutions corresponding to large increases in cost have a small probability of being accepted, whereas solutions corresponding to small increases in cost have a greater probability of being accepted. A standard simulated annealing algorithm can be written as in Figure 11.1. The best solution found so far is represented by π^*, L is the number

```
Algorithm 14.  Standard simulated annealing algorithm
Let π ∈ Φ_n be an initial solution; π* ← π; i ← 0;
repeat
     while i ≤ L do
     begin
          i ← i + 1;
          Randomly generate a solution β
          from the neighborhood N(π) of the current solution π;
          if C_max(β) < C_max(π*) then π* ← β;
          if C_max(β) < C_max(π) then π ← β else
               if Ψ_t(π, β) > random[0, 1)  then π ← β
     end; {i}
     i ← 0; modify control parameter t;
until Stop Criterion;
```

Fig. 11.1. Outline of the simulated annealing algorithm.

of iterations for the fixed value of the parameter t. The *initial solution* π of the algorithm is devised by the heuristic method NEH (Navaz, Enscore, Ham [194]).

Let B_k $(k = 1, 2, \ldots, m)$ be the k-th block in permutation π, B_k^f and B_k^l the subblocks (see Section 3.4.3). For job $j \in B_k^f$ let us denote by $N_k^f(j)$ a set of permutations created by moving the job j to the beginning of the block B_k (before the first job in the block $\pi(f_k)$). Analogously, for the job $j \in B_k^f$ let us denote by $N_k^l(j)$ a set of permutations created by moving the job j to the end of the block B_k (after the last job in the block $\pi(l_k)$). The neighborhood of the

solution π is as follows

$$\mathcal{N}(\pi) = \bigcup_{k=1}^{m} \bigcup_{j \in B_k} (N_k^f(j) \cup N_k^l(j)). \tag{11.1}$$

We propose a new probability acceptance function

$$\Psi_t(\pi, \beta) = \exp\left[(-LB(\beta) + C_{\max}(\pi^*))\frac{-2\ln(t)}{C_{\max}(\pi^*)}\right], \tag{11.2}$$

where $LB(\beta)$ is a lower bound of the value $C_{\max}(\beta)$. Using blocks in neighborhood creation the *strong connectivity* property is lost, thereby causing lack of theoretical convergence (however there are known excellent metaheuristic algorithms without strong connectivity property, e.g. TSAB algorithm of Nowicki and Smutnicki [196], one of the best metaheuristics for the flow shop problem). Therefore, we propose a new acceptance function which though does not give a theoretical convergence of the whole SA, but is experimentally more beneficial – theoretical convergence conditions are not fulfilled for all practical simulated annealing implementations.

The initial value of control parameter $t \leftarrow t_0$, $(0 < t \leq 1)$, where t_0 is the probability of accepting a solution which is worse by half compared to the best solution π^*. To modify parameter t we use a geometric decreasing scheme: $t \leftarrow t * a$, $(0 < a < 1)$. If there is no improvement of the best solution π^* after T_iter iterations, then $t \leftarrow t_0$. The algorithm stops after Max_iter iterations.

11.1.2. Parallel concepts

The chosen model of parallel computing is the SIMD machine of processors without shared memory – with the time of communication between processors much longer than the time of communication inside the process which is being executed on one processor. There are two ways of parallelization used here. One method is simultaneous independent search – concurrently executing a number of independent simulated annealing algorithms without any communication between them and selecting the best solution from solutions obtained by all processes. The other method is to broadcast the best solution of one processor to the other processors when the new best solution is found.

As we have mentioned in Section 2.1.2, parallel simulated annealing (with move acceleration parallelism) has been theoretically proved to be convergent to the global optimum. However, in our implementation even sequential SA has no convergence (because of the lack of strong connectivity of the blocks neighborhood), that is why neither the parallel SA is theoretically convergent. In practice, convergence is only a minor property of a metaheuristic, not connected with its real efficiency.

Frequency of communication between processors (broadcasting of the π^*) is very important for this parallel algorithm performance. This must not take place very often because of the long time of communication between processors. In this implementation processor is getting a new value of π^* only when it wants to broadcast its own π^* (so it exchanges and compares the best solutions with its own π^*). An outline of the parallel SA algorithm is presented in Figure 11.2.

```
Algorithm 15.  Parallel SA with broadcasting
Let π ∈ Φ_n be an initial solution (the same for each of p processors).
π*← π; i ← 0; (all variables are local)
parfor j = 1, 2, ..., p
      while i ≤ L do
      begin
            i ← i + 1;
            Randomly generate a solution β
            from the neighborhood N(π) of the current solution π;
            if C_max(β) < C_max(π*) then
                  π* ← β;   broadcast π* to other processors with
                  comparing to others (exchanging π*);
            if C_max(β) < C_max(π) then π ← β else
                  if Ψ_t(π, β) > random[0, 1) then π ← β
      end; {i}
      i ← 0; modify control parameter t
end parfor.
```

Fig. 11.2. Outline of the parallel SA with broadcasting.

11.1.3. Computational experiments

The algorithm has been tested in several commonly employed instances of various size and hardness levels:

a) 50 instances of 12 different sizes with 100, ..., 500 operations ($n \times m$ = 20×5, 20×10, 20×20, 50×5, 50×10) due to Taillard [243], (from the OR-Library: [22]),

b) 100 instances of 5 different sizes with 2,000, ..., 10,000 operations (200×5, 200×10, 200×20, 200×25, 200×50).

The computational results are presented in Tables A.4 and A.7 (in Appendix A). We used the following parameter specifications in algorithms:

t_0	=	0.5	–	initial value of control parameter,
a	=	0.98	–	constant in control parameter formula,
L	=	n	–	number of iterations for fixed parameter t,
T_iter	=	10	–	number of iterations without improvement of the best solution after which parameter t is set to t_0.

All algorithms were implemented in Ada95 language and ran on the SGI Altix 3700 Bx2 supercomputer installed in Wrocław Centre of Networking and Supercomputing [266] under the Novell SUSE Linux Enterprise Server operating system. The maximal number of iterations Max_iter is 200 for one-processor implementation and 50 for each of the processors in the implementation for 4 processors (thus we have the same complexity, the value of speedup is 4 – the frequency of communication between processors is very rare so it has no influence at all on complexity estimation). As we can see in Figure 11.3, the results are better for parallel program. So speedup is even greater than 4 in a sense (parallel program needs less than 50 iterations to obtain the same results as sequential algorithm for 200 iterations).

We compare solutions of our algorithm with the best known in the literature approximate algorithm NEH (Navaz, Enscore, Ham [194]).

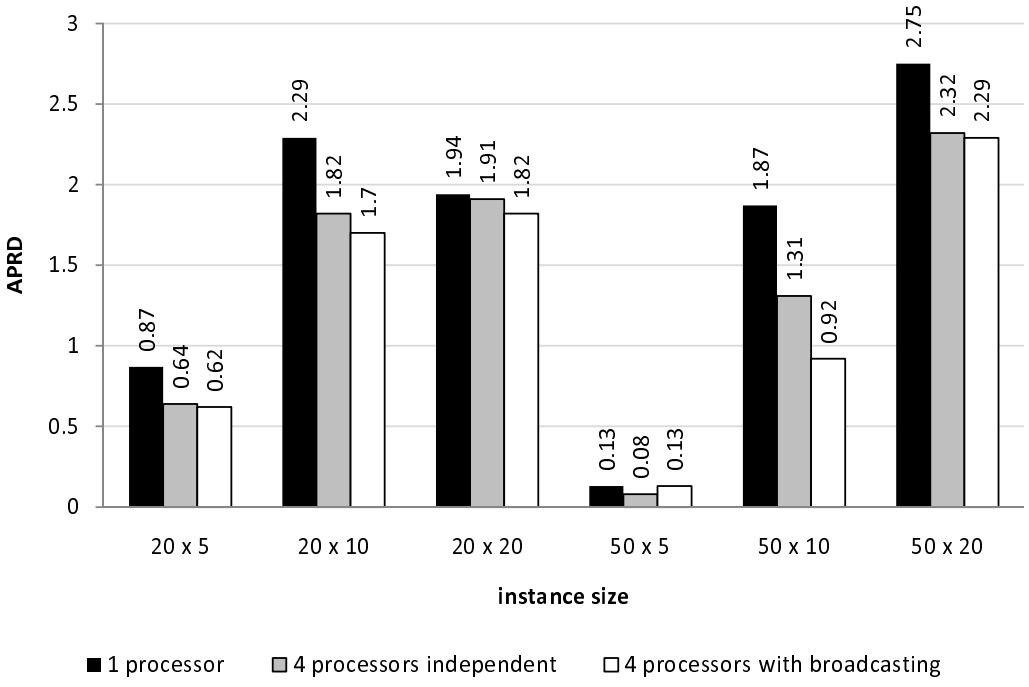


Fig. 11.3. APRD for Taillard [243] instances of the sequential and parallel SA (independent and cooperative, with broadcasting).

As we can see in Figure 11.3 as well as in Tables A.4 and A.7 in Appendix A, results of the parallel algorithm are best for the large values of quotient n and m $(20\times 5, 50\times 5, 100\times 5)$. In such a case the size (length) of blocks is most profitable for sequential and parallel algorithm performance. Besides, improvement of solution value for the parallel algorithm compared to the sequential one was at the level of 18%. The parallel algorithm with broadcasting of the upper bound value was better than the sequential one at the level of 24%, all parallel algorithms being executed with the same number of iterations (as the sum of iterations on each processor) like sequential algorithm.

11.2. Total completion time criterion

In this section, we present a tool for intensifying and diversifying the SA searching process. We take under consideration the flow shop problem with the total completion time criterion $F\|C_{\text{sum}}$.

11.2.1. Intensification and diversification of calculations

Here we present a new acceptance function and cooling scheme. In order to intensify calculations we introduce:

a) backtrack jump – return to the neighborhood where improvement of the current solution takes place,
b) changes of temperature – exact exploration of the promising region.

For better diversification of calculations we will apply:

a) multi-step – moving computations to another remote region of solutions,
b) changes (increasing) of temperature enabling approval of considerably worse solutions.

Acceptance function and cooling scheme. If for randomly determined permutation $\beta \in \mathcal{N}(\pi)$ there occurs $F(\beta) < F(\pi)$, then β is the base solution in the next iteration. On the contrary, i.e., when $F(\beta) \geq F(\pi)$, the probability of acceptance for the base solution in the next iteration is determined by the acceptance function. We propose a new one

$$\Psi_{\alpha,\lambda}(\pi,\beta) = \exp\left[\left(\frac{F(\beta)-F(\pi)}{F(\pi^*)}\right)\cdot\frac{\ln\lambda}{\alpha}\right], \tag{11.3}$$

which also depends on the best solution π^* determined so far. Parameters α and λ $(0<\lambda<1,\ \alpha>0)$ play the role of changing the temperature in the classical acceptance function. We can intensify or diversify calculations by changing these

parameters (cooling scheme). By reducing them we intensify calculations, whereas by increasing their values we make it possible to move away from the current local minimum (diversification of computations).

Backtrack-jump. Let π be the current base solution, t_π – the temperature in the current iteration connected with the current base solution π, and π^* – the best solution determined so far. By LM we denote long-term memory. On LM we will record certain attributes of the algorithm iteration: base solution and temperature. If for a randomly chosen permutation $\delta \in \mathcal{N}(\pi)$, $F(\delta) < F(\pi)$ then we record on LM the pair (π, t_π), in other words, $LM \leftarrow LM \cup (\pi, t_\pi)$.

If 'return condition' comes out (e.g. lack of improvement of the best solution after having executed a fixed number of iterations), then we get the pair (β, t_β) from the LM memory. Instead of the current base solution of the algorithm we take permutation β, and t_β becomes the current temperature. In SA algorithms long-term memory LM is implemented using a stack or a queue.

Multi-step. If during a certain number of iterations the value of objective function is growing, then we execute a multi-step: generating distant permutation (in the sense of the number of moves) from the current base solution. Let $\beta_s, \beta_{s+1}, \ldots, \beta_t$ be a search trajectory, i.e., a sequence of consecutive base solutions. If $F(\beta_s) < F(\beta_{s+1}) <, \ldots, < F(\beta_t)$ and $|t - s| > Lbp$, where Lbp is a fixed parameter, then we execute the multi-step. For fixed parameter k we generate permutation $\delta = r_k(r_{k-1}(, \ldots, r_1(\beta_t), \ldots,))$, where $r_1, r_2, \ldots, r_k$ are randomly generated moves. We take permutation δ as a base solution and $t_\delta \leftarrow t_0$.

11.2.2. Parallel simulated annealing

The first parallelizations of the simulated annealing method were based on the crossing along a single trajectory through the solution space (so-called single-walk strategy). There were two ideas used in this strategy: move acceleration and parallel moves. Kravitz and Rutenbar [159] described one of the first examples of this type of parallelization of the simulated annealing metaheuristic.

Another method, crossing along a multiple trajectory through the solution space (multiple-walk strategy), was proposed by Miki et al. [189] and Czech [92]. This method is based on execution of several threads (independent, semi-independent or cooperative) running simulated annealing at different temperatures. This model was the base of our research.

The new elements of the pSA method: intensification and diversification, multi-step and long-term memory were implemented in parallel as follows. A master-slave model was used in implementation of the parallel algorithm. The master process keeps shared data, such as the best known solution and backtrack-jump heap. Slave processes $i = 1, 2, \ldots, p$ run their own simulated annealing threads with different temperatures $t^i = (\alpha^i, \lambda^i)$. If one process finds a new best solu-

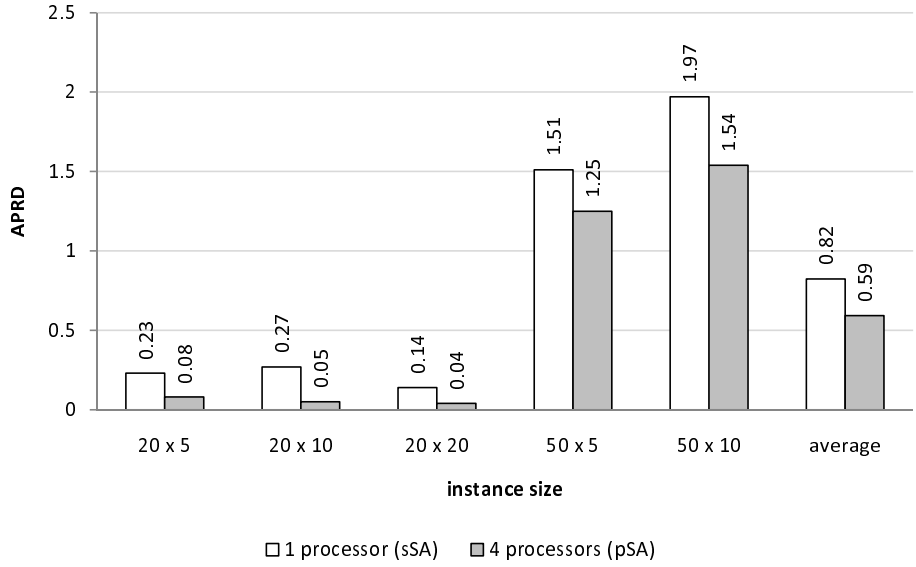


Fig. 11.4. Results of APRD for sSA and pSA algorithms.

tion π^*, then it sends it to the master process and runs an intensification procedure for the next $S = 10$ iterations. Slave processes obtain up-to-date π^* from the master process every $K = 10$ iterations. If the process shows no improvement of its own search procedure for $R = 20$ 'big' iterations (without modification of temperature), it runs the backtrack-jump procedure, consisting in getting new current solutions from the backtrack heap of the master process. Both sequential and parallel algorithms make n iterations without modification of the temperature inside one 'big' iteration.

11.2.3. Computational results

The proposed algorithm was coded in Ada95 and ran on the SGI Altix 3700 Bx2 supercomputer installed in Wrocław Centre of Networking and Supercomputing [266] under the Novell SUSE Linux Enterprise Server operating system, and tested on the benchmark problems taken from the literature (see OR-Library [22] for more details). Main tests were based on 50 instances with $100, \ldots, 500$ operations. Each instance of the test problems was executed 8 times, and the average and minimal result was used for comparison. The standard deviation of results was computed too – it was the measure of algorithm stability.

Results of the tests are shown in Table A.8 in Appendix A. Starting solutions for the first process were taken from the quick approximate algorithm NEH, other processes start with random solutions. For all the algorithms tested, the number of

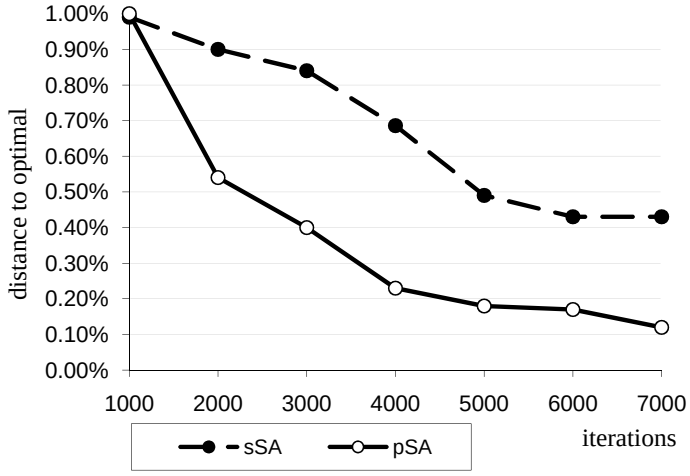


Fig. 11.5. Comparison of convergence for sSA and pSA algorithms.

iterations was given as a sum of iterations on processors. All algorithms, sequential and parallel, make 20,000 iterations, so the 4-processor implementations make 5,000 iterations on each of the 4 processors.

As we can see in Figure 11.4 (which is based on Table A.8 from Appendix A) the average distance to reference solutions for sequential sSA and parallel pSA was at the level of 0.82% for 1-processor implementation and at the level of 0.59% for 4-processor implementation. All algorithms have the same number of iterations and comparable costs. Additionally, the results of the parallel algorithm in 8 runs were more stable: standard deviations of the result equalled 0.13%, compared to 0.18% for the sequential algorithm. Another test was made for comparing algorithms in terms of convergence speed. Tests were conducted on the first 10 instances of the benchmarks [22], the average distance to reference solutions was used for comparison. As we can see in Figure 11.5 the parallel algorithm achieves better results than the sequential algorithm after the same number of iterations (as the sum of iterations on each processor).

11.3. Remarks and conclusions

We discussed an approach to the permutation of flow shop scheduling based on a randomization version of iterative improvement, namely the parallel simulated annealing algorithm based on asynchronous multiple-walk strategy. Parallelization increases the quality of the solutions obtained, the probabilistic element of the algorithm makes simulated annealing much better than the iterative improvement approach. The advantage is especially visible for large problems. As compared to

the sequential algorithm parallelization increases the quality of solutions obtained. The idea of the backtrack-jump and the multi-step was employed. Computer experiments show that the parallel algorithm is also considerably more efficient and stable in comparison to the sequential algorithm.

The pSA algorithm is relatively easy to design. We propose the new annealing scheme which allows cooperation between threads during parallel computations (a classic SA does not cooperate). Making use of this skill allows us to improve results of the pSA.

Chapter 12

Parallel scatter search

The aim of this chapter is to present a parallel variant of the scatter search method, one of most promising methods of combinatorial optimization. The parallel algorithm has been projected and researched experimentally, in the application of flow shop scheduling problems with $C_{\max}$ and C_{sum} criteria. The parallel algorithm based on the scatter search method not only accelerates the computations, but it also improves the quality of the results. In some cases the effect of superlinear speedup has been observed.

12.1. Scatter search method

The main idea of the scatter search method can be found in the paper of James, Rego and Glover [146]. The algorithm is based on the idea of evaluation of the so-called starting solution set. In the classic version a linear combination of the starting solution is used to construct a new solution. In the case of a permutational representation of the solution using linear combination of permutations provides us with an object which is not a permutation. Therefore, in this chapter a path relinking procedure is followed to construct a path from one solution of the starting set to another solution from this set. The best element of such a path is chosen as a candidate to add to the starting solution set. The method stops after having executed a fixed number of max_iter_num iterations. An outline of the (sequential) scatter search method is presented in Figure 12.1.

12.1.1. Path relinking

The base of the path relinking procedure, which connects two solutions π_1, $\pi_2 \in \Phi_n$, is a multi-step crossover fusion (MSXF) described by Reeves and Yamada [215]. Its idea is based on a stochastic local search, starting from solution π_1, to

```
Algorithm 16.  Scatter search
for i ← 1 to max_iter_num do
Step 1. Generate a set of unrepeated
        starting solutions S; n ← |S|.
Step 2. For randomly chosen n/2 pair from the S apply
        path relinking procedure generating a set S' - of n/2 solutions
        which lies on paths.
Step 3. Apply local search procedure to improve value of the cost
        function of solutions from the set S'.
Step 4. Add solutions from the set S' to the set S.
        Leave in the set S at most n solutions by deleting
        the worst and repeated solutions.
Step 5. if |S| < n then
           Add new random solutions to the set S such,
           that elements in the set S does not duplicate and |S| = n.
end for.
```

Fig. 12.1. Outline of the scatter search method.

find a new good solution where another solution π_2 is used as a reference point. The neighborhood $N(\pi)$ of the permutation (individual) π is defined as a set of new permutations that can be obtained from π by exactly one adjacent pairwise exchange operator which exchanges the positions of two adjacent jobs of a problem solution connected with permutation π. The distance measure $d(\pi,\sigma)$ is defined as a number of adjacent pairwise exchanges needed to transform permutation π into permutation σ. Such a measure is known as Kendall's τ measure (measures for permutations are described in Diaconis [99]). The condition of termination consisted in exceeding a given number of iterations. An outline of the procedure is presented in Figure 12.2.

12.2. Parallel scatter search algorithm

The parallel algorithm was designed to be executed on two machines:

- the cluster of 152 dual-core Intel Xeon 2.4 GHz processors connected by Gigabit Ethernet with 3Com SuperStack 3870 switches (for the $F||C_{\text{sum}}$ problem),
- Silicon Graphics SGI Altix 3700 Bx2 with 128 Intel Itanium2 1.5 GHz processors and cache-coherent Non-Uniform Memory Access (CC-NUMA),

craylinks NUMAflex4 in fat tree topology with the bandwidth 4.3 Gbps (for the $F||C_{\max}$ problem),

installed in the Wrocław Center of Networking and Supercomputing (WCNS) [266]. Both supercomputers have got a distributed memory, where each processor has its local cache memory (in the same node) which is accessible in a very short time (compared to the time of access to the memory in another node). Taking into consideration this type of architecture we propose a client-server model for the scatter search algorithm considered here, where calculations of path-relinking procedures are executed by processors on local data and communication takes place rarely to create a common set of new starting solutions. The process of communication and evaluation of the starting solution set S is controlled by a processor number 0. We call this model *global*. Using special properties of the flow shop problem (blocks in the neighborhood determination inside a path-relinking procedure) makes it possible to obtain an efficient method of solving this NP-hard optimization problem.

```
Algorithm 17.  MSXF path-relinking procedure ([215])
Let π1, π2 be reference solutions. Set x = q = π1;
repeat
  For each element yi ∈ N(π), calculate d(yi, π2);
  Sort yi ∈ N(π) in ascending order of d(yi, π2);
  repeat
    Select yi from N(π) with a probability inversely
      proportional to the index i; Calculate F(yi);
    Accept yi with probability 1 if F(yi)    ≤ F(x), and with
      probability PT(yi) = exp((F(x) − F(yi)) / t) otherwise
      (t is a temperature parameter);
    Change the index of yi from i to n and the indices of
      yk, k = i+1, ..., n from k to k−1;
  until yi is accepted;
  x ← yi;
  if F(x) < F(q) then q ← x;
until some termination condition is satisfied;
return q { q is the best solutions lying on the path from π1 to π2 }
```

Fig. 12.2. Outline of the path-relinking procedure.

For comparison a model without communication was also implemented in which independent scatter search threads are executed in parallel. The result of such an algorithm is the best solution out of solutions generated by all the searching threads. We call this model *independent*. Here we also use block properties

inside path-relinking procedure, but there is no communication among concurrently working processors.

Algorithms were implemented in C++ language using MPI (mpich 1.2.7) library and executed under the OpenPBS batching system which measures the times of processor usage. An outline of the procedure is presented in Figure 12.3.

```
Algorithm 18.  Parallel scatter search algorithm
for the SIMD model without shared memory

parfor p := 1 to number_of_processors do
for i := 1 to max_iter_num do
  Step 1.  if (p = 0) then {only procesor number 0}
         Generate a set of unrepeated starting
         solutions S; n ← |S|.
         Broadcast a set S among all the processors.
    else {other processors}
         Receive from the procesor 0 a set of starting solutions S.
    end if;
  Step 2. For randomly chosen n/2 pair from the S
         apply path relinking procedure to generate a
         set S' - of n/2 solutions which lies on paths.
  Step 3. Apply local search procedure to improve
         value of the cost function of solutions from the set S'.
  Step 4. if (p ≠ 0) then
            Send solutions from the set S' to procesor 0
         else  {only processor number 0}
            Receive sets S' from other processors
            and add its elements to the set S
  Step 5. Leave in the set S at most n
            solutions by deleting the worst and repeated solutions.
            if |S| < n then
              Add a new random solutions to the
              set S such, that elements in the set
              S does not duplicate and |S| = n.
            end if;
    end if;
end for;
end parfor.
```

Fig. 12.3. Outline of the parallel scatter search method.

12.3. Computer simulations

Tests were based on 50 instances with 100, ..., 500 operations ($n \times m = 20 \times 5, 20 \times 10, 20 \times 20, 50 \times 5, 50 \times 10$) proposed by Taillard [243], taken from the OR-Library [202]. The results were compared to the best known ones taken from [202] for the $C_{\max}$ criterion. For the C_{sum} flow shop problem, the results obtained were compared to the values of solutions obtained by Reeves and Yamada [215].

12.3.1. Calculations of the $C_{\max}$ criterion

Tables A.9 and A.10 in Appendix A (supplementary tables) present results of computations of the parallel scatter search method for the number of iterations (as a sum of iterations on all the processors) equal 9,600. The cost of computations, understood as a sum of time-consumption on all the processors, is about 7 hours for all the 50 benchmark instances of the flow shop problem. The best results (average percentage deviations from the best known solutions) are obtained by a 2-processor version of the global model of the scatter search algorithm (with communication), which are 70.4% better compared to the average 1-processor implementation (0.029% vs. 0.098%). Since the time-consumption on all processors is a little bit longer than the time of the sequential version we can say that the speedup of this version of the algorithm if almost-linear. For the 4 and 8-processor implementations of the global model and for 2, 4 and 8-processor implementations of the independent model the average results of APRD are better than APRD of the 1-processor versions, but the time-consumption on all processors (t_{cpu}) is *shorter*. That is why these algorithms obtain better results with a smaller cost of computations – the speedup is superlinear. This anomaly can be understood as a situation where the sequential algorithm executes its search threads such that there is a possibility to choose a better path of the solution space trespass, which the parallel algorithm does.

12.3.2. Calculations of the C_{sum} criterion

Tables A.11, A.12, A.13 and A.14 in Appendix A present results of computations of the scatter search method for the number of iterations (as a sum of iterations on all the processors) equal 1,600. Similarly as for the $C_{\max}$ algorithm, the time of sequential computations is about 7 hours for all the 50 benchmark instances of the flow shop problem (Table A.12, A.14, Appendix A). The best results (average percentage deviations from the best known solutions) are obtained by a 2-processor version of the global model of the scatter search algorithm (with communication), see Figure 12.4. Since the time-consumption on all processors is a little bit longer than the time of the sequential version we can say that the

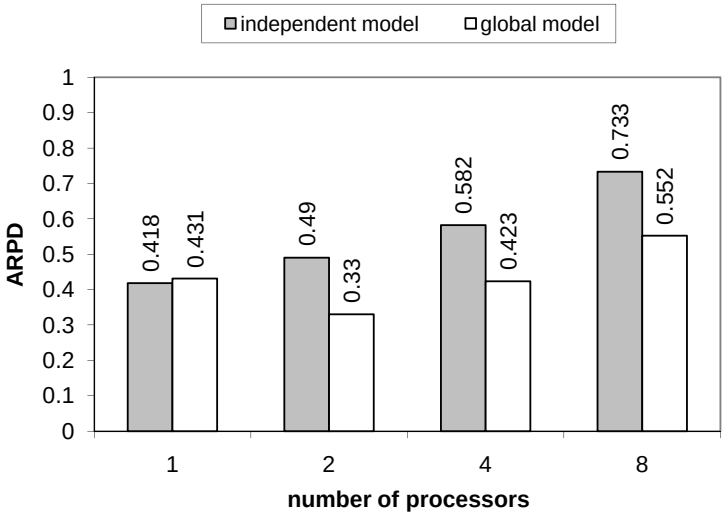


Fig. 12.4. APRD of the global and independent scatter search ($iter = 1{,}600$) for 50 instances from [202].

speedup of this version of the algorithm if almost-linear (or even superlinear, because sequential algorithms, for both: independent and global models, have worse APRD).

The situation is clearer for the number of iterations equal 16,000 (Tables A.15, A.16, A.17 and A.18, Appendix A). The cost of computations, a sum of time-consumption an all processors, is about 75 hours for all the 50 benchmark instances of the flow shop problem (Tables A.16, A.18, Appendix A). The best results are achieved for a 8-processor version of the global model version of scatter search and they are 58.6% better than the results of sequential global scatter search algorithm, and 52.3% better than the results of sequential independent model of scatter search algorithm (see Figure 12.5). The time-consumption on all 8 processors is shorter than the time of both sequential versions. We can say that the speedup of 8-processor global version of the scatter search algorithm is superlinear: better results are achieved with lower cost of computations.

This anomaly can be understood as the situation where the sequential algorithm executes its search threads such that there is a possibility to chose a better path of the solution space trespass, which the parallel algorithm does. As we can observe in Table A.15 such a situation takes place only for the global model of the scatter search algorithms – independent searches are not so effective. More about this anomaly can be found in *Speedup calculation* (Section 12.4). The advantage of the global model of calculations over the independent searches is specially vis-

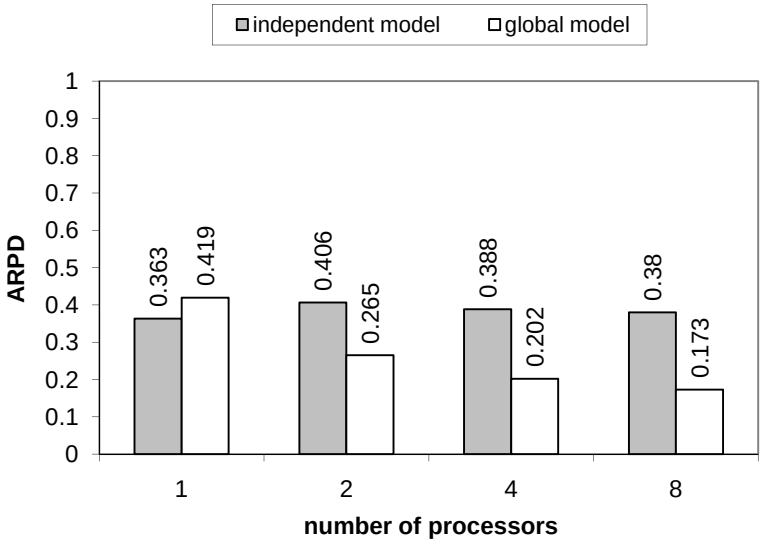


Fig. 12.5. APRD of the global and independent scatter search ($iter = 16{,}000$) for 50 instances from [202].

ible for the large instance of the flow shop problem – for $n = 50$, $m = 5, 10$. The APRD is about 50% better for the 8-processor implementation compared to 1-processor version, for the same number of iterations calculated as a sum of iterations executed on all processors.

12.4. Speedup anomalies

Since we do not know the best algorithm for the flow shop instances, it is impossible to use the *strong speedup* definition, i.e., comparing the parallel runtime against the best-so-far sequential algorithm. Therefore, we have to use the weak definition of speedup. We cannot compute speedup against a sequential scatter search algorithm, because we compare different algorithms. Hence, we turn to compare the same parallel scatter search algorithm on 1 versus p processors. Such a speedup is known as the orthodox speedup (see Alba [7]).

Several authors reported superlinear speedup [9, 84] with reference to the following sources:

- implementation source – the sequential algorithm is inefficient, i.e., it uses data structures which can be managed faster by the parallel algorithm,
- numerical source – the parallel algorithm finds a good solution more quickly because it changes the order in which solution space is searched compared

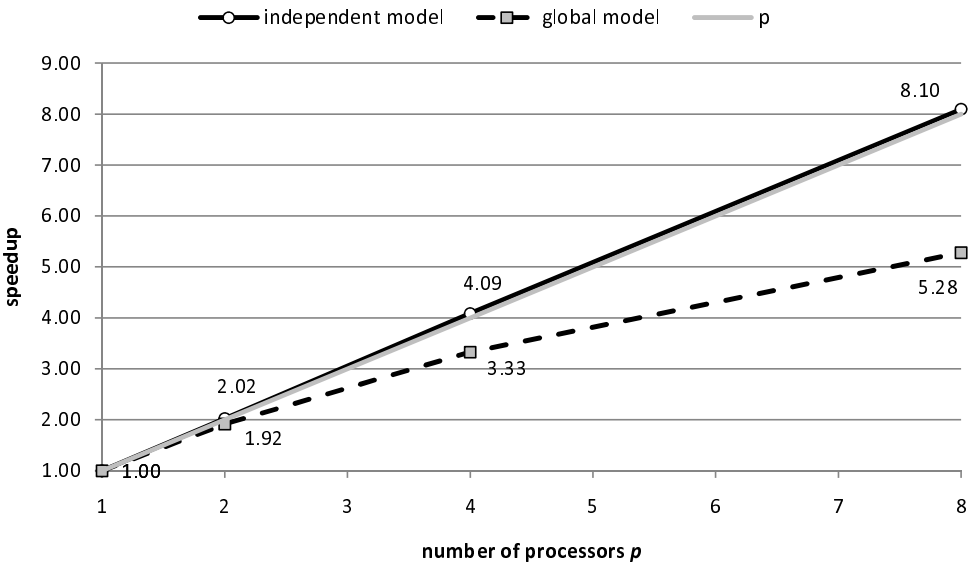


Fig. 12.6. Orthodox speedup of the parallel scatter search, $iter = 16{,}000$.

to sequential algorithm,

- physical source – the parallel algorithm is characterized by more than a simple increase in the computational power of CPUs, i.e., other resources such as the total size of a fast cache memory.

In this chapter, we observe a situation where the work performed by parallel and sequential algorithms is different.

Flow shop with $C_{\max}$ criterion. As the time-consumption on all processors is a little bit longer than the time of the sequential version, we can say that the speedup of this version of an algorithm is almost-linear (see Tables A.11 and A.12). For the 4 and 8-processor implementation of the global model and for 2, 4 and 8-processor implementation of the independent model the average results of APRD are better than APRD of the 1-processor versions, but the time-consumption on all the processors (t_{cpu}) is *shorter*. So these algorithms obtain better results with a smaller cost of computations – the orthodox speedup is superlinear. This effect can be observed in Figure 12.6.

Flow shop with C_{sum} criterion. Also here the orthodox superlinear speedup effect has been observed for the 8 and 16-processor implementation of the global model of parallel scatter search (see Tables A.13 and A.14). The total time-consumption of this implementation for all 50 instances (74:38:51 and 73:13:35, hours:minutes:seconds) was smaller than the total time of sequential algorithm execution (75:20:42). Such a situation takes place only for the global model:

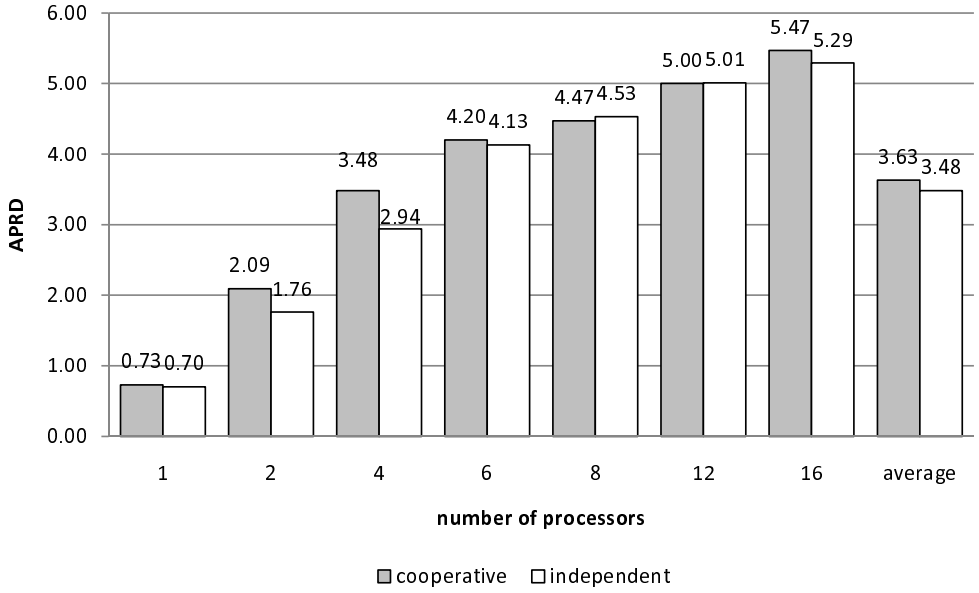


Fig. 12.7. Orthodox speedup of the parallel scatter search, $iter = 1{,}600$.

independent searches are not so effective, considering both results (APRD) and speedup (see Figure 12.6).

The superlinear speedup anomaly obtained here has the numerical source and it can be understood as the situation where the sequential algorithm may have to search a large portion of solutions before finding a good one. Parallel algorithm may find the solution of similar quality more quickly due to the change in the order in which the space is searched. This situation can be interpreted in terms of diversification versus intensification of the search in the solution space – a parallel algorithm can achieve better solutions faster than a sequential algorithm as a result of searching process diversification in the first phase of the algorithm work (due to the multiple-walk strategy) and intensification in the second phase after finding a 'good' region by one of the multiple walking parallel searching threads.

12.5. Remarks and conclusions

The methodology of solving the flow shop problem by using scatter search algorithm has been described here, however the job scheduling problem considered can be understood only as a case study – the proposed parallelization methodology is a general approach, which increases the quality of solutions obtained maintaining comparable costs of computations. Superlinear speedup is observed in cooperative (global) model of parallelism. This anomaly has a numerical source – the

parallel algorithm by using cooperation (data broadcasting) generates a shorter path (in the sense of the number of visited solutions needed to obtain a result with comparable quality) in the solution space than the sequential algorithm. The parallel scatter search outline can be easily adapted to solve other NP-hard problems with permutational solution representation, such as traveling salesman problem (TSP), quadratic assignment problem (QAP). The most efficient results can be obtained for scheduling problems with block properties (such as job shop problem with makespan or single machine total weighted tardiness problem, see Wodecki [268, 270]), in which reduced neighborhood can be used inside the path-relinking procedure.

Chapter 13

Parallel genetic approach

A multiple-walk parallelization of the island model based genetic algorithm is proposed in this chapter. A multi-step crossover fusion operator, based on the local search procedure, is used for inter-island communication. We take under consideration the permutation flow shop scheduling problem with C_{sum} criterion, indicated by the $F||C_{\text{sum}}$.

13.1. Parallel genetic algorithm

There are three basic types of parallelization strategies which can be applied to the genetic algorithm: global, diffusion model and island model (migration model). Algorithms based on the island model divide the population into a few subpopulations. Each of them is assigned to a different processor which performs a sequential genetic algorithm based on its own subpopulation. The crossover involves only individuals within the same population. Occasionally, the processor exchanges individuals through a migration operator. The main determinants of this model are:

- size of subpopulations,
- topology of connection network,
- number of individuals to be exchanged,
- frequency of exchanging.

The island model is characterized by a significant reduction of the communication time, compared to previous models. Shared memory is not required, so this model is more flexible, too. Bubak and Sowa [68] developed implementation of the parallel genetic algorithm for the TSP problem using the island model.

Below a parallel genetic algorithm is proposed. This algorithm is based on the hybrid island model of parallelism, with inter-island communication. In the standard island model of GA, individuals are just copied from one island to another. Here we propose an approach in which a robust genetic operator Multi-Step Crossover Fusion (MSXF) is used as a local search method making a solution path from one solution taken from an island to a second solution taken from another island (used as a reference point). MSXF has been originally described by Reeves and Yamada [215] as a normal genetic operator. We propose to use it as a communication step of the parallel GA.

The neighborhood $\mathcal{N}(\pi)$ of the permutation (individual) π in the local search communication procedure MSXF is defined as a set of new permutations that can be reached from π by exactly one adjacent pairwise exchange operator which exchanges the positions of two adjacent jobs of a problem solution connected with permutation π. The distance measure $d(\pi, \sigma)$ is defined as a number of adjacent pairwise exchanges needed to transform permutation π into permutation σ. Such a measure is known as Kendall's τ measure. An outline of the MSXF procedure was presented in Figure 12.2, Section 12.1.1.

In our implementation, MSXF is an inter-subpopulation crossover operator which constructs a new individual using the best individuals of different subpopulations connected with different processors. The condition of termination consisted in exceeding 100 iterations by the MSXF function. Frequency of communication between processors (migration and MSXF operator) is very important for the parallel genetic algorithm performance. This must not take place very often because of a long time of communication between processors. In this implementation the processor gets new individuals quite rarely, every $R = 20$ (MSXF operator) or every $S = 35$ (migration) iterations. An outline of the whole parallel genetic algorithm is presented in Figure 13.1.

13.2. Computational experiments

The algorithm was implemented in the Ada95 language and run on the SGI Altix 3700 Bx2 supercomputer installed in Wrocław Centre of Networking and Supercomputing [266] under the Novell SUSE Linux Enterprise Server operating system. Tests were based on 50 instances with $100, \ldots, 500$ operations ($n \times m = 20 \times 5, 20 \times 10, 20 \times 20, 50 \times 5, 50 \times 10$) due to Taillard [243], taken from the OR-Library [202]. The results were compared to the best known ones taken from [215]. Each instance of the test problems was executed six times, and the average result was used for comparison. The standard deviation of results was computed, too, as a measure of algorithm stability.

```
Algorithm 19.  Parallel genetic algorithm
parfor j = 1, 2, ..., p   { p is number of processors }
  i ← 0;
  P_j ← random subpopulation connected with processor j;
  p_j ← number of individuals in j subpopulation;
  repeat
    Selection(P_j, P'_j);
    Crossover(P'_j, P''_j);
    Mutation(P''_j);
    if (k mod R = 0) then {every R iteration}
      r ← random(1, p);
      MSXF(P'_j(1), P_r(1));
    end if;
    P_j ← P''_j;
    if there is no improvement of the average  C_sum then
      {Partial restart}
      r ← random(1,p);
      Remove α = 90 percentage of individuals in subpopulation P_j.;
      Replenish P_j by random individuals;
    end if;
    if (k mod S = 0) then {Migration}
      r ← random(1,p);
      Remove β = 20 percentage of individuals in subpopulation P_j;
      Replenish P_j by the best individuals from subpopulation P_r
      taken from processor r;
    end if;
  until Stop_ Condition;
end parfor
```

Fig. 13.1. Outline of the parallel genetic algorithm.

Firstly, we made efficiency tests of the classical genetic operators (seek Goldberg [117]) for our flow shop problem on the sequential genetic algorithm. Next, we chose the PMX, CX and SX crossover operator and the mutation operator I (random adjacent pairwise exchange) for further research. After having chosen the operators, we implemented the parallel genetic algorithm. The chosen model of parallel computing was the MIMD machine of processors without shared memory – with the time of communication between processors much longer than the time of communication inside the process which is being executed on one processor. The implementation was based on the island model of the parallel genetic

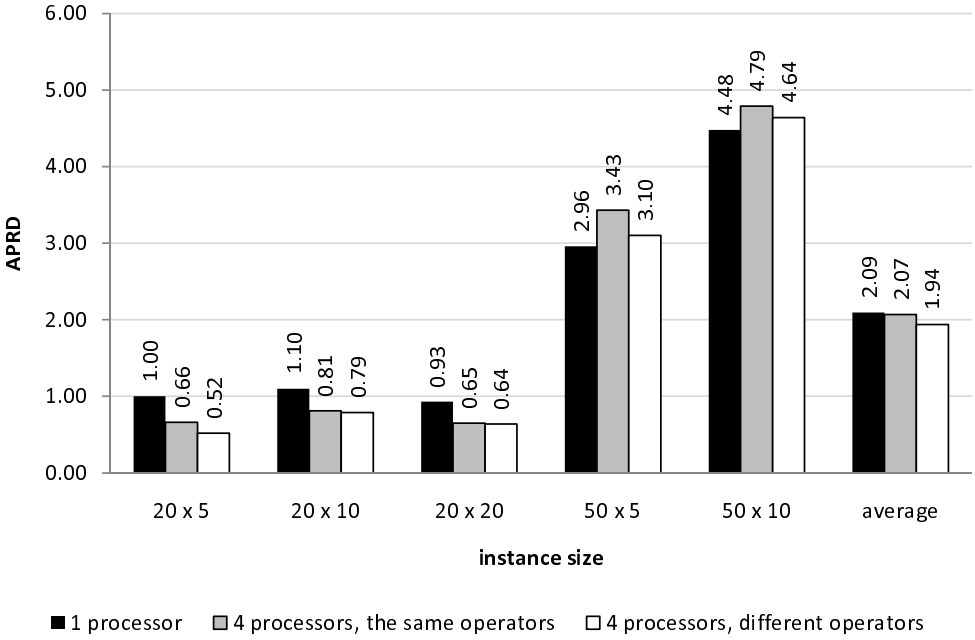


Fig. 13.2. A comparison between sequential and parallel cooperative genetic algorithms.

algorithm with one central processor and slave processors. The central processor mediated in communication and stored data of the best individuals. Slave processors executed their own genetic algorithms based on subpopulations of the main population. Co-operation was based on migration between 'islands' and execution of the MSXF operator with parents taken from the best individuals of different subpopulations (processors).

We tested the efficiency of the parallel algorithm which was activated with a combination of three strategies: with the same or different start subpopulations, as independent or cooperative search threads and with the same or different genetic operators. The number of iterations was permanently set to 1,000. Results of tests for different start subpopulations for every processor are shown in Table A.21, Appendix A (supplementary tables). The results of the computations for the strategy of the same start subpopulations were similar, but slightly worse. A comparison between sequential and the most efficient parallel genetic algorithm (cooperative) is also shown in Figure 13.2. A four-processor implementation of parallel GA with the same and different genetic operators kept by each island was chosen for comparison. The best results (average) were obtained by the version with different genetic operators executed on each island.

As it turned out, the strategy of starting the computation from different subpopulations on each processor with different crossover operators and co-operation, was significantly better than others. The improvement of the distance to reference solutions was at the level of 7%, compared to the sequential algorithm, with the same number of iterations equal 1,000 for the sequential algorithm and 250 for the 4-processor parallel algorithm. The computing time amounting to a few seconds up to a few dozen seconds, depends on the size of the problem instance. Moreover, the parallel algorithm has more stable results – the standard deviation of the results was on average equal to 0.12% for the best parallel algorithm, compared to 0.20% for the sequential algorithm – so the improvement of the standard deviation was at the level of 40% in relation to the sequential algorithm.

13.3. Remarks and conclusions

We discussed an approach to the permutation flow shop scheduling based on the parallel asynchronous genetic algorithm. The advantage is especially visible for large problems. As compared to the sequential algorithm, parallelization increases the quality of solutions obtained. The idea of the best individual migration and the inter-subpopulation operator was used. Computer experiments show that the parallel algorithm is considerably more efficient in relation to sequential algorithm. Results of tests (after a small number of iterations) are insignificantly different from those best known. An extension of this approach can add to the algorithm more elements of coevolutionary schemas, e.g. predators (predator-prey model), food, etc., which will cause further improvement of the parallel algorithm efficiency.

Chapter 14

Parallel hybrid approach

In this chapter, we propose the two new double-level metaheuristic optimization algorithms applied to solve the flexible job shop problem (FJSP) with makespan criterion, defined in Section 3.6. Algorithms proposed here include two major modules: the machine selection module to be executed sequentially, and the operation scheduling module executed in parallel. On each level a metaheuristic algorithm is used, therefore we propose to call this method Meta^{2}Heuristic. We carry out computational experiment using Graphics Processing Units (GPU).

14.1. Hybrid metaheuristics

The hybrid approach to solving difficult optimization problems by using several metaheuristics simultaneously makes it possible to use all of them. Talbi [247] provides a systematic characterization of parallel hybrid metaheuristics, which is visualized in Figure 14.1.

The upper part of the figure presents the hierarchical structure of the hybridization. In *high-level* hybrid algorithms the different metaheuristics are self-contained – there is no direct relationship to the internal workings of metaheuristics. The *low-level* hybridization addresses the functional composition of a single optimization method – a given function of metaheuristic is replaced by another metaheuristic. A *teamwork* hybridization represents cooperative models of optimization in which many parallel agents cooperate and each agent makes a search in its own part of the solution space. On the other hand, in *relay* hybrids, a number of metaheuristics are applied one after another; each one uses the output of the previous one as its own input, as in a pipeline.

The lower part of Figure 14.1 (so-called flat part) specifies the features of algorithms involved in the hybrid. In *homogeneous* hybrids all the combined algorithms use the same metaheuristic methods. On the contrary, in *heterogeneous*

algorithms, different metaheuristics are used. In *global* hybrids, all the algorithms search in the whole solution space – the goal is to explore the space more thoroughly. All the algorithms solve the whole optimization problem. On the other hand, in *partial* hybrids, the problem considered is decomposed into subproblems; each one having its own solution space. Each algorithm is dedicated to search in one of these subspaces. Subproblems are linked with each other involving constraints between optima which are found by each algorithm. Algorithms establish communication to respect these constraints and create a solution of the main problem. In *general* hybrids, all algorithms solve the same target optimization problem. *Specialist* hybrids combine algorithms which solve different problems, i.e., by solving another optimization problem into which the main problem is transformed.

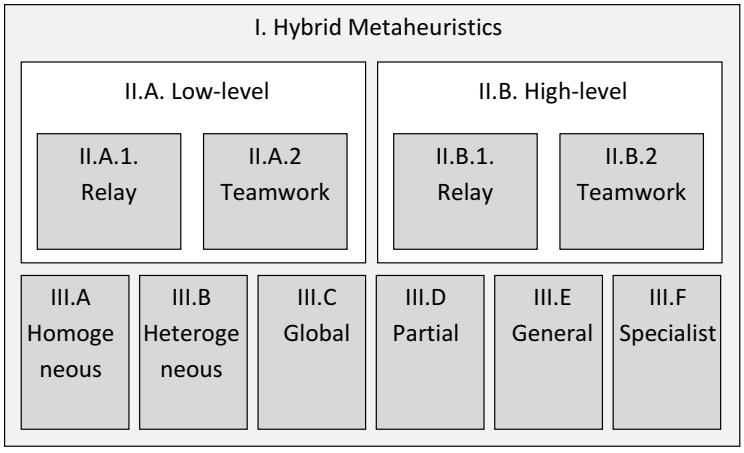


Fig. 14.1. Classification of hybrid metaheuristics proposed by Talbi [247].

To conduct a survey of parallel hybrid metaheuristics application we can cite Bożejko and Makuchowski, who proposed a hybrid metaheuristic solving no-wait job shop problem [29]. Malek et al. [177] proposed a parallel hybrid metaheuristic based on combined simulated annealing and tabu search approaches applied to solve the traveling salesman problem (TSP). Bożejko and Wodecki described a hybrid parallel evolutionary algorithm for the traveling salesman problem (TSP, [44, 55]), for the quadratic assignment problem (QAP, [47]), for the single machine total tardiness problem ([51, 56]) and for the flow shop scheduling problem ([56]). Porto and Ribeiro [209] presented parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. Rogalska, Bożejko and Hejducki [225, 226] proposed a hybrid population-based method in the application to the time/cost scheduling problems. Alba et al. [10] presented library skeletons using hybrid approaches for the resource allocation problems.

14.2. Algorithms proposed

The proposed here flexible job shop problem solving algorithms include two major modules: the machine selection module and the operation scheduling module.

- *Machine selection module.* This module is based on the tabu search (1st approach) and the population-based metaheuristic (2nd approach) methods and it works sequentially. It helps an operation to select one of the parallel machines from the set of machine types to process it.

- *Operation scheduling module.* This module is used to schedule the sequence and the timing of all operations assigned to each machine from the center. It has to solve classic job shop problems after having assigned operations to machines. Two approaches: constructive INSA [195] and TSAB [195] (tabu search) were used on this level.

On each level a metaheuristic algorithm is used, so we call this method Meta^{2}Heuristic (*meta-square-heuristic*). Both algorithms proposed in this chapter belong to the *high-level teamwork general homogeneous* hybrid metaheuristics (according to the taxonomy from Section 14.1). Metaheuristics connected with each module are executed one after another, acting in a pipeline fashion. The algorithms proposed belong to the *partial hybrids*, because the problem in order to be solved is decomposed into subproblems connected with machine workloads.

14.2.1. Parallel Tabu Search Based Meta^{2}Heuristic

The tabu search method was used here as a machine selection module. The algorithm operates on solutions which constitute job-to-machine assignments. The general idea of the tabu search method applied for scheduling problems can be found in [119] and [195]. The tabu list T stores pairs (v, k) where v is the position in the assignment vector and k is the machine to which v is assigned before the move. The first assignment $\mathcal{Q}^0$ is generated by the search for the global minimum in the processing time table taken from [207]. An outline of the double-layer metaheuristic algorithm including both machine selection module and operation scheduling module is presented in Figure 14.2.

In the second step of the algorithm a neighborhood $\mathcal{N}(\mathcal{Q})$ is divided into disjoint sets

$$\bigcup_{i=1}^{k} \mathcal{N}_i(\mathcal{Q}) = \mathcal{N}(\mathcal{Q}), \qquad \bigcap_{i=1}^{k} \mathcal{N}_i(\mathcal{Q}) = \emptyset. \tag{14.1}$$

For each group k values of the makespan are calculated using p GPU processors. The number of processors used in the third step depends on the neighborhood size.

Algorithm 20. Parallel Tabu Search Based Meta^{2}Heuristic (TSBM^{2}h)

$\Theta^* = (\mathcal{Q}^*, \pi(\mathcal{Q}^*))$ – the best known solution, where
$\mathcal{Q}^*$ – the best known assignment and
$\pi(\mathcal{Q}^*)$ – the operation sequence corresponding to $\mathcal{Q}^*$;
Step 0: Find the starting solution $\Theta^0 = (\mathcal{Q}^0, \pi(\mathcal{Q}^0))$; $\Theta^* \leftarrow \Theta^0$;
$\Theta = (\mathcal{Q}, \pi(\mathcal{Q}))$ – current solution; $\Theta \leftarrow \Theta^0$;
Step 1: Generate the neighborhood $\mathcal{N}(\mathcal{Q})$ of the assignment $\mathcal{Q}$.
Exclude from $\mathcal{N}(\mathcal{Q})$ elements from tabu list T;
Step 2: Divide $\mathcal{N}(\mathcal{Q})$ into $k = \left\lceil \frac{|\mathcal{N}(\mathcal{Q})|}{p} \right\rceil$ groups;
Each group consists of at most p elements;
Step 3: For each group k find (using p processors)
an operation sequence $\pi(\mathcal{X})$
corresponding to the assignment $\mathcal{X} \in \mathcal{N}(\mathcal{Q})$ and value
of the makespan $C_{\max}(\mathcal{X}, \pi(\mathcal{X}))$;
Step 4: Find an assignment $Z \in \mathcal{N}(\mathcal{Q})$ such that
$$C_{\max}(\mathcal{Z}, \pi(\mathcal{Z})) = min\{C_{\max}(\mathcal{X}, \pi(\mathcal{X})) : \mathcal{X} \in \mathcal{N}(\mathcal{Q})\};$$
Step 5: **if** $C_{\max}(\mathcal{Z}, \pi(\mathcal{Z})) < C_{\max}(\mathcal{Q}^*, \pi(\mathcal{Q}^*))$ **then**
$\pi(\mathcal{Q}^*) \leftarrow \pi(\mathcal{Z})$;
$\mathcal{Q}^* \leftarrow \mathcal{Z}$;
Include Z to the list T;
$\mathcal{Q} \leftarrow \mathcal{Z}$;
$\pi(\mathcal{Q}) \leftarrow \pi(\mathcal{Z})$;
Step 6: **if** (*Stop condition* is true) **then** Stop;
else go to Step 1;

Fig. 14.2. Outline of the Parallel Tabu Search Based Meta^{2}Heuristic.

In Step 3 the value of makespan corresponding to the assignment is calculated by means of INSA or TSAB algorithms. A general scheme of the TSBM^{2}H execution on GPU for the CUDA programming environment is presented in Figure 14.3 as a case of heterogeneous programming model (i.e., using both CPU and GPUs).

14.2.2. Parallel Population-Based Meta^{2}Heuristic

The basic idea of this approach is to start with an initial population (any subset of the solution space) – job-to-machine assignments. Next, for each element of the population, a local optimization algorithm is applied to determine a local minimum. In this way we obtain a set of solutions – local minima. If there is an element which is in the same position in several solutions, then it is fixed in

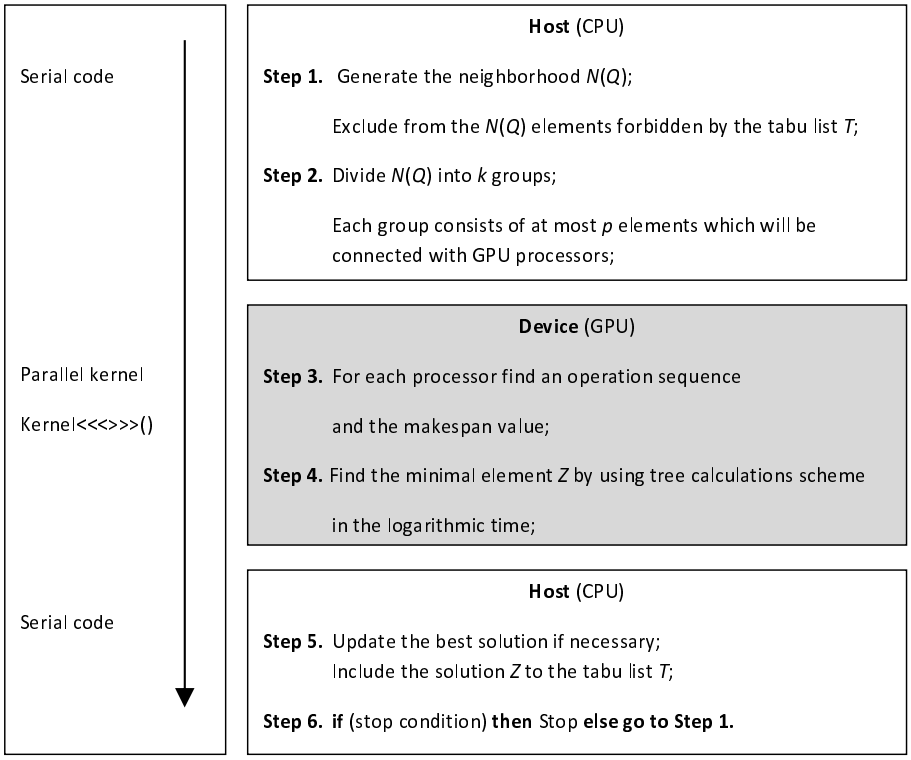


Fig. 14.3. General scheme of the TSBM^{2}H execution on CPU and GPU for the CUDA environment.

this position in the solution, and other positions and elements of solutions are still free. A new population (a set of assignments) is generated by drawing free elements in free positions (because there are fixed elements in fixed positions). After having determined a set of local minima (for the new population) we can increase the number of fixed elements. To prevent the algorithm from finishing its work after executing some number of iterations (when all positions are fixed and there is nothing left to draw), in each iteration 'the oldest' fixed elements are set free.

The method mentioned above was proposed in paper [27] for the permutational scheduling problems. Here we have adopted this approach to flexible job shop scheduling problem, where solution is a pair consisting of a job-to-machine assignment and a sequence of job permutations on each machine. An outline of the proposed approach is presented in Figure 14.4.

In the population based algorithm the initial population is generated randomly. The size of generated population equals $pop\_size = 100$. In the first step

Algorithm 21. Parallel Population-Based Meta^{2}Heuristic (PBM^{2}H)

$\Theta^* = (\mathcal{Q}^*, \pi(\mathcal{Q}^*))$ – the best known solution, where
$\mathcal{Q}^*$ – the best known assignment and
$\pi(\mathcal{Q}^*)$ – the operation sequence corresponding to $\mathcal{Q}^*$;

Step 0: Generate an initial population P_0 of assignments;
$i \leftarrow 1$; $P_i \leftarrow P_0$ – the first generation of the population;
α – a constant threshold, $0 < \alpha < 1$;

Step 1: Divide P_i into $k = \left\lceil \frac{|P_i|}{p} \right\rceil$ groups;
Each group consist of at most p elements;

Step 2: For each element $\mathcal{X}$ of each group find (using p processors)
an operation sequence $\pi(\mathcal{X})$
corresponding to the assignment $\mathcal{X} \in P_i$ and value
of the makespan $C_{\max}(\mathcal{X}, \pi(\mathcal{X}))$ using tabu search algorithm;

Step 3: Find assignment $\mathcal{Z} \in P_i$ such that
$C_{\max}(\mathcal{Z}, \pi(\mathcal{Z})) = min\{C_{\max}(\mathcal{Y}, \pi(\mathcal{Y})) : \mathcal{Y} \in P_i\}$;

Step 4: **if** $C_{\max}(\mathcal{Z}, \pi(\mathcal{Z})) < C_{\max}(\mathcal{Q}^*, \pi(\mathcal{Q}^*))$, **then**
$\pi(\mathcal{Q}^*) = \pi(\mathcal{Z})$;
$\mathcal{Q}^* = \mathcal{Z}$;

Step 5: For each position in population P_i find a number of
assignments υ in which the machine m is in the position l;

Step 6: Fix a position in population P_i for which $\frac{\upsilon}{|P_i|} > \alpha$;

Step 7: Insert randomly drawn free elements (machines)
in free positions;
$P_{i+1} \leftarrow P_i$;

Step 8: **if** (*Stop condition* is true) **then** Stop;
else go to Step 1;

Fig. 14.4. Outline of the Parallel Population-Based Meta^{2}Heuristic.

of algorithm population P_i in the i-th generation is divided into disjoint sets

$$\bigcup_{j=1}^{pop\_size} P_i^j = P_i, \qquad \bigcap_{j=1}^{pop\_size} P_i^j = \emptyset. \tag{14.2}$$

Each element in the randomly generated initial population P_0 starts an assignment for the tabu search algorithm. The tabu search algorithm finds a new assignment corresponding to this assignment operation sequence and the value of makespan. A general scheme of the PBM^{2}H execution on GPU for the CUDA programming environment is shown in Figure 14.5.

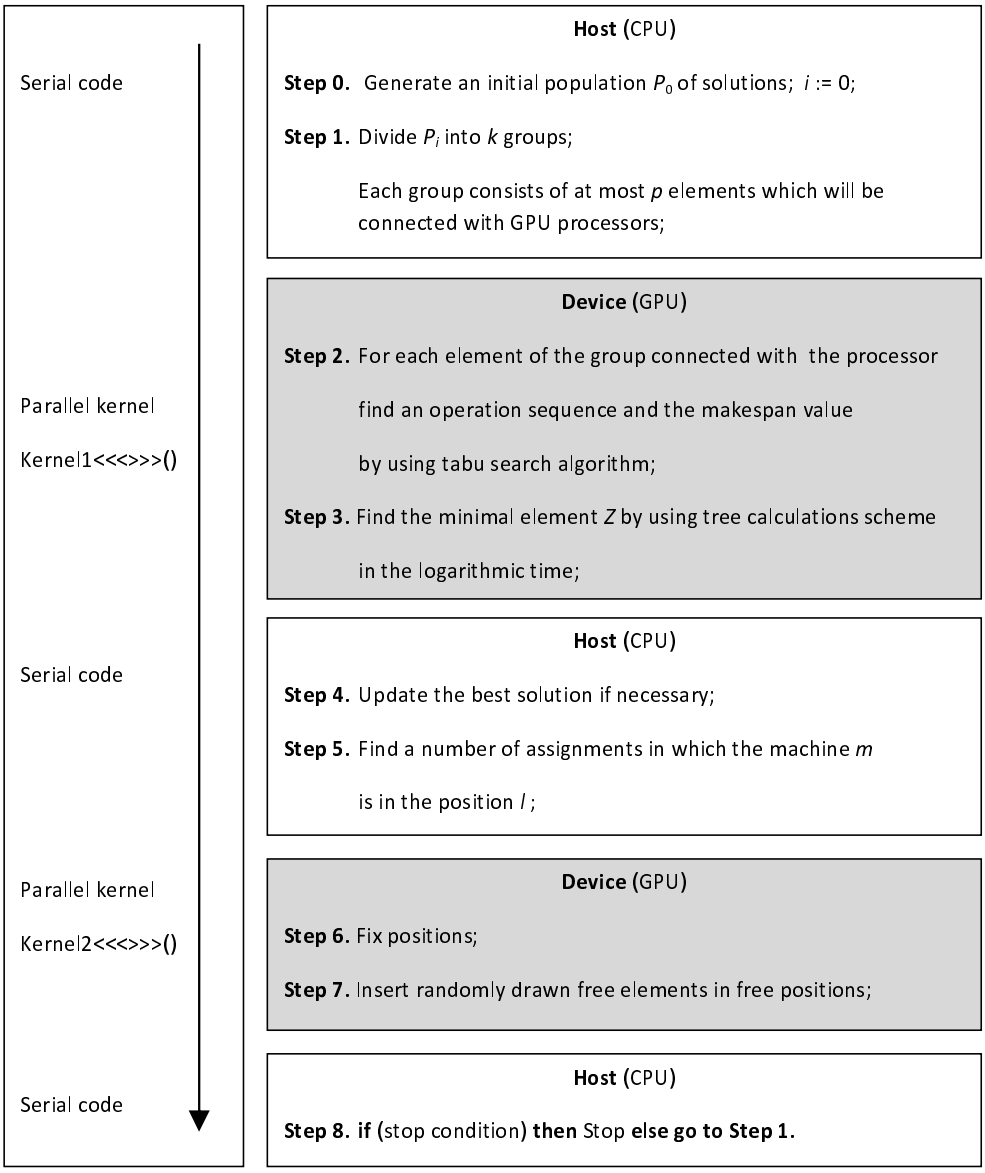


Fig. 14.5. General scheme of the PBM^{2}H execution on CPU and GPU for the CUDA environment.

The parallel population-based algorithm and the parallel tabu search algorithm were coded in C++ language with MPI library and tested on the cluster in the Wrocław Center of Networking and Supercomputing. Algorithms were executed under the OpenPBS batching system.

14.3. Computational results

Parallel Meta^{2}Heuristics (TSBM^{2}H and PBM^{2}H) for the flexible job shop problem were coded in C (CUDA) for GPU and were ran on the Tesla C870 GPU (512 GFLOPS) with 128 streaming processor core and tested on the benchmark problems from the literature. The GPU was installed on the processors with 1 MB cache memory and 8 GB RAM working under 64-bit Linux Debian 5.0 operating system. We compare our results with those obtained by other authors:

1. the set of 10 problem instances taken from Brandimarte [67],
2. the set of 21 problem instances taken from Barnes and Chambers [21],
3. the set of 15 problem instances taken from Hurink et al. [139].

The first phase of computational experiments was devoted to determination of parallelization efficiency by estimating experimental speedup values. The sequential algorithm using one GPU processor was coded with the aim of determining the speedup value of the parallel algorithm. Such an approach is called orthodox speedup (see Alba [7]) and it compares the execution times of algorithms on machines with the same processors (1 versus p processors). Tables 14.1 and 14.2 show computational times for the sequential and the parallel algorithm as well as speedup values. Flex. denotes the average number of equivalent machines per operation. The obtained orthodox speedup measure value is visualized in Figure 14.6. As we can notice the highest speedup values were obtained for the problem instances with a bigger number of both jobs n and operations o. In this phase a simple INSA algorithm was applied in the operation scheduling module of the parallel Meta^{2}Heuristic.

For test instances of Barnes and Chambers [21] an average number of equivalent machines per operation is between 1.07 and 1.30; for test instances of Hurink et al. [139] it equals 2 for each instance from the set. Test instances with greater number of equivalent machines per operation are more difficult to solve because there are more possible assignments of operations to machines. The PBM^{2}h gives better result for instances of Hurink et al. (Table 14.3) because the size of the search space in this algorithm is bigger than in TSBM^{2}H. Increasing the exploration measure (i.e., executing local optimization procedure for a longer time) gives better results for PBM^{2}H in comparison with TSBM^{2}H. The PBM^{2}H gives

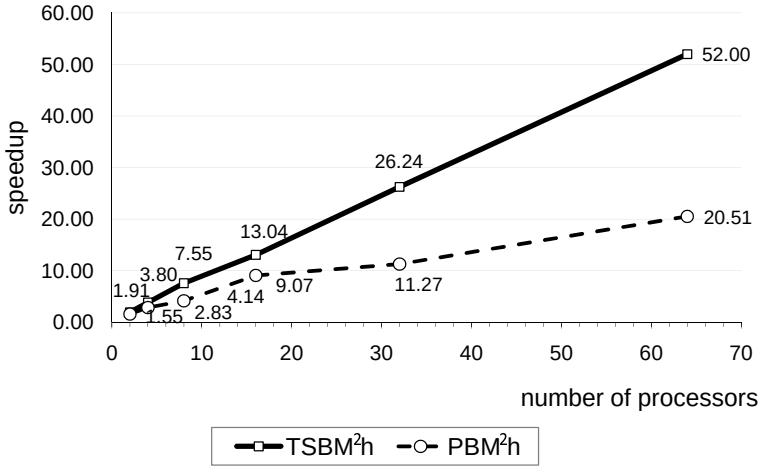


Fig. 14.6. Comparison of the parallel tabu search TSBM^{2}H and population-based PBM^{2}H algorithms speedups.

Table 14.1. Experimental results of the TSBM^{2}H for Brandimarte [67] tests.

Problem	$n \times m$	Flex.	o	t_s [s]	t_p [s]	Speedup s *
Mk01	10×6	2.09	55	133.61	10.79	12.38
Mk02	10×6	4.10	58	218.02	10.55	20.67
Mk03	15×8	3.01	150	6495.35	136.19	47.69
Mk04	15×8	1.91	90	620.69	29.59	20.98
Mk05	15×4	1.71	106	1449.80	74.55	19.45
Mk06	10×15	3.27	150	8094.39	147.83	54.75
Mk07	20×5	2.83	100	1939.33	57.92	33.48
Mk08	20×10	1.43	225	8950.91	643.39	13.91
Mk09	20×10	2.53	240	24586.00	641.88	38.30
Mk10	20×15	2.98	240	31990.55	593.49	53.90

* The INSA algorithm was used in the operation scheduling module.

better results than TSBM^{2}H but computation time is longer. A number of equivalent machines per operation can be used for determination of parameters – if a number of equivalent machines per operation is small in a particular test instance, there should be executed more iterations in the operation scheduling module.

The second phase of the tests refers to obtaining as good results of the cost function as possible. In this phase a specialized TSAB algorithm of Nowicki and Smutnicki [195] was used in the operation scheduling module of the parallel Meta^{2}Heuristic. Despite of being more time-consuming the quality of the results obtained is much better than in the case of using INSA. The results were also

Table 14.2. Experimental results of the TSBM^{2}H for Barnes and Chambers [21] instances.

Problem	$n \times m$	Flex.	o	t_s [s]	t_p [s]	Speedup s *
mt10c1	10×11	1.10	100	69.27	28.18	2.46
mt10cc	10×12	1.20	100	130.47	27.28	4.78
mt10x	10×11	1.10	100	69.26	28.04	2.47
mt10xx	10×12	1.20	100	134.78	28.01	4.81
mt10xxx	10×13	1.30	100	133.99	28.03	4.78
mt10xy	10×12	1.20	100	130.93	27.43	4.77
mt10xyz	10×13	1.30	100	187.19	26.37	7.10
setb4c9	15×11	1.10	150	333.71	96.52	3.46
setb4cc	15×12	1.20	150	631.39	92.71	6.81
setb4x	15×11	1.10	150	332.64	96.90	3.43
setb4xx	15×12	1.20	150	654.09	95.27	6.87
setb4xxx	15×13	1.30	150	648.42	94.43	6.87
setb4xy	15×12	1.20	150	632.55	92.95	6.81
setb4xyz	15×13	1.30	150	896.66	88.54	10.13
seti5c12	15×16	1.07	225	747.64	340.46	2.20
seti5cc	15×17	1.13	225	1458.94	335.48	4.35
seti5x	15×16	1.07	225	753.07	342.35	2.20
seti5xx	15×17	1.13	225	1493.45	341.35	3.38
seti5xxx	15×18	1.20	225	1481.39	339.62	4.36
seti5xy	15×17	1.13	225	1459.27	335.42	4.35
seti5xyz	15×18	1.20	225	2123.35	325.94	6.51

* The INSA algorithm was used in the operation scheduling module.

compared to other recent approaches proposed in the literature for the flexible job shop problem. The proposed parallel TSBM^{2}H algorithm managed to obtain the average relative percentage deviation from the best known solution of the Barnes and Chambers' instances on the level of 0.014% versus 0.036% of the MG [180] algorithm of Mastrolilli and Gambardella and 0.106% of the hGA [111] algorithm of Gao et al.

14.4. Remarks and conclusions

We have proposed a new approach to the scheduling problems with parallel machines, where the assignment of operations to machines defines a classical problem without parallel machines. We propose double-level parallel metaheuristic, where each solution of the higher level, i.e., job-to-machine assignment, defines an NP-

Table 14.3. Comparison of the results obtained by Mastrolilli and Gambardella [180], TSBM^{2}H and PBM^{2}H algorithms.

Problem*	$n \times m$	(LB,UB)	MG [180]	PBM^{2}H	TSBM^{2}H
la01	10×5	(570,574)	571	572	574
la02	10×5	(529,532)	530	530	532
la03	10×5	(477,479)	478	478	482
la04	10×5	(502,504)	502	502	509
la05	10×5	(457,458)	457	458	462
la06	15×5	(799,800)	799	799	801
la07	15×5	(749,750)	750	750	751
la08	15×5	(765,767)	765	765	767
la09	15×5	(853,854)	853	853	856
la10	15×5	(804,805)	804	805	807
la11	20×5	(1071,1072)	1071	1071	1072
la12	20×5	936	936	936	937
la13	20×5	1038	1038	1038	1039
la14	20×5	1070	1070	1070	1071
la15	20×5	(1089,1090)	1090	1090	1090

* For test instances taken from Hurink et al. [139].

hard job shop problem, which we solve by the second metaheuristic – we call such an approach Meta^{2}Heuristic. On the Machine Selection Module (higher level), we apply two metaheuristics: the tabu search and the population-based approach to determine an assignment of operations to machines. The distributed tabu search threads are used as Operations Scheduling Module (lower level). Using the exact algorithms on both levels (e.g. branch and bound) makes it possible to obtain an optimal solution of the problem. It was possible to obtain the new best known solution for Barnes and Chambers' instances [21] (TSAB algorithm was used in the operation scheduling module of the M^{2}H). The new best solutions are presented in Table A.22, Appendix A.

Our proposition of Meta^{2}Heuristic can be placed between standard metaheuristic and hyperheuristic classes in the taxonomy of approximate algorithms. Hyperheuristics present a different approach – they select a metaheuristic to be used in the given optimization problem. Our approach is still a metaheuristic, but more complex – two completely different heuristic approaches have to be chosen on two levels of the scheduling problem considered.

Chapter 15

Application: parallel tabu search approach

The main purpose of this chapter is to present the methodology of the tabu search method parallelization. The parallel algorithm is built for the flow shop scheduling problem. Partitioning a solution (permutation) into blocks (see Section 3.4, *block properties*) enables us to decrease the neighborhood size and its division into separated subsets, which in turn facilitates its independent generation and reviewing. A road building process is analyzed (Section 15.4) as a real-world application. It is also used as a case study (Section 15.5) for the parallel tabu search algorithm application.

15.1. Introduction

Optimization of the job flow process through the system is based on finding some optimal permutation in the set of all permutations of jobs. Many methods of algorithm construction consist in reviewing (directly or indirectly) all or part of the set of feasible solutions. Such a mechanism is based on generating from the current (base) solution another solution, or a set of solutions (so-called neighborhood), from which the best solution is chosen. This solution is the base solution in the next iteration. Such a mechanism can be met (among others) in branch and bound (B&B) method and in many other algorithms which consist in improving the solution, as well as in the approximate algorithms, metaheuristics: tabu search and simulated annealing methods. The quality of these algorithm solutions depends on: the number of iterations, the method of neighborhood description and its reviewing. The time of computations can be shortened by performing them in multiprocessor environment. Unfortunately direct parallelization of the sequential algorithms (for example, by using a parallel compiler) does not result in

satisfactory efficiency and speedup. In this chapter, we propose some additional elements of the parallel local search algorithm. The first one, *representatives approach*, offers the possibility of parallelizing the neighborhood search process. The second one, using *block properties*, makes the whole search process shorter.

15.2. Parallel tabu search method

The tabu search (TS) method is a metaheuristic approach designed to find a near-optimal solution of combinatorial optimization problems. The basic version of TS starts from an initial solution x^0. The elementary step of the method performs, for a given solution x^i, a search through the neighborhood $\mathcal{N}(x^i)$ of x^i. The neighborhood $\mathcal{N}(x^i)$ is defined by a move performed from x^i. The move transforms a solution into another solution. The aim of this elementary search is to find in $\mathcal{N}(x^i)$ a solution x^{i+1} with the lowest cost functions. Then the search repeats from the best solutions found, as a new starting solution, and the process is continued. In order to avoid cycling, becoming trapped to a local optimum, and more general to conduct the search in 'good regions' of the solution space, a memory of the search history is introduced. Among many classes of the memory introduced for tabu search (see Glover [127]), the most frequently used is the short term memory, called the tabu list. This list memorizes, for a chosen span of time, selected attributes of these solutions or moves.

There are two basic types of tabu search parallelization discussed in the literature. The first one, called single-walk, is based on neighborhood decomposition onto concurrent working processors. The solutions obtained are exactly the same as in the sequential algorithm, but computing time is shorter. Aarts and Verhoeven [1] make the distinction between single-step and multiple-step parallelism within this type. In the case of single-step implementations, neighbors are searched and evaluated in parallel after neighborhood partitioning. The algorithm subsequently selects and performs one single move. In multiple-step parallelizations, a sequence of consecutive moves in the neighborhood is made simultaneously.

The second type of parallelization, called multiple-walk type, is based on concurrent working tabu search threads, running on different processors. There are two sub-types of this parallelism: independent search, where there is no communication between threads, and cooperative search, with exchanging e.g. the best known solution found by the thread. Classification of multiple-walk tabu search algorithms was created by Voss in [261]. The first parallel implementations of tabu search based on this type of strategy seem to concern the quadratic assignment problem and job shop scheduling, Taillard [245, 244].

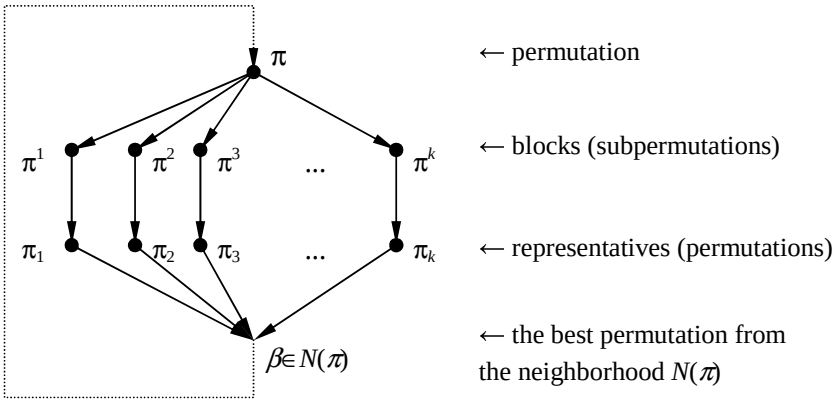


Fig. 15.1. Outline of the PSTS algorithm.

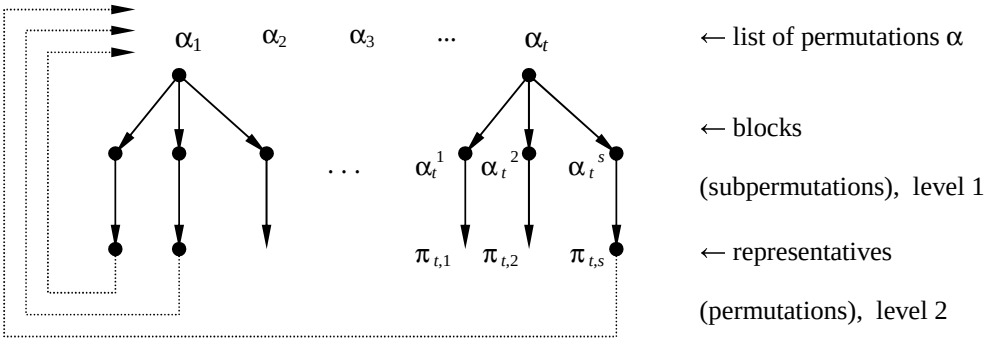


Fig. 15.2. Outline of PATS algorithm.

We propose a hybrid type of tabu search parallelism in this chapter. We use blocks (see Section 3.4.3) to partition the neighborhood of the solution. Such an algorithm PSTS (parallel synchronous tabu search) is the classical parallel single-walk, single-step tabu search (Figure 15.1). The new algorithm PATS (parallel asynchronous tabu search) uses a special list $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_t)$ of current solutions instead of one current solution in classical tabu. The length of a list α is permanently equal to t. This parallel algorithm is based on two-level parallelism. One level is based on concurrently explored solutions from the list α by parallel working processors. These solutions are explored concurrently by block partitioning to find the best representative (solution) of each block, and added to the list α (Figure 15.2); this is the second level of parallelism made by another group of parallel working processors. Of course, we also use a tabu list to prevent generating solutions on the list α serially. The second level of parallelism does not have any influence on results of computations (only speedup), but the first level of parallelism does. Additionally, in most parallel computing systems, we do not

know the order of solutions (representatives) entering list α from processors – so such an algorithm is not deterministic. A parallel tabu search algorithm written for the EREW PRAM model of parallel computations is given in Figure 15.3.

```
Algorithm 22.  Parallel Tabu Search
Master processor
     Commission slave processors on level 1 to determine
          representatives for permutations of list α in parallel;
     Get representatives from slave processors on level 2
          (asynchronously, when they come);
     Reject representatives forbidden by tabu list, unless
          its goal value is less than the best known;
     Add representatives to α list (instead of the worst
          element of α, if the length of list is maximal fixed).
Slave processors on level 1
     Get permutation α_i to explore from the list
          α of the master processor;
     Partition permutation α_i into blocks (compute critical
          path);
     Commission slave processors on level 2 to determine
          representatives for each block of α_i in parallel.
Slave processors on level 2
     Get block α_i^j of permutation α_i
          from the slave processor on level 1;
     Determine representative (permutation π_{t,j})
          of neighborhood generated for this block;
     Send π_{t,j} to master processor.
```

Fig. 15.3. Outline of the parallel tabu search algorithm.

If the length of list $|\alpha|$ equals 1, we obtain synchronous PSTS algorithm out of the foregoing scheme. For $|\alpha| > 1$ we obtain asynchronous PATS algorithm, because slave processors work independently and asynchronously.

15.3. Computational experiments

The proposed algorithm was implemented in Ada95 language and ran on the SGI Altix 3700 Bx2 supercomputer installed in Wrocław Centre of Networking and Supercomputing [266] under the Novell SUSE Linux Enterprise Server operating system. Tests ware based on 70 instances with 100, ..., 1,000 operations ($n \times$

$m = 20 \times 5, 20 \times 10, 20 \times 20, 50 \times 5, 50 \times 10, 50 \times 20, 100 \times 5$) due to Taillard [243], taken from the OR-Library [202]. The results ware compared to those of Taillard (optimal or the best known), taken from this library. Starting solutions were taken from quick approximate algorithm NEH (Navaz, Enscore and Ham, [194]). For all the algorithms tested, the number of iterations was computed as a sum of iterations on processors. For example, for 1,000 iterations, a 4-processor implementation makes 250 iterations on each of the 4 processors.

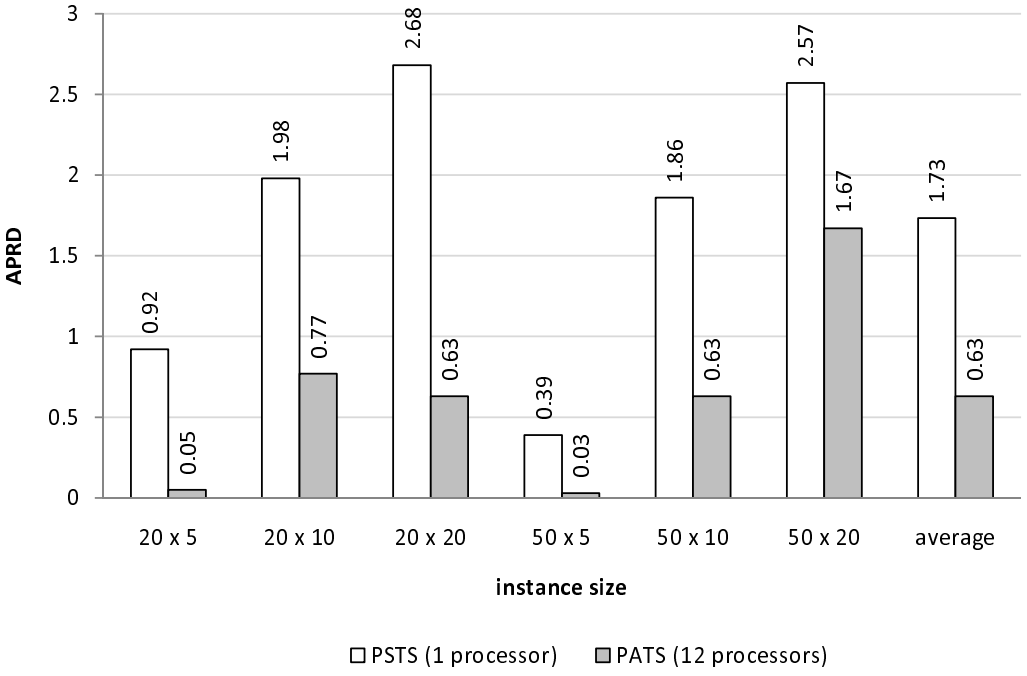


Fig. 15.4. APRD of the sequential (PSTS) and asynchronous parallel (PATS) tabu search algorithm for instances of Taillard [243].

As we can see in Figure 15.4 and in Tables A.19 and A.20, Appendix A, the algorithms produce best results for the large value of quotient n and m ($20 \times 5, 50 \times 5, 100 \times 5$). In such a case the size (length) of blocks is most profitable for sequential and parallel algorithm performance. Besides, for 1,000 iterations, improvement of results for parallel PATS algorithm compared to the PSTS (which has the same results as sequential tabu search) was at the level of 14% for 1-processor implementation (by the advantage of the list α), at the level of 58% for 4 and 8-processor implementations and even at the level of 64% for 12-processor implementation, all algorithms with the same number of iterations (as the sum of iterations on each processor) and comparable costs (the product of the number of processors and computation time).

15.4. Application of the tabu search algorithm – road building

When planning roadworks, e.g. a road repair, the whole project can be divided into working parcels with different sizes whose boundaries are set for instance by crossroads/intersections with existing road paths. The sequence of actions taken by working brigades on parcels will affect the total time of the whole project, the delay time of machines or working brigades. The problem of setting the optimal sequence of works on individual parcels in compliance with the established criterion, for example the minimal time of carrying out a project, the minimal delay time of the working brigades or working costs concerns tasks sequencing. To make a right division of works, it is necessary to determine the kind of works according to the general classification. The general works classification in the road and bridge construction is as follows:

1. preparatory works,
2. earthworks,
3. lands and pavements consolidation,
4. profiled lands and gravel pavements building,
5. reinforcement of soil-surfaced road pavement and subgrade,
6. broken stone pavement building,
7. asphalt concrete pavement building,
8. cement concrete pavement building,
9. repairing, conservation, maintenance and reconstruction of pavement,
10. production, conversion and purification of aggregates including aggregates usage obtained in the process of recycling,
11. loading, unloading and transport works,
12. bridges and culverts building,
13. energy production and transfer,
14. old objects demolition.

Roadwork constitutes a special case of the general works classification mentioned above, namely the national road section, which consists of eight activities. Figure 15.5 presents works scheduling for a single road segment 766 m long. The total time of completing works took 42 workdays. The technological order of works is as follows:

1. earthworks,
2. sand drainage blanket preparation,

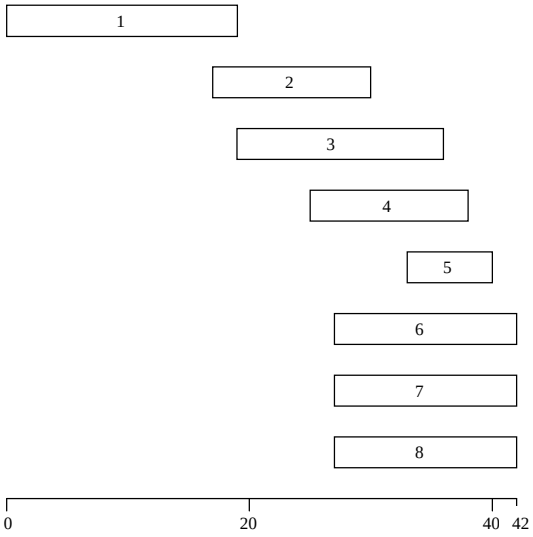


Fig. 15.5. Scheduling example for a 766 m long road segment (in workdays).

3. preparation of broken-stone or macadam foundation,
4. binding course preparation with asphalt medium-grained concrete,
5. surface course preparation with asphalt fine-grained concrete,
6. roadsides preparation with stone dust,
7. drainage process,
8. planting and eventual topsoil removal.

Figure 15.6 presents the road section whose construction is the subject of the analysis. The problem described above leads to the flow shop problem considered with a certain criterion, i.e., $C_{\max}$ (for the minimal time of carrying out a project).

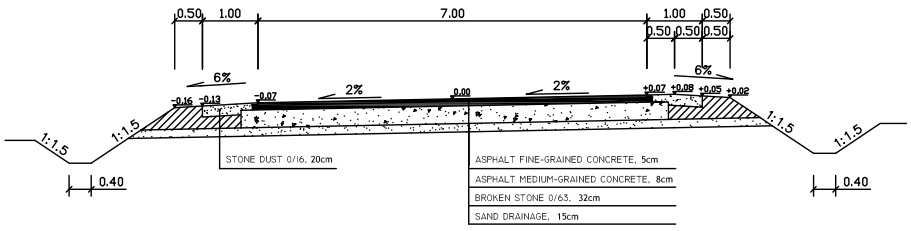


Fig. 15.6. The section of an access road to a dumping ground.

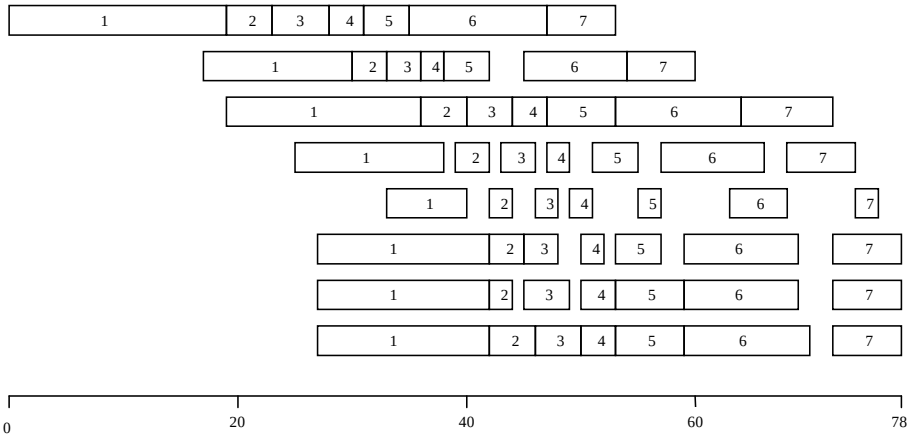


Fig. 15.7. Building schedule for individual road segments for the natural permutation (in workdays).

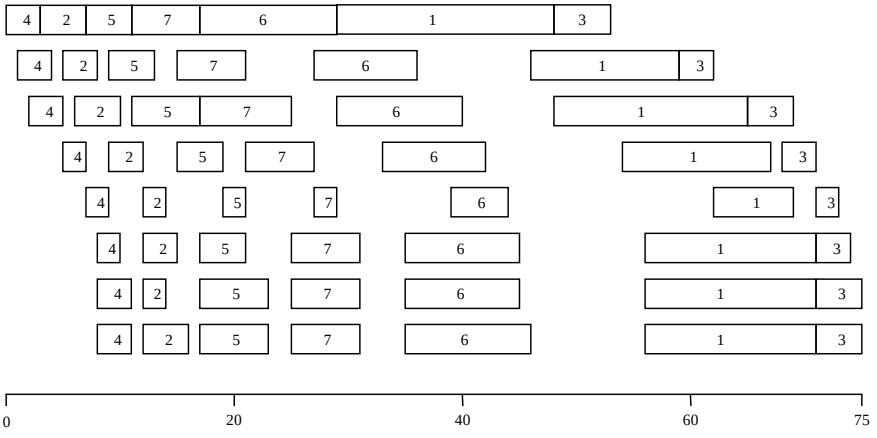


Fig. 15.8. Building schedule for individual road segments for the permutation obtained by the parallel tabu search algorithm (in workdays).

15.5. Case study

The problem presented in this section concerns scheduling of construction projects in which – using the language of automation – jobs should start on the next machine before finishing on the previous one. In a classical permutation flow shop problem each of the jobs should be carried out one after another; moreover the sequence of carrying out jobs on each machine has to be the same; a job must

Table 15.1. Data for the case study. Total times of actions on working segments represented as workdays.

$jobs \rightarrow$	1	2	3	4	5	6	7
length [meters] $\rightarrow$	750	150	175	100	200	500	300
p_{1j}	19	4	5	3	4	12	6
p_{2j}	13	3	3	2	4	9	6
p_{3j}	17	4	4	3	6	11	8
p_{4j}	13	3	3	2	4	9	6
p_{5j}	7	2	2	2	2	5	2
p_{6j}	15	3	3	2	4	19	6
p_{7j}	15	2	4	3	6	10	6
p_{8j}	15	4	4	3	6	11	10
t_{1j}	−2	−2	−2	−2	−2	−2	−2
t_{2j}	−11	−2	−2	−1	−2	−7	−4
t_{3j}	−11	−1	−1	0	−2	−7	−4
t_{4j}	−5	0	0	0	0	−3	0
t_{5j}	−13	−3	−2	−1	−4	−5	−4
t_{6j}	−15	−3	−3	−2	−4	−10	−6
t_{7j}	−15	−2	−4	−3	−6	−10	−6
t_{8j}	0	0	0	0	0	0	0

not start on a next machine until it is finished on the previous one. Optimization consists in determining such a sequence of jobs that will minimize the total time of their execution. To model the phenomenon of jobs 'overlapping', transport times of jobs between machines, which could have negative values, have been used. The problem thus described was defined in Section 3.4.4. Using the real-world data presented above, the data referring to the building of road segments with different length have been generated. In Table 15.1 there have been included data presented as the total times of actions on working segments represented as workdays. The parallel tabu search algorithm considered has been applied to data from Table 15.1. This algorithm operation has resulted in a permutation $\pi = (4, 2, 5, 7, 6, 1, 3)$ for which the goal function value is $C_{\max}(\pi) = 75$ (workdays). In Figure 15.8, there is a building schedule for individual road segments for the obtained permutation π. The goal function value in the case of scheduling for the natural permutation id (Figure 15.7, $C_{\max}(id) = 78$) is bigger than for a scheduling obtained from results of the tabu search algorithm operation (Figure 15.8, $C_{\max}(\pi) = 75$).

15.6. Remarks and conclusions

We discussed a methodology of the flow shop scheduling parallelization based on an asynchronous hybrid version of parallel tabu search method. Parallelism of the algorithm makes the performance much better than the iterative improvement approach. The advantage is especially visible for large problems. The method is based on a two-level approach. On the first level of parallelization subpermutations (blocks) are used for concurrent searching of a neighborhood. On the second level permutations (representatives) are used to choose the best element of a neighborhood in parallel. This method can be easily adapted to solve any scheduling problem with block properties (e.g. single machine, job shop, etc.).

Chapter 16

Final remarks

The book concerns the new methodology of solving scheduling problems in parallel and distributed environments by using multithread approximate methods. Two main approaches, which do not exclude each other, are considered for designing efficient multithread algorithms: single-walk and multiple-walk. The approaches proposed have been created to solve complex discrete optimization problems, and tested using their special case which scheduling problems are. Not only are they classical problems encountered in computer-aided manufacturing and management systems, but also the new, much more complex problems connected with automation in construction, logistics, telecommunication and services development. Very big size of practical instances and exponential solution time of exact algorithms, on the one hand, and multi-core nature of existing hardware, on the other, cause the necessity of employing multithread approximate algorithms ready to be used in multiprocessor environment.

For multithread single-walk parallelism, parallelization derived from a cost function multithread implementation allows us to design cost-optimal algorithms in many cases, especially for job shop scheduling (Chapter 5). It was possible to answer a few interesting questions concerning theory and applicability:

(1) which approaches can be used to design parallel algorithms for scheduling problems, in the context of needs of different local search techniques used in metaheuristics, and

(2) which variants of multithread algorithms are cost-optimal.

Especially interesting is also Chapter 7 in which it was possible to estimate theoretical speedup of the single-walk parallel genetic algorithm based on the master-server approach.

For multithread multiple-walk parallelism, the book presents a new parallel approach based on metaheuristics: tabu search, simulated annealing, genetic

algorithm, scatter search, population-based approaches, path-relinking method, memetic algorithm, designed for solving permutational scheduling problems. In most cases multithread search parallelization increased the quality of solutions obtained keeping comparable costs of computations.

The methodology proposed was successfully applied by the author to solve many scheduling optimization problems (single machine total tardiness problem [51, 53, 54, 60], single machine total weighted tardiness problem with sequence-dependent setups [26, 45], flow shop problem [25, 35, 37, 38, 39, 40, 41, 43, 61, 269], job shop problem [34], flexible job shop problem [28]) as well as other discrete optimization problems such as TSP [36, 27, 44, 55], QAP [47] and real-world problems of automation in construction [225, 226].

The idea of the book was to overview the 'state-of-the-art' in the field of parallel scheduling algorithms. However, some of the special areas can be still researched, especially in the field of the new multithread optimization algorithms for multi-core hardware environments (such as GPGPU and multi-core *Cell* processors jointly developed by Sony Computer Entertainment, Toshiba and IBM). Another challenge is the single-walk parallelization of complex scheduling problems, such as flexible multi-machine manufacturing systems, by proposing new special properties which makes it possible to create efficient parallelization methods. Particulary interesting are theoretical estimations of speedups possible to obtain for a given model of parallelism. This kind of analysis was introduced in Chapters 4, 5 and 6.

16.1. New approaches

The following new approaches have been proposed as regards single-walk parallel algorithms design methodology for job scheduling problems solving:

- the new methods of huge neighborhoods searching in parallel for various single machine scheduling problems (Chapter 4).
- the new methods of single-walk parallelization of the cost function calculation for multi-machine job scheduling problems (Chapter 5),
- the new methods of single-walk parallelization of the workload determination for flexible scheduling problems (Chapter 6).

For multiple-walk optimization algorithms parallelization designed to job scheduling problems solving, the following new techniques have been presented:

- multithread methods of scheduling problems solving methods parallelization were introduced for such local search approaches as tabu search (TS, Chapter 15) and simulated annealing (SA, Chapter 11),

- population-based metaheuristics such as memetic algorithm (Chapter 8), genetic search (Chapter 13), scatter search (Chapter 12) as well as hybrid methods (Chapters 9 and 14) were also parallelized in application to job scheduling problems,
- a parallel branch and bound method has also been parallelized for the single machine total weighted tardiness problem in Chapter 10.

Noteworthy is also Chapter 7 in which there were proved some new properties of the speedup measure behavior for the master-slave model of the parallel genetic algorithm. It has been proven, inter alia, that one can indicate the number of processors which minimizes the parallel running time of the master-slave parallel genetic algorithm based on miscellaneous models of data broadcasting.

16.2. Open problems

Scheduling problems are taken into consideration in this book as a difficult-to-solve subclass of the discrete optimization problems class. Nevertheless, most of the approaches presented here, mainly those of multiple-walk parallelization, can also be applied to any discrete optimization problems, especially with permutational solution representation, such as traveling salesman problem (TSP, see Bożejko and Wodecki [44] – parallel evolutionary algorithm) or quadratic assignment problem (QAP, see Taillard [244] – parallel tabu search, Bożejko and Wodecki [47] – parallel population-based approach). Metaheuristic parallelization methodology remains the same. However, there are open problems which can show the direction of future work.

16.2.1. Continuous optimization

Parallelization of numerical optimization methods in continuous spaces can be achieved in many ways. The simplest one is to implement a single-walk code by evaluation of objective and constraint functions in parallel, however the parallelization method has to be fitted to the specificity of the problem being solved, similarly as in single-walk parallelization of discrete optimization problems. Another approach is a parallel implementation of linear algebra computations, such as solving linear system by the Newton method in parallel, as a key element (i.e., the most computationally complex) of the whole continuous optimization problem. It is also possible to simultaneously explore different regions via multiple starting points, as in parallel multiple-walk local search algorithms. Similarly, multi-directional searches in direct search methods can be applied, beginning from the same (or different) starting solutions. Decomposition methods for structured problems (linear, quadratic, or separable programming) are also used (see Dennis

and Wu [97]). What is more, continuous optimization has a strong relationship with partial differential equations and generally with numerical algebra – a typical procedure of continuous optimization requires solving a linear system (in every iteration of the algorithm); constraint or objective function requires solving a partial differential equation. Parallelization of these elements usually maintains convergence property of the method, accelerating the computations. However, as was mentioned before, the programmer quest, as an open problem, is to design a parallelization method which is adjusted directly to specificity of the problem under consideration.

16.2.2. Multiobjective optimization

The characteristics of the resources and the number of jobs to be allocated may change over time in a scheduling problem. On the other hand, many of the problems, especially taken from the real-world engineering optimization problems, have to optimize more than one objective at a time (which are usually in conflict). Multiobjective optimization is not restricted to finding a single solution, instead it points out a set of solutions, known as the Pareto front.

Parallel processing can be useful in efficient solving multiobjective optimization problems in (at least) two ways:

(1) NP-hard multiobjective optimization problems demand high computational resources – it is possible to parallelize the most complex elements of an algorithm (single-walk parallelizations), and

(2) parallel processing gives a possibility to find the whole front of Pareto-optimal solutions instead of a single Pareto-optimal solution (multiple-walk parallelization).

Up to now, parallel multiobjective optimization has usually been connected with population-based approach (Toro et al. [253], Van Velhuizen et al. [259]). This state results from several reasons. Firstly, many objectives and the Pareto dominance relationships have to be evaluated at the same time which naturally leads us to a population evaluation. Calculating the Pareto dominance relationships requires statistics of the whole population, which makes the master-worker model of computations (called the global model in the field of evolutionary algorithms) a well-fitted approach. Secondly, a hybrid model can be easily implemented in a parallel evolutionary approach (single-walk parallelism) by parallel goal function calculation together with diversification of population among processors (multiple-walk parallelization). In practice, each processor evaluates a subset of goal functions for a subset of the population. An open problem is theoretical algorithm characterization. Camara et al. [71] report the superlinear speedup in

parallel evolutionary algorithm for multi-objective optimization in dynamic environments – the knowledge about the source of this anomaly (also reported by other authors – see Sections 1.1 and 12.2) is generally weak for the case of parallel multiobjective evolutionary approach. Furthermore, it is interesting to study the scalability and performance behavior, especially for the asynchronous parallel algorithm, also with different communication schemes.

16.2.3. Uncertain data

Realization of the real-world application is frequently connected with technological and management idle times which appear during work process. External factors (e.g. weather) and internal distractions make scheduling out-of-date and aberrations from terms which are stated in contracts. Therefore a way of data representation which determines real terms is required. The consequence of errors is high penalties or even removing of the company from offer processes. Most of the discrete optimization problems presented in the literature do not take into consideration the difficulty with assigning an exact value of parameters, e.g. times of job execution. Such an uncertainty can be modelled by using fuzzy numbers theory.

Scheduling problems were fuzzified by using the concept of fuzzy due date and processing times. Dumitru and Luban [101] investigate the application of fuzzy sets to the problem of production scheduling. Tsujimura et al. [255] present a branch and bound algorithm for the three-machine flow shop problem when job processing times are described by triangular fuzzy numbers. Especially fuzzy logic application to the scheduling problems (by using fuzzy processing times) is presented in papers: Bożejko et al. [42, 33], Ishibuschi and Murata [143], Izzettin and Serpil [144] and Peng and Liu [205]. There are computational experiments conducted for the permutational flow shop problem in the work of Bożejko et al. [42] both for deterministic and fuzzy versions of the genetic algorithm. The obtained values of the algorithm stability level show that the fuzzy representation of the problem data and using relevant algorithm results in solutions which are 'proof' against data distractions.

As a module, fuzzy logic can be added to almost every parallel optimization algorithm, obtaining better stability – such an approach can be called *low-level approach* of fuzzification (see Bożejko et al. [33]). Another open problem is a *high-level approach* to data uncertainty in which multiple threads of the parallel optimization algorithm work on different (disturbed) instances of data (let us call them scenarios) and give a single solution which is ready for data distractions, keeping a good value of the goal function.

16.3. Future work

The presented comprehensive analysis of the results obtained clearly indicates the high efficiency of the multi-threaded approach proposed. The results fulfill the expectations of computing practitioners, so that many approaches presented here can be adapted and used in solving more complex real-world job scheduling problems. Further research is purposeful for the application of multi-threaded approach to solving the following issues which were not analyzed in this book:

- job scheduling problems with variable job execution times,
- job scheduling problems with resources,
- scheduling with fuzzy problem parameters (fuzzy processing times, deadlines, etc.).

Another sphere of research can be a broader class of NP-hard discrete optimization problems such as:

- complex variants of the traveling salesman problems (many salesmen, etc.) and their generalizations,
- packing problems,
- assignment problems,
- task scheduling with resources – tree factorial [24, 193],

with specific real-world goal functions (e.g. energy or fuel optimization).

Based on the research conducted we can formulate the following proposals. Solving scheduling problems can be speeded up in the multiprocessor environment, however proper design of an algorithm which would effectively use computational power is a nontrivial task. Designing acceptable in practice multithread algorithms requires an individual approach to each problem and it is usually a separate research problem, because properties of the multithread computing environment strongly depend on the choice of strategic approach (single or multiple-walk parallelization) as well as its elements, possible cooperation, hybridization, etc. However, the robustness of parallel and distributed calculation environments makes them a commonly used hardware. The aim of this book was to show the methodology of its using for efficient optimization.

Appendix A

Supplementary tables

Table A.1. Relative deviation of solutions of sequence and parallel memetic algorithms described in Section 8.3.

n	1 processor		4 processors*	
	av. dev.	max. dev.	av. dev.	max. dev.
40	2.907	99.963	0.057	1.534
50	4.035	167.576	0.064	1.362
100	0.005	1.054	0.004	0.103
average	**2.317**	**89.531**	**0.042**	**0.999**

* Compared to the best known solutions (taken from Bożejko and Wodecki [50]).

Table A.2. Total time of the parallel population-based algorithm described in Section 9.2.

Processors	Cooperative			Independent*		
	APRD	t_{total}(s)	t_{cpu}(s)	APRD	t_{total}(s)	t_{cpu}(s)
1	1.48%	5658	5655	1.48%	5647	5645
2	0.65%	5383	10765	0.60%	5643	11287
4	−0.26%	5580	22323	−0.17%	17836	53516
6	−0.74%	5400	32283	−0.73%	8548	52516
8	−0.32%	5218	41753	−0.97%	12129	83196
12	−1.13%	5065	60722	−1.25%	5980	67995
16	−1.78%	6865	105670	−0.80%	20124	238615
average	**−0.23%**	**5595.6**	**39881.6**	**−0.26%**	**10843.9**	**73252.9**

* The algorithm stops when the cost function value of the benchmark is achieved. Times per 120 instances (taken from Bożejko [26]).

Table A.3. Convergence of the parallel population-based metaheuristic described in Section 9.2.

Processors	Cooperative			Independent*		
	APRD	t_{total}(s)	t_{cpu}(s)	APRD	t_{total}(s)	t_{cpu}(s)
1	−0.73%	9462	9459	−0.70%	8113	8113
2	−2.09%	9669	19334	−1.76%	8310	16621
4	−3.48%	10124	40166	−2.94%	11535	39998
6	−4.20%	11963	67905	−4.13%	16916	84066
8	−4.47%	10479	83108	−4.53%	12058	88930
12	−5.00%	10311	123602	−5.01%	8770	105131
16	−5.47%	10329	165150	−5.29%	8732	139760
average	**−3.63%**	**10333.9**	**72674.9**	**−3.48%**	**10633.4**	**68945.6**

* Constant iterations number $R = 10$. Times per 120 benchmark instances (taken from Bożejko [26]).

Table A.4. PRDs of simulated annealing solution and NEH described in Section 11.1.3.

$n \times m$	1 processor	4 processors independent	4 processors with broadcasting	NEH*
20×5	0.87%	0.64%	0.62%	2.87%
20×10	2.29%	1.82%	1.70%	4.74%
20×20	1.94%	1.91%	1.82%	3.69%
50×5	0.13%	0.08%	0.13%	0.89%
50×10	1.87%	1.31%	0.92%	4.53%
50×20	2.75%	2.32%	2.29%	5.24%
100×5	0.0011%	0.0003%	0.0003%	0.46%
average	**1.41%**	**1.15%**	**1.07%**	**3.20%**

* Compared to the best solution by Taillard [243] (taken from [63]).

Table A.5. Results of computational experiments of the algorithm described in Section 9.2, Part 1.

No.	F^{**}_{alg}	PRD	No.	F^{**}_{alg}	PRD*
1	696	−28.83%	61	76373	−4.40%
2	5367	−17.29%	62	44869	−6.25%
3	**1782**	−24.11%	63	76146	−3.39%
4	6615	−20.41%	64	92860	−3.65%
5	4774	−14.84%	65	128593	−4.66%
6	7500	−9.02%	66	59852	−6.56%
7	3765	−13.39%	67	29394	−15.77%
8	153	−53.21%	68	**22120**	−16.22%
9	6628	−12.77%	69	71534	−5.14%
10	**1943**	−20.73%	70	75801	−6.65%
11	4452	−15.41%	71	**148230**	−8.06%
12	0	0.00%	72	50171	−11.88%
13	**5225**	−15.00%	73	**29076**	−20.26%
14	**2967**	−24.71%	74	**31711**	−17.19%
15	1788	−38.66%	75	23244	−24.97%
16	**4326**	−35.54%	76	56198	−16.81%
17	**127**	−72.51%	77	35932	−11.41%
18	**1337**	−46.82%	78	**20294**	−19.16%
19	**0**	−100.00%	79	**117734**	−6.43%
20	3273	−21.94%	80	**18620**	−41.53%
21	0	0.00%	81	**384547**	−0.67%
22	0	0.00%	82	**410336**	−0.76%
23	0	0.00%	83	**458879**	−1.54%
24	**1060**	−40.82%	84	**330022**	−0.49%
25	0	0.00%	85	556891	−0.30%
26	0	0.00%	86	**363265**	−0.69%
27	0	−100.00%	87	**399202**	−0.95%
28	0	−100.00%	88	**434010**	−0.65%
29	0	0.00%	89	**410739**	−1.48%
30	**0**	−100.00%	90	403601	−0.82%

* For the problem $1|s_{ij}|\sum w_iT_i$. The new 65 upper bounds are marked with bold font (counted together with those given in Table A.6).

** F_{alg} – values of the cost function obtained by the parallel algorithm considered.

Table A.6. Results of computational experiments of the algorithm described in Section 9.2, Part 2.

No.	F^{**}_{alg}	PRD	No.	F^{**}_{alg}	PRD *
31	0	0.00%	91	**342615**	−1.31%
32	0	0.00%	92	**362079**	−1.01%
33	0	0.00%	93	**407915**	−0.62%
34	0	0.00%	94	**333588**	−0.81%
35	0	0.00%	95	523309	−0.87%
36	0	0.00%	96	462961	−0.31%
37	**551**	−77.11%	97	417890	−0.57%
38	0	0.00%	98	**527603**	−0.92%
39	0	0.00%	99	**368353**	−1.72%
40	0	0.00%	100	**436004**	−1.33%
41	**69252**	−5.36%	101	**353018**	−0.79%
42	**58183**	−5.94%	102	**493473**	−0.54%
43	**146549**	−2.29%	103	**378864**	−0.34%
44	**35511**	−8.30%	104	**358073**	−1.09%
45	**59280**	−5.54%	105	**350806**	−23.13%
46	**35442**	−6.71%	106	**454769**	−1.12%
47	**73412**	−4.89%	107	**352766**	−1.09%
48	65943	−4.32%	108	**461953**	−1.32%
49	**78463**	−6.75%	109	**413019**	−0.67%
50	**31996**	−11.70%	110	**419437**	−0.44%
51	**50459**	−13.85%	111	**344604**	−1.74%
52	**97052**	−7.89%	112	376036	−0.37%
53	**90028**	−5.68%	113	**260124**	−1.17%
54	124708	0.93%	114	**469900**	−0.70%
55	71657	−6.17%	115	464415	0.91%
56	**77552**	−12.29%	116	537799	−0.45%
57	**68415**	−2.84%	117	**508325**	−1.98%
58	**47754**	−13.99%	118	357087	−0.14%
59	**53693**	−9.09%	119	**577318**	−1.14%
60	**66966**	−8.68%	120	402422	0.68%
			Average		**−12.08**

* For the problem $1|s_{ij}|\sum w_iT_i$.

** F_{alg} – values of the cost function obtained by the parallel algorithm considered.

Table A.7. Improvement of NEH solution of algorithms from Section 11.1.3.

$n \times m$	1 processor	4 processors independent	4 processors with broadcasting*
20×5	1.94%	2.17%	2.19%
20×10	2.34%	2.79%	2.90%
20×20	1.69%	1.72%	1.80%
50×5	0.75%	0.80%	0.75%
50×10	2.54%	3.08%	3.45%
50×20	2.37%	2.77%	2.80%
100 ×5	0.46%	0.46%	0.46%
average	**1.74%**	**1.98%**	**2.07%**

* Taken from Bożejko and Wodecki [63].

Table A.8. Results of APRD for reference solutions [273] obtained by algorithms presented in Section 11.2.3.

$n \times m$	1 processor (sSA)			4 processors (pSA)*		
	average	minimal	std. dev.	average	minimal	std. dev.
20×5	0.23%	0.05%	0.16%	0.08%	0.00%	0.08%
20×10	0.27%	0.04%	0.17%	0.05%	0.00%	0.05%
20×20	0.14%	0.01%	0.12%	0.04%	0.00%	0.04%
50×5	1.51%	1.25%	0.19%	1.25%	0.91%	0.20%
50×10	1.97%	1.56%	0.26%	1.54%	1.09%	0.29%
average	**0.82%**	**0.58%**	**0.18%**	**0.59%**	**0.40%**	**0.13%**

* Taken from Bożejko and Wodecki [59].

Table A.9. Values of APRD for parallel scatter search algorithm for the $F||C_{\max}$ problem from Section 12.2 (global model).

$n \times m$	Processors*				
	1	2	4	8	16
iterations→	9,600	4,800	2,400	1,200	600
20 × 5	0.000%	0.000%	0.000%	0.000%	0.096%
20 × 10	0.097%	0.060%	0.072%	0.131%	0.196%
20 × 20	0.039%	0.035%	0.061%	0.062%	0.136%
50 × 5	0.007%	−0.001%	−0.015%	-0.001%	0.007%
50 × 10	0.345%	0.104%	0.113%	0.123%	0.272%
average	**0.098%**	**0.029%**	**0.046%**	**0.063%**	**0.142%**
t_{total} (h:min:sec)	30:04:40	15:52:13	7:40:51	3:35:47	1:42:50
t_{cpu} (h:min:sec)	30:05:02	31:44:21	30:41:54	28:45:30	27:24:58

* The sum of iterations for all processors is 9,600 (from [40]).

Table A.10. Values of APRD for parallel scatter search algorithm for the $F||C_{\max}$ problem from Section 12.2 (independent model).

$n \times m$	Processors*				
	1	2	4	8	16
iterations→	9,600	4,800	2,400	1,200	600
20 × 5	0.000%	0.000%	0.000%	0.000%	0.096%
20 × 10	0.097%	0.080%	0.066%	0.039%	0.109%
20 × 20	0.039%	0.062%	0.048%	0.031%	0.031%
50 × 5	0.007%	0.000%	0.007%	0.007%	0.000%
50 × 10	0.345%	0.278%	0.148%	0.238%	0.344%
average	**0.098%**	**0.084%**	**0.054%**	**0.063%**	**0.097%**
t_{total} (h:min:sec)	30:04:40	14:38:29	6:58:59	3:15:34	1:32:46
t_{cpu} (h:min:sec)	30:05:02	29:16:14	27:54:19	26:03:33	24:41:24

* The sum of iterations for all processors is 9,600 (from [40]).

Table A.11. The parallel scatter search (independent model – no communication) from Section 12.2 for C_{sum} criterion.

$n \times m$	Processors*			
	1	2	4	8
iterations→	1,600	800	400	200
20x5	0.007	0.021	0.065	0.111
20x10	0.000	0.012	0.010	0.024
20x20	0.000	0.013	0.047	0.046
50x5	1.024	1.093	1.364	1.662
50x10	1.060	1.312	1.425	1.821
average	**0.418**	**0.490**	**0.582**	**0.733**

* Average percentage deviations ARPD. The sum of iterations for all processors is 1,600 (from [40]).

Table A.12. The parallel scatter search (independent model) from Section 12.2 for C_{sum} criterion.

Processors	Cluster of Xeon 3000 2.4 GHz processors*	
	t_{total}(hours:min:sec)	t_{cpu}(hours:min:sec)
1	7:13:30	7:13:13
2	3:34:08	7:04:44
4	1:46:05	6:58:43
8	0:53:33	6:57:44

* Execution times, for all 50 instances, *iter* = 1,600 (taken from [40]).

Table A.13. The parallel scatter search (global model – with communication) from Section 12.2 for C_{sum} criterion.

$n \times m$	Processors*			
	1	2	4	8
iterations→	1,600	800	400	200
20×5	0.21	0.020	0.007	0.077
20×10	0.037	0.006	0.004	0.013
20×20	0.008	0.000	0.004	0.015
50×5	0.917	0.762	0.978	1.208
50×10	1.171	0.860	1.126	1.448
average	**0.431**	**0.330**	**0.423**	**0.552**

* Average percentage deviations ARPD. The number of iterations for all processors totals 1,600 (from [40]).

Table A.14. The parallel scatter search (global model) from Section 12.2 for C_{sum} criterion.

Processors	Cluster of Xeon 3000 2.4 GHz processors*	
	t_{total}(hours:min:sec)	t_{cpu}(hours:min:sec)
1	7:26:00	7:25:51
2	3:52:36	7:17:39
4	2:14:02	7:04:07
8	1:24:31	7:06:52

* Execution times, for all 50 instances, *iter* = 1,600 (from [40]).

Table A.15. The parallel scatter search (independent model – no communication) from Section 12.2 for C_{sum} criterion.

$n \times m$	Processors*			
	1	2	4	8
iterations→	16,000	8,000	4,000	2,000
20×5	0.000	0.007	0.000	0.006
20×10	0.000	0.000	0.000	0.000
20×20	0.000	0.000	0.000	0.000
50×5	0.904	1.037	0.906	0.903
50×10	0.913	0.986	1.033	0.989
average	**0.363**	**0.406**	**0.388**	**0.380**

* The average percentage deviations ARPD. The number of iterations for all processors totals 16,000 (from [40]).

Table A.16. The parallel scatter search (independent model) from Section 12.2 for C_{sum} criterion.

Processors	Cluster of Xeon 3000 2.4 GHz processors*	
	t_{total}(hours:min:sec)	t_{cpu}(hours:min:sec)
1	75:27:40	75:25:48
2	37:40:08	75:02:51
4	18:38:23	74:10:18
8	9:06:24	72:19:26

* Execution times, for all 50 instances, *iter* = 16,000 (taken from [40]).

Table A.17. The parallel scatter search (global model – with communication) from Section 12.2 for C_{sum} criterion.

$n \times m$	Processors*			
	1	2	4	8
iterations→	16,000	8,000	4,000	2,000
20×5	0.000	0.000	0.000	0.008
20×10	0.000	0.000	0.000	0.004
20×20	0.000	0.000	0.000	0.000
50×5	0.993	0.677	0.537	0.449
50×10	1.103	0.648	0.474	0.404
average	**0.419**	**0.265**	**0.202**	**0.173**

* The average percentage deviations ARPD. The sum of iterations for all processors is 16,000 (from [40]).

Table A.18. The parallel scatter search (global model) from Section 12.2 for C_{sum} criterion.

Processors	Cluster of Xeon 3000 2.4 GHz processors*	
	t_{total}(hours:min:sec)	t_{cpu}(hours:min:sec)
1	75:23:43	75:20:42
2	41:19:51	77:57:57
4	23:28:19	75:46:07
8	14:30:03	74:38:51

* Times of execution, for all 50 instances, $iter$ = 16,000 (from [40]).

Table A.19. Relative percentage distance of parallel synchronous tabu search (PSTS) solutions presented in Section 15.3.

$n \times m$	PSTS algorithm	NEH algorithm*
20×5	0.92%	2.87%
20×10	1.98%	4.74%
20×20	2.68%	3.69%
50×5	0.39%	0.89%
50×10	1.86%	4.53%
50×20	2.57%	5.24%
100×5	0.14%	0.46%
average	**1.51%**	**3.20%**

* Compared to the best solution by Taillard [243], 1,000 iterations (taken from Bożejko and Wodecki [58]).

Table A.20. Relative percentage distances of parallel asynchronous tabu search (PATS) from Section 15.3.

$n \times m$	Processors*			
	1	4	8	12
20×5	0.84%	0.19%	0.24%	0.05%
20×10	1.59%	0.89%	0.83%	0.77%
20×20	1.10%	0.62%	0.62%	0.63%
50×5	0.25%	0.01%	0.02%	0.03%
50×10	2.34%	0.87%	0.76%	0.63%
50×20	2.81%	1.81%	1.89%	1.67%
100×5	0.15%	0.01%	0.04%	0.04%
average	**1.30%**	**0.63%**	**0.63%**	**0.55%**

* Compared to the best solution by Taillard [243], for 1,000 iterations (taken from Bożejko and Wodecki [58]).

Table A.21. Parallel genetic algorithm described in Section 13.1.

$n \times m$	1 processor	4 processors*			
		independent		cooperative	
		the same operators	different operators	the same operators	different operators
20×5	1.00%	0.81%	0.73%	0.66%	0.52%
20×10	1.10%	1.00%	0.97%	0.81%	0.79%
20×20	0.93%	0.75%	0.74%	0.65%	0.64%
50×5	2.96%	3.70%	3.44%	3.43%	3.10%
50×10	4.48%	4.97%	4.70%	4.79%	4.64%
average	**2.13%**	**2.25%**	**2.11%**	**2.07%**	**1.98%**
std. dev.**	0.20%	0.15%	0.12%	0.16%	0.12%

* Different start subpopulations, various genetic operators (taken from [61]).

** Standard deviation.

Table A.22. Algorithms from Section 14.2.

Problem	$n \times m$	(LB,UB)	MG [180]	hGA [111]	TSBM^{2}H*
mt10c1	10×11	(655,927)	928	927	927
mt10cc	10×12	(655,914)	910	910	**908**
mt10x	10×11	(655,929)	918	918	922
mt10xx	10×12	(655,929)	918	918	918
mt10xxx	10×13	(655,936)	918	918	918
mt10xy	10×12	(655,913)	906	905	905
mt10xyz	10×13	(655,849)	847	849	849
setb4c9	15×11	(857,924)	919	914	914
setb4cc	15×12	(857,909)	909	914	**907**
setb4x	15×11	(846,937)	925	925	925
setb4xx	15×12	(846,930)	925	925	925
setb4xxx	15×13	(846,925)	925	925	925
setb4xy	15×12	(845,924)	916	916	**910**
setb4xyz	15×13	(838,914)	905	905	**903**
seti5c12	15×16	(1027,1185)	1174	1175	1174
seti5cc	15×17	(955,1136)	1136	1138	1136
seti5x	15×16	(955,1218)	1201	1204	**1198**
seti5xx	15×17	(955,1204)	1199	1202	**1197**
seti5xxx	15×18	(955,1213)	1197	1204	1197
seti5xy	15×17	(955,1148)	1136	1136	1136
seti5xyz	15×18	(955,1127)	1125	1126	1128

* Values of solutions obtained for Barnes and Chambers [21] benchmark instances. The TSAB algorithm was used in the operation scheduling module of the M^{2}H. The new best known solutions are marked with bold font.

Bibliography

[1] Aarts E.H.L., Verhoeven M., *Local search*, in: M. Dell'Amico, F. Maffioli, S. Martello (Eds.), Annotated Bibliographies in Combinatorial Optimization, Wiley and Sons, Chichester, (1997).

[2] Aarts E., Lenstra J., *Local search in combinatorial optimization*, New York (1997).

[3] Abdul-Razaq T.S. , Potts C.N., Van Wassenhove L.N., *A survey of algorithms for the single machine total weighted tardiness scheduling problem*, Discrete Applied Mathematics 26 (1990), 235–253.

[4] Adrabiński A, Wodecki M., *An algorithm for solving the machine sequencing problem with parallel machines*, Zastosowania Matematyki XVI 3 (1979), 513–541.

[5] Ahn C.W., Goldberg D.E., Ramakrishna R.S., *Multiple-deme parallel estimation of distribution algorithms: Basic framework and application*, Parallel Processing and Applied Mathematics PPAM 2003, Lecture Notes in Computer Science No. 3019, Springer (2004), 544–551.

[6] Ahuja R.K., Ergun O., Orlin J.B., Punnen A.P., *A survey of very large-scale neighborhood search techniques*, Discrete Applied Mathematics 123 (2002), 75–102.

[7] Alba E., *Parallel Metaheuristics. A New Class of Algorithms*, Wiley & Sons Inc. (2005).

[8] Alba E., Troya J.M., *Analyzing synchronous and asynchronous parallel distributed genetic algorithms*, Future Generation Computer Systems 17 (2001), 451–465.

[9] Alba E., Nebro A.J., Troya J.M., *Heterogeneous Computing and Parallel Genetic Algorithms*, Journal of Parallel and Distributed Computing 62 (2002), 1362–1385.

[10] Alba E., Almeida F., Blesa M., Cotta C., Diaz M., Dorta I., Gabarró J., González J., León C., Moreno L., Petit J., Roda J., Rojas A., Xhafa F., *Malba: A library of skeletons for combinatorial optimisation*, in: B. Monien,

R. Feldman (Eds.), Euro-Par 2002 Parallel Processing, Lecture Notes in Computer Sceince No. 2400, Springer (2002), 927–932.

[11] Amza C., Cox A.L., Dwarkadas S., Keleher P., Rajamony R., Lu H., Yu W., Zwaenepoel W., *ThreadMarks: Shared memory computing on networks of workstations*, IEEE Computer 29(2), (1996), 18–28.

[12] Angel E., Bampis E., *A multi-start dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem*, European Journal of Operational Research 162 (2005), 281–289.

[13] Argonne National Laboratory, *PGAPack Parallel Genetic Algorithm Library*, on-line document, http://wwwfp.mcs.anl.gov/ CCST/ research/ reports/ pre1998/ comp bio/ stalk/ pgapack.html

[14] Armentano V.A., Scrich C.R., *Tabu search for minimizing total tardiness in a job shop*, International Journal of Production Economics 63(2), (2000), 131–140.

[15] Babu B., Peridy L., Pison E., *A branch and bound algorithm to minimize total weighted tardiness on a single processor*, Annals of Operations Research 129 (2004), 33–46.

[16] Badeau P., Guertin F., Gendreau M., Potvin J.Y., Taillard E., *A parallel tabu search heuristic for the vehicle routing problem with time windows*, Transportation Research-C 5 (1997), 109–122.

[17] Baker K.R., Scudder G.D., *Sequencing with earliness and tardiness penalties: a review*, Operations Research 38 (1990), 22–36.

[18] Balas E, Vazacopoulos A. *Guided local search with shifting bottleneck for job-shop scheduling*, Management Science 44(2), (1969), 262–275.

[19] Bank J., Werner F., *Heuristic algorithm for unrelated parallel machine scheduling with a common due date, relase dates, and linear earliness and tardiness penalties*, Mathematical and Computer Modelling 33 (2001), 363–383.

[20] Barr R.S., Hickman B.L., *Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions*, ORSA Journal on Computing 5(1), (1993), 2–18.

[21] Barnes J.W., Chambers J.B., *Flexible job shop scheduling by tabu search, Graduate program in operations research snd industrial engineering*, The University of Texas at Austin (1996), Technical Report Series: ORP96-09.

[22] Beasley J.E., *OR-Library: distributing test problems by electronic mail*, Journal of the Operational Research Society 41 (1990), 1069–1072. (http://people.brunel.ac.uk/~mastjjb/jeb/info.html)

[23] Berger J., Barkaoui M., *A Memetic Algorithm for the Vehicle Routing Problem with Time Windows*, Proceedings of the 7th International Command and Control Research and Technology Symposium, http://www.dodccrp.org/events/7th_ICCRTS/Tracks/pdf/035.pdf

[24] Belinschi S., Bożejko M., Lehner F., Speicher R., *The normal distribution is $\boxplus$-infinitely divisible*, Advances in Mathematics (2010), doi: 10.1016/j.aim.2010.10.025

[25] Bożejko W., *Solving the flow shop problem by parallel programming*, Journal of Parallel and Distributed Computing 69, Elsevier (2009), 470–481.

[26] Bożejko W., *Parallel path relinking method for the single machine total weighted tardiness problem with sequence-dependent setups*, Journal of Intelligent Manufacturing 21(6), Springer (2010), 777–785.

[27] Bożejko W., Wodecki M., *Solving Permutational Routing Problems by Population-Based Metaheuristics*, Computers & Industrial Engineering 57, Elsevier (2009), 269–276.

[28] Bożejko W., Uchroński M., Wodecki M., *Parallel hybrid metaheuristics for the flexible job shop problem*, Computers & Industrial Engineering 59, Elsevier (2010), 323—333.

[29] Bożejko W., Makuchowski M., *A fast hybrid tabu search algorithm for the no-wait job shop problem*, Computers & Industrial Engineering 56, Elsevier (2009), 1502–1509.

[30] Bożejko W., Uchroński M., Wodecki M., *The new golf neighborhood for the flexible job shop problem*, Proceedings of the ICCS 2010, Procedia Computer Science 1, Elsevier (2010), 289–296.

[31] Bożejko W., Uchroński M., Wodecki M., *Parallel Meta^{2}2heuristics for the Flexible Job Shop Problem*, in: L. Rutkowski et al. (Eds.), Proceedings of the ICAISC 2010, Lecture Notes in Artificial Intellignece No. 6114 (2010), Springer, 395–402.

[32] Bożejko W., Uchroński M., *A Neuro-Tabu Search Algorithm for the Job Shop Problem*, in: L. Rutkowski et al. (Eds.), Proceedings of the ICAISC 2010, Lecture Notes in Artificial Intelligence No. 6114 (2010), Springer, 387–394.

[33] Bożejko W., Czapiński M., Wodecki M., *Parallel Hybrid Metaheuristics for the Scheduling with Fuzzy Processing Times*, in: L. Rutkowski et al. (Eds.), Proceedings of the ICAISC 2010, Lecture Notes in Artificial Intellignece No. 6114 (2010), Springer, 379–386.

[34] Bożejko W., Pempera J., Smutnicki C., *Parallel simulated annealing for the job shop scheduling problem*, in: Allen G et al. (Eds.) ICCS 2009, Part I, Lecture Notes in Computer Science No. 5544, Springer (2009), 631–640.

[35] Bożejko W., Smutnicki C., Uchroński M., *Parallel calculating of the goal function in metaheuristics using GPU*, in: G. Allen et al. (Eds.), ICCS 2009, Part I, Lecture Notes in Computer Science No. 5544 (2009), 1022–1031.

[36] Bożejko W., Wodecki M., *Parallel population training metaheuristics for the routing problem*, in: L. Zadeh, L. Rutkowski, R. Tadeusiewicz, J. Zurada (Eds.), International Conference on Artificial Intelligence and Soft Computing (ICAISC 2008), IEEE Computational Intelligence Society – Poland Chapter and the Polish Neural Network Society (2008), 463–472.

[37] Bożejko W., Pempera J., Smutnicki A., *Multi-thread parallel metaheuristics for the flow shop problem*, in: L. Zadeh, L. Rutkowski, R. Tadeusiewicz, J. Zurada (Eds.), International Conference on Artificial Intelligence and Soft Computing (ICAISC 2008), IEEE Computational Intelligence Society – Poland Chapter and the Polish Neural Network Society (2008), 454–462.

[38] Bożejko W., Wodecki M., *Parallel path-relinking method for the flow shop scheduling problem*, in: International Conference on Computational Science (ICCS 2008), Lecture Notes in Computer Science No. 5101, Springer (2008), 264–273.

[39] Bożejko W., Pempera J., Smutnicki C., *Parallel single-thread strategies in scheduling*, in: L. Rutkowski, R. Tadeusiewicz, L.A. Zadeh, J.M. Zurada (Eds.), Artificial Intelligence and Soft Computing – ICAISC 2008, Lecture Notes in Artifivial Intelligence No. 5097, Springer (2008), 995–1006.

[40] Bożejko W., Wodecki M., *Parallel scatter search algorithm for the flow shop sequencing problem*, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007), Lecture Notes in Computer Science No. 4967, Springer (2008), 180–188.

[41] Bożejko W., Pempera J., *Parallel Tabu Search Algorithm for the Permutation Flow Shop Problem with Criterion of Minimizing Sum of Job Completion Times*, Conference on Human System Interaction HSI'08, IEEE Computer Society, 1-4244-1543-8/08/(c)2008 IEEE.

[42] Bożejko W., Hejducki Z., Wodecki M., *Fuzzy Blocks in Genetic Algorithm For the Flow Shop Problem*, Conference on Human System Interaction HSI'08, IEEE Computer Society, 1-4244-1543-8/08/(c)2008 IEEE.

[43] Bożejko W., Wodecki M., *Applying Multi-Moves in Parallel Genetic Algorithm for the Flow Shop Problem*, in: Computation in Modern Science and Engineering: Proceedings of the International Conference on Computational Methods in Science and Engineering 2007 (ICCMSE 2007): Volume 2, Part B, AIP Conference Proceedings Volume 963 (2007), 1162–1165.

[44] Bożejko W., Wodecki M., *Parallel Evolutionary Algorithm for the Traveling Salesman Problem*, Journal of Numerical Analysis, Industrial and Applied Mathematics 2(3–4), (2007), 129–137.

[45] Bożejko W., Wodecki M., *A parallel metaheuristics for the single machine total weighted tardiness problem with sequence-dependent setup times*, Proceedings of the 3rd Multidisciplinary International Scheduling Conference: Theory and Applications, Paris 28–31 August (2007), 96–103.

[46] Bożejko W., Wodecki M., *On the theoretical properties of swap multimoves*, Operations Research Letters 35(2), Elsevier (2007), 227–231.

[47] Bożejko W., Wodecki M., *Population-Based Approach for the Quadratic Assignment Problem*, International Conference on Numerical Analysis and Applied Mathematics 2006, Wiley-VCH Verlag (2006), 61–64.

[48] Bożejko W., Wodecki M., *The new concepts in neighborhood search for permutation optimization problems*, in: E.K. Burke, H. Rudová (Eds.), Practice and Theory of Automated Time-tabling PATAT 2006, Brno (2006), 363–366.

[49] Bożejko W., Wodecki M., *Theoretical properties of multimoves in metaheuristics in aspect of involutions*, Proceedings of Tenth International Workshop on Project Management and Scheduling, NAKOM, Poznań (2006), 88–94.

[50] Bożejko W., Wodecki M., *A new inter-island genetic operator for optimization problems with block properties*, Lecture Notes in Artificial Intelligence No. 4029, Springer (2006), 324–333.

[51] Bożejko W., Wodecki M., *Parallel population training algorithm for single machine total tardiness problem*, in: A. Cader, L. Rutkowski, R. Tadeusiewicz, J. Zurada (Eds.), Artificial Intelligence and Soft Computeing, Academic Publishing House EXIT (2006), 419–426.

[52] Bożejko W., Wodecki M., *Evolutionary Heuristics for Hard Permutational Optimization Problems*, Internationam Journal of Computational Intelligence Research 2(2), (2006), 151–158.

[53] Bożejko W., Grabowski J., Wodecki M., *Block approach tabu search algorithm for single machine total weighted tardiness problem*, Computers & Industrial Engineering 50(1–2), Elsevier (2006), 1–14.

[54] Bożejko W., Wodecki M., *A fast parallel dynasearch algorithm for some scheduling problems*, Proceedings of PARELEC 2006, IEEE Computer Society (2006), 275–280.

[55] Bożejko W., Wodecki M., *Parallel Evolution Heuristic Approach for the Traveling Salesman Problem*, International Conference on Numerical Analysis and Applied Mathematics 2005, Wiley-VCH Verlag (2005), 90–93.

[56] Bożejko W., Wodecki M., *A hybrid evolutionary algorithm for some discrete optimization problems*, Proceedings of the 5th International Conference on Intelligent Systems Design and Applications ISDA 2005, IEEE Computer Society (2005), 326–331.

[57] Bożejko W., Wodecki M., *Task realiation's optimization with earliness and tardiness penalties in distributed computation systems*, Lecture Notes in Computer Science No. 3528, Springer (2005), 69–75.

[58] Bożejko W., Wodecki M., *Parallel tabu search method approach for very difficult permutation scheduling problems*, Proceedings of PARELEC 2004, IEEE Computer Society Press (2004), 156–161.

[59] Bożejko W., Wodecki M., *The new concepts in parallel simulated annealing method*, Lecture Notes in Computer Science No. 3070, Springer (2004), 853–859.

[60] Bożejko W., Wodecki M., *Parallel genetic algorithm for minimizing total weighted completion time*, Lecture Notes in Computer Science No. 3070, Springer (2004), 400–405.

[61] Bożejko W., Wodecki M., *Parallel genetic algorithm for the flow shop scheduling problem*, Lecture Notes in Computer Science No. 3019, Springer (2004), 566–571.

[62] Bożejko W., *Parallel scheduling algorithms* (Ph.D. thesis, in Polish), Technical Report of the Institute of Engineering Cybernetics No. 29/2003, Wrocław University of Technology (2003), 1–205.

[63] Bożejko W., Wodecki M., *Solving the flow shop problem by parallel simulated annealing*, Lecture Notes in Computer Science No. 2328, Springer Verlag 2002, 236–247.

[64] Bożejko W., Wodecki M., *Parallel algorithm for some single machine scheduling problems*, Automatyka 134 (2002), 81–90.

[65] Bożejko W., Wodecki M., *Solving the flow shop problem by parallel tabu search*, Proceedings of PARALEC 2002, IEEE Computer Society (2002), 189–194.

[66] Bradwell R., Brown K., *Parallel asynchronous memetic algorithms*, in: E. Cantu-Paz, B. Punch (Eds.), Evolutionary Computation (1999), 157–159.

[67] Brandimarte P., *Routing and scheduling in a flexible job shop by tabu search*, Annals of Operations Research 41 (1993), 157–183.

[68] Bubak M., Sowa K., *Objectoriented implementation of parallel genetic algorithms*, in: R. Buyya (Ed.), High Performance Cluster Computing: Programming and Applications Vol. 2, Prentice Hall (1999), 331–349.

[69] Bushee D.C., Svestka. J.A., *A bi-directional scheduling approach for job shops*, International Journal of Production Research 37(16), (1999), 3823–3837.

[70] Cahon S., Melab N., Talbi E.-G., *ParadisEO on Condor-MW for optimization on computational grids*, http://www.lifl.fr/∼cahon/cmw/index.html (2004).

[71] Camara M., Ortega J., Toro F.J., *Parallel Processing for Multi-objective Optimization in Dynamic Environments*, 2007 IEEE International Parallel and Distributed Processing Symposium (2007), 243–250.

[72] Cantú-Paz E., *Theory of Parallel Genetic Algorithms*, in: E. Alba (Ed.), Parallel Metaheuristics, Wiley (2005), 425–444.

[73] Carlier J., Pinson E., *An algorithm for solving the job shop problem*, Management Science 35 (1989), 164–176.

[74] Calrier J., Villon P., *A new heuristic for the traveling salesman problem*, RAIRO Operations Research 24 (1990), 245–253.

[75] Černý V., *Thermodynamical approach to travelling salesman problem: An efficient simulation algorithm*, Journal of Optimization Theory and Applications 45 (1985), 41–51.

[76] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc. (2001).

[77] Cheng T.C.E., Ng C.T., Yuan J.J., Liu Z.H., *Single machine scheduling to minimize total weighted tardiness*, European Journal of Operational Research 165 (2005), 423–443.

[78] Cicirello V.A., Smith S.F., *Enhancing stochastic search performance by value-based randomization of heuristics*, Journal of Heuristics 11 (2005), 5–34.

[79] Cicirello V.A., *Non-Wrapping Order Crossover: An Order Preserving Crossover Operator that Respect Absolute Position*, 8th Annual Genetic and Evolutionary Computation Conference GECCO 2006, ACM Press (2006), 1125–1131.

[80] Cole, R., *Parallel merge sort*, SIAM Journal on Computing 17(4), (1988), 770–785.

[81] Congram K.R., Potts C.N., van de Velde S., *An interated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem*, INFORMS Journal on Computing 14 (2002), 52–67.

[82] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Introduction to Algorithms*, MIT Press and McGraw-Hill (1990).

[83] Cormen T.H., Leiserson C.E., Rivest R.L., *Introduction to Algorithms*, 2nd revised edition, MIT Press (2001).

[84] Crainic T.G., Toulouse M., *Parallel metaheuristics*, in: T.G. Crainic, G. Laporte (Eds.), Fleet management and logistics, Kluwer (1998), 205–251.

[85] Crainic T.G., Toulouse M., Gendreau M., *Parallel asynchronous tabu search in multicommodity locationallocation with balancing requirements*, Annals of Operations Research 63 (1995), 277–299.

[86] Crainic T.G., Gendreau M., *Cooperative Parallel Tabu Search for Capacited Network Design*, Journal of Heuristics 8 (2002), 601–627.

[87] Crainic T.G., Gendreau M., Hansen P., Mladenović N., *Cooperative parallel variable neighbourhood search for the p-median*, Journal of Heuristics 10 (2004), 293–314.

[88] Crauwels H.A.J., Potts C.N., Van Wassenhowe L.N., *Local search heuristics for the single machine total weighted tardiness scheduling problem*, INFORMS Journal on Computing 10(3), (1998), 341–350.

[89] CSEP, *Computational Science Education Project*, electronic book, http://www.phy.ornl.gov/csep/

[90] Cung V.-D., Martins S.L., Ribeiro C.C., Roucairol C., *Strategies for the parallel implementation of metaheuristics*, in: C.C. Ribeiro, P. Hansen (Eds.), Essays and surveys in metaheuristics, Kluwer Academic Publ. (2002), 263–308.

[91] Czech Z., *Wprowadzenie do obliczeń równoległych*, PWN, Warsaw (2010).

[92] Czech Z., *Three parallel algorithms for simulated annealing*, Lecture Notes in Computer Science No. 2328, Springer Verlag (2002), 210–217.

[93] Dauzère-Pérès S., Pauli J., *An integrated approach for modeling and solving the general multiprocessor job shop scheduling problem using tabu search*, Annals of Operations Research 70(3), (1997), 281–306.

[94] Davidor Y., *A naturally occuring niche and species phenomenon: The model and first results*, in: R.K. Belew, L.B. Booker (Eds.), Proceedings of the Fourth International Conference of Genetic Algorithms, (1991), 257–263.

[95] De Falco I., Del Balio R., Tarantino E., *Testing parallel evolution strategies on the quadratic assignment problem*, in: Proc. IEEE International Conference in Systems, Man and Cybernetics, Vol. 5 (1993), 254–259.

[96] Den Basten, M., Stützle T., Dorigo M., *Design of Iterated Local Search Algoritms An Example Application to the Single Machine Total Weighted Tardiness Problem*, in: J.W. Boers et al. (Eds.), Evo Worskshop 2001, Lecture Notes in Computer Science No. 2037 (2001), 441–451.

[97] Dennis J.E., Wu Z., *Parallel continuous optimizatin*, J. Dongarra et al. (Eds.), Sourcebook of Parallel Computing, Morgan Kauffman (2003), 649–670.

[98] DePuy G.W., Morga R.J., Whitehouse G.E., *Meta-RaPS: a simple and effective approach for solving the traveling salesman problem*, Transportation Research Part E 41 (2005), 115–130.

[99] Diaconis P., *Group Representations in Probability and Statistics*, Lecture Notes – Monograph Series Vol. 11, Institute of Mathematical Statistics, Harvard University (1988).

[100] Doerner K.F., Hartl R.F., Kiechle G., Lucka M., Reimann M., *Parallel Ant Systems for the Capacited VRP*, in: J. Gottlieb, G.R. Raidl (Eds.), EvoCOP'04, Springer (2004), 72–83.

[101] Dumitru V., Luban F., *Membership functions, some mathematical programming models and production scheduling*, Fuzzy Sets and Systems 8 (1982), 19–33.

[102] Emmons H., *One machine sequencing to Minimize Certain Functions of Job Tardiness*, Operations Research 17 (1969), 701–715.

[103] Feldmann M., Biskup D., *Single-machine scheduling for minimizing earliness and tardiness penalties by meta-heuristic approaches*, Computers & Industrial Engineering 44 (2003), 307–323.

[104] Fernández F., Tomassini M., Punch W.F., Sánchez-Pérez J.M., *Experimental study of multipopulation parallel genetic programming*, in: Proc. of the European Conf. on GP, Springer (2000), 283–293.

[105] Fisher M.L., *A Dual Algorithm for the One Machine Scheduling Problem*, Mathematical Programming 11 (1976), 229–252.

[106] Fiechter C.N., *A parallel tabu search algorithm for large traveling salesman problems*, Discrete Applied Mathematics 51 (1994), 243–267.

[107] Fisher, H., Thompson, G.L., *Industrial scheduling*, Englewood Cliffs, NJ: Prentice-Hall (1963).

[108] Flynn M.J., *Very highspeed computing systems*, Proceedings of the IEEE 54 (1966), 1901–1909.

[109] Folino G., Pizzuti C., Spezzano G., *CAGE: A tool for parallel genetic programming applications*, in: J. Miller et al. (Eds.), Proceedings of EuroGP'2001, Lecture Notes in Computer Science No. 2038, Springer (2001), 64–73.

[110] Gagné C., Price W.L., Gravel M., *Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times*, Journal of the Operational Research Society 53 (2002), 895–906.

[111] Gao J., Sun L., Gen M., *A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems*, Computers & Operations Research 35 (2008), 2892–2907.

[112] García-López F., Melián-Batista, Moreno-Pérez J., Moreno-Vega J.M., *The parallel variable neighborhood search for the p-median problem*, Journal of Heuristics 8 (2002), 375–388.

[113] García-López F., Melián-Batista, Moreno-Pérez J., Moreno-Vega J.M., *Parallelization of the Scatter Search*, Parallel Computing 29 (2003), 575–589.

[114] García-López F., García Torres M., Melián-Batista, Moreno-Pérez J., Moreno-Vega J.M., *Solving Feature Subset Selection Problem by a Parallel Scatter Search*, European Journal of Operational Research Volume 169(2), (2006), 477–489.

[115] Garey M.R., Johnson D.S., Seti R., *The complexity of flowshop and jobshop scheduling*, Mathematics of Operations Research 1 (1976), 117–129.

[116] Geist A., Beguelin A., Dongarra J., Manchek R., Jaing W., Sunderam V., *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Boston (1994).

[117] Goldberg D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc., Massachusetts (1989).

[118] Grabowski J., Wodecki M., *A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion*, Computers & Operations Research 31 (2004), 1891–1909.

[119] Grabowski J., Wodecki M., *A very fast tabu search algorithm for the job shop problem*, in: C. Rego, B. Alidaee (Eds.), Adaptive memory and evolution, tabu search and scatter search. Kluwer Academic Publishers, Dordrecht (2005).

[120] Grabowski J., *A new algorithm of solving the flow-shop problem*, Operations Research in Progress, D. Reidel Publishing Company (1982) 57–75.

[121] Grabowski J., Pempera J., *New block properties for the permutation flow shop problem with application in tabu search*, Journal of Operational Research Society 52 (2000), 210–220.

[122] Grabowski J., *Generalized problems of operations sequencing in the discrete production systems*, (in Polish), Monographs 9, Scientific Papers of the Institute of Technical Cybernetics of Wrocław Technical University (1979).

[123] Graham M.R., Lawler E.L., Lenstra J.K., Rinnoy Kan A.H.G., *Optimization an approximation in detrministic sequencing and scheduling: a survey*, Annals of Discrete Mathematics 3 (1979), 287–326.

[124] Grama A., Gupta A., Karypis G., Kumar V., *Introduction to Parallel Computing*, Second Edition, Pearson Addison Wesley (2003).

[125] Grosso A., Della Croce F., Tadei R., *An enhanced dynasearch neighborhood for the single-machine total weighter tardiness scheduling problem*, Operations Research Letters 32 (2004), 68–72.

[126] Glover F., *Future Paths for Integer Programming and Links to Artificial Intelligence*, Computers & Operations Research 1(3), (1986), 533–549.

[127] Glover F., Laguna M., *Tabu Search*, Kluwer Academic Publishers, Boston (1997).

[128] Gupta S.K., Kyparisis J., *Single machine scheduling research*, OMEGA International Journal of Management Science 15 (1987), 207–227.

[129] Gutin G.M., Yeo A., *Small diameter neighborhood graphs for the traveling salesman problem: at most four moves from tour to tour*, Computers & Operations Research 26 (1999), 321–327.

[130] Gutin G., *Exponential neighborhood local search for the traveling salesman problem*, Special Issue of Computers & Operations Research 26 (1999), 313–320.

[131] Haldar A.M., Nayak A., Choudhary A., Banerjee P., *Parallel Algorithms for FPGA Placement*, Proceedings of the Great Lakes Symposium on VLSI (GVLSI 2000), Chicago, IL (2000).

[132] Hanafi S., *On the Convergence of Tabu Search*, Journal of Heuristics 7 (2000), 47–58.

[133] He Z., Yang T., Tiger A., *An exchange heuristic embedded with simulated annealing for due-dates job-shop scheduling*, European Journal of Operational Research 91 (1996), 99–117.

[134] High Performance Fortran Forum, *High Performance Fortran language specification*, Scientific Programming 2, 13 (1993), 1–170.

[135] Ho N.B., Tay J.C., *GENACE: an efficient cultural algorithm for solving the Flexible Job-Shop Problem*, IEEE International Conference on Robotics and Automation (2004), 1759–1766.

[136] Holland J.H., *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*, University of Michigan Press (1975).

[137] Holthaus O., Rajendran C., *Efficient jobshop dispatching rules: further developments*, Production Planning and Control 11 (2000), 171–178.

[138] Hoogeveen J.A., van de Velde S.L., *A branch and bound algorithm for single-machine earliness-tardiness scheduling with idle time*, INFORMS Journal on Computing 8 (1996), 402–412.

[139] Hurink E., Jurisch B., Thole M., *Tabu search for the job shop scheduling problem with multi-purpose machine*, OR Spektrum 15 (1994), 205–215.

[140] Ignall E., Schrage L.E., *Application of the branch-and-bound technique to some flow-shop scheduling problems*, Operations Research 13(3), (1965), 400–412.

[141] Ingber L., *Lester Ingber's Archive*, http://www.ingber.com/

[142] Ishibuschi H., Misaki S., Tanaka H., *Modified Simulated Annealing Algorithms for the Flow Shop Sequencing Problem*, European Journal of Operational Research 81 (1995), 388–398.

[143] Ishibuschi H., Murata T., *Scheduling with Fuzzy Duedate and Fuzzy Processing Time*, in: R. Słowiński, M. Hapke (Eds.), Scheduling Under Fuzziness, Springer (2000), 113–143.

[144] Izzettin T., Serpil E., *Fuzzy branch-and-bound algorithm for flow shop scheduling*, Journal of Intelligent Manufacturing 15 (2004), 449–454.

[145] Jain A.S., Rangaswamy B., Meeran S., *New and stronger job-shop neighborhoods: A focus on the method of Nowicki and Smutnicki (1996)*, Journal of Heuristics 6(4), (2000), 457—480.

[146] James T., Rego C., Glover F., *Sequential and Parallel Path-Relinking Algorithms for the Quadratic Assignment Problem*, IEEE Intelligent Systems 20(4), (2005), 58–65.

[147] Janiak A., Janiak W., Lichtenstein M., *Tabu search on GPU*, Journal of Universal Computer Science 14(14), (2009), 2416–2426.

[148] Janiak A., Oguz C., Zinder Y., Do Van Ha, Lichtenstein M., *Hybrid flow-shop scheduling problems with multiprocessor task systems*, European Journal of Operational Research 152(1), (2004), 115–131.

[149] Jia H.Z., Nee A.Y.C., Fuh J.Y.H., Zhang Y.F., *A modified genetic algorithm for distributed scheduling problems*, International Journal of Intelligent Manufacturing 14 (2003), 351–362.

[150] Johnson S.M., *Optimal two and three-stage production schedules with setup times included*, Naval Research Logistic Quertely 1 (1954), 61–68.

[151] Juille H., Pollack J.B., *Massively parallel genetic programming*, in: Peter J. Angeline, K.E. Kinnear Jr. (Eds.), Advances in Genetic Programming 2, MIT Press, Cambridge (1996), 339–358.

[152] Kacem I., Hammadi S., Borne P., *Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems*, IEEE Transactions on Systems, Man, and Cybernetics, Part C 32(1), (2002), 1–13.

[153] Kawamura H., Yamamoto M., Suzuki K., Ohuchi A., *Multiple ant colonies algorithm based on colony level interactions*, IEICE Transactions on Fundamentals, E83-A(2), (2000), 371–379.

[154] Kirkpatrick S., Gellat C.D., Vecchi M.P., *Optimization by simulated annealing*, Science 220 (1983), 671–680.

[155] Kliewer G., Klohs K., Tschoke S., *Parallel simulated annealing library (parSA): User manual*, Technical report, Computer Science Department, University of Paderborn (1999).

[156] Knox J., *Tabu search performance on the symmetric traveling salesman problem*, Computers & Operations Research 21 (1994), 867–876.

[157] Koza J.R., *Genetic Programming*, The MIT Press, Cambridge (1992).

[158] Knuth D.E., *The art of computer programming*, Vol. 3., 2nd ed., Addison Wesley Longman, Inc. (1998).

[159] Kravitz S.A., Rutenbar R.A., *Placement by simulated annealing on a multiprocessor*, IEEE Transactions on Computer Aided Design 6 (1998), 534–549.

[160] Kwiatkowski J., Pawlik M., Konieczny D., *Parallel Program Execution Anomalies*, Proceedings of the International Multiconference on Computer Science and Information Technology (2006), 355–362.

[161] Lageweg B.J., Lenstra J.K., Rinnooy Kan A.H.G., *A General Bouding Scheme for the Permutation Flow-Schop Problem*, Operations Research 26 (1978), 53–67.

[162] Lawler, E.L., *A Pseudopolynomial Algorithm for Sequencing Jobs to Minimize Total Tardiness*, Annals of Discrete Mathematics 1 (1977), 331–342.

[163] Lawrence S., *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques*, Technical Report, Graduate School of Industrial Ad-ministration, Carnegie-Mellon University, Pittsburgh, Pennsylvania (1984).

[164] Lai H.T., Sahni S., *Anomalies in parallel branchandbound algorithms*, Communications of the ACM 27 (1984), 594–602.

[165] Lai T.H., Sprague A., *A note on anomalies in parallel branchandbound algorithms with onetoone bounding functions*, Information Processing Letters 23 (1986), 119–122.

[166] Lee Y.H., Bhaskaran K., Pinedo M., *A heuristic to minimize the total weighted tardiness with sequence-dependent setups*, IIE Transactions 29 (1997), 45–52.

[167] Lee S.Y., Lee K.G., *Synchronous and asynchronous parallel simulated annealing with multiple Markov chains*, IEEE Transactions on Parallel and Distributed Systems 7 (1996), 993–1008.

[168] Lenstra J.K., Rinnoy Kan A.G.H., Brucker P., *Complexity of Machine Scheduling Problems*, Annals of Discrete Mathematics, 1 (1977), 343–362.

[169] Lenstra J.K., *Sequencing by Enumeration Methods*, Mathematical Centre Tract 69, Mathematisch Centrum, Amsterdam (1977).

[170] Leung K.-S., Jin H.-D., Xu Z.-B., *An expanding self-organizing neural network for the traveling salesman problem*, Neurocomputing 62 (2004), 267–292.

[171] Li G.J., Wah B.W., *Coping with anomalies in parallel branchandbound algorithms*, IEEE Transactions on Computers C 35 1986, 568–573.

[172] Liao C.-J., Juan H.C. , *An ant opimization for single-machine tardiness shceduling with sequence-dependent setups*, Computers & Operations Research 34 (2007), 1899–1909.

[173] Lin S.-W., Ying K.-C., *Solving single-machine total weighted tardiness problems with sequence-dependent setup times by meta-heuristics*, International Journal of Advanced Manufacturing Technology 34(11–12), (2007), 1183–1190.

[174] Lin S., Kerninghan B., *An effective heuristic algorithm for the traveling salesman problem*, Operations Research 21 (1973), 498–516.

[175] Lo C.C., Hus C.C., *Annealing framework with learning memory*, IEEE Transaction on System, Man, Cybernetics, Part A 28(5), (1998), 1–13.

[176] Lobo F.G., Lima C.F., Mártires H., *An architecture for massively parallelization of the compact genetic algorithm*, in: Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2004, Lecture Notes in Computer Science No. 3103, Springer (2004), 412–413.

[177] Malek M., Guruswamy M., Pandya M., Owens H., *Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem*, Annals of Operations Research 21 (1989), 59–84.

[178] Mans B., Roucairol C., *Performances of parallel branch and bound algorithms with bestfirst search*, Discrete Applied Mathematics 66 (1996), 57–76.

[179] Martins S.L., Ribeiro C.C., Souza M.C., *A Parallel Grasp for the Steiner Problem in Graphs*, Lecture Notes in Computer Science No. 1457, Springer (1998), 285–297.

[180] Mastrolilli M., Gambardella L.M., *Effective neighborhood functions for the flexible job shop problem*, Journal of Scheduling 3(1), (2000), 3–20.

[181] Matsuo H., Suh C.J., Sullivan R.S., *A controlled search simulated annealing method for the single machine weighted tardiness problem*, Working paper 87-12-2, Departament of Management, University of Texas at Austin, TX, (1987).

[182] Mattfeld D.C., Bierwirth C., *An efficient genetic algorithm for job shop scheduling with tardiness objectives*, European Journal of Operational Research 155(3), (2004), 616–630.

[183] Meise C., *On the convergence of parallel simulated annealing*, Stochastic Processes and their Applications 76 (1998), 99–115.

[184] Mendes R., Pereira J.R., Neves J., *A Parallel Architecture for Solving Constraint Satisfaction Problems*, Proceedings of Metaheuristics Int. Conf. 2001, Porto, Portugal (2001), 109–114.

[185] Mendiburu A., Miguel-Alonso J., Lozano J.A., *Implementation and performance evaluation of a parallelization of estimation of bayesian network algorithms*, Parallel Processing Letters 16(1), (2006), 133–148.

[186] Metropolis N., Rosenbluth A.W., Teller A.H., Tellet E., *Equation of state calculation by fast computing machines*, Journal of Chemical Physics 21 (1953), 1187–1191.

[187] Michalewicz Z., *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd ed., Springer Verlag (1994).

[188] Middendorf M., Reischle F., Schmeck H., *Multi Colony Ant Algorithm*, Journal of Heuristics 8 (2002), 305–320.

[189] Miki M., Hiroyasu T., Kasai M., *Application of the temperature parallel simulated annealing to continous optimization problems*, IPSL Transactions 41 (2000), 1607–1616.

[190] Morton T.E., Rachamadougu R.M., Vepsalainen A., *Accurate myopic heuristics for tardiness scheduling*, GSIA Working Paper No. 36-83-84, Cornegie-Mellon Univercity, PA, (1984).

[191] Mühlenbein H., Gorges-Schleuter M., Kramer O., *Evolution algorithm in combinatorial optimization*, Parallel Computing 7 (1988), 65–85.

[192] Mühlenbein H., PaaßG., *From Recombination of Genes to the Estimation of Distributions Binary Parameters*, Parallel Problem Solving from Nature — PPSN IV, Lecture Notes in Computer Science No. 1141, Sprinter (1996), 178–187.

[193] Naimi M., Trehel M., Arnold A, *A $log(n)$ distributed mutual exclusion algorithm based on path reversal*, J. Parallel Distrib. Comput. 34 (1996), 1—13.

[194] Navaz M., Enscore E.E. Jr, Ham I., *A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem*, OMEGA 11(1), (1983), 91–95.

[195] Nowicki E., Smutnicki C., *A fast tabu search algorithm for the job shop problem*, Management Science 42 (1996), 797–813.

[196] Nowicki E., Smutnicki C., *A fast tabu search algorithm for the permutation flow shop problem*, European Journal of Operational Research 91 (1996), 160–175.

[197] Nowicki E., Smutnicki C., *The flow shop with parallel machines: A tabu search approach*, European Journal of Operational Research 106 (1998), 226–253.

[198] Nowicki E., Smutnicki C., *An advanced tabu search algorithm for the job shop problem*, Journal of Scheduling 8(2), (2005), 145–159.

[199] Nowicki E., Smutnicki C., *Some aspects of scatter search in the flow-shop problem*, European Journal of Operational Research 169 (2006), 654–666.

[200] Oceanásek J., Schwarz J., *The distributed bayesian optimization algorithm for combinatorial optimization*, in: EUROGEN 2001 – Evolutionary Methods for Design, Optimisation and Control, CIMNE, Athens, Greece, ISBN 84-89925-97-6 (2001), 115–120.

[201] Ogbu F., Smith D., *The Application of the Simulated Annealing Algorithm to the Solution of the $n/m/C_{max}$ Flowshop Problem*, Computers & Operations Research 17(3), (1990), 243–253.

[202] OR-Library, http://people.brunel.ac.uk/∼mastjjb/jeb/info.html

[203] Osman I., Potts C., *Simulated Annealing for Permutation Flow-Shop Scheduling*, OMEGA 17(6), (1989), 551–557.

[204] Pauli J., *A hierarchical approach for the FMS schduling problem*, European Journal of Operational Research 86(1), (1995), 32–42.

[205] Peng J., Liu B., *Parallel machine scheduling models with fuzzy processing times*, Information Sciences 166 (2004), 49–66.

[206] Pezzella F., Merelli E., *A tabu search method guided by shifting bottleneck for the job-shop scheduling problem*, European Journal of Operational Research 120 (2000), 297–310.

[207] Pezzella F., Morganti G., Ciaschetti G., *A genetic algorithm for the Flexible Job-schop Scheduling Problem*, Computers & Operations Research 35 (2008), 3202–3212.

[208] Pinedo M., *Scheduling: theory, algorithms and systems*, Englewood Cliffs, NJ: Prentice-Hall (2002).

[209] Porto S.C., Ribeiro C.C., *Parallel tabu search messagepassing synchronous strategies for task scheduling under precedence constraints*, Journal of Heuristics 1(2), (1996), 207–223.

[210] Porto S.C., Ribeiro C.C., *A tabu search approach to task scheduling on heterogeneous processors under precedence constraints*, International Journal of High Speed Computing 7 (1995), 45–71.

[211] Porto S.C., Ribeiro C.C., *A case study on parallel synchronous implementations of tabu search based on neighborhood decomposition*, Investigaci'on Operativa 5 (1996), 233–259.

[212] Porto S.C., Kitajima J.P., Ribeiro C.C., *Performance evaluation of a parallel tabu search task scheduling algorithm*, Parallel Computing 26 (2000), 73–90.

[213] Potts C.N., Van Wassenhove L.N., *Single machine tardiness sequencing heuristics*, IIE Transactions 23 (1991), 346–354.

[214] Potts C.N., Van Wassenhove L.N., *A Branch and Bound Algorithm for the Total Weighted Tardiness Problem*, Operations Research 33 (1985), 177–181.

[215] Reeves C.R., Yamada T., *Genetic algorithms, path relinking and the flowshop sequencing problem*, Evolutionary Computation 6 (1998), 45–60.

[216] Reeves C., *Improving the Efficiency of Tabu Search for Machine Sequencing Problems*, Journal of Operational Research Society 44(4), (1993), 375–382.

[217] Reeves C., *A Genetic Algorithm for Flowshop Sequencing*, Computers & Operations Research 22(1), (1995), 5–13.

[218] Reinelt G., *The traveling salesman: computational Solutions for TSP applications*, Berlin, Springer (1994).

[219] Resende M.G.C., Ribeiro C.C., *GRASP with path-relinking: Recent advances and applications*, in: T. Ibaraki, K. Nonobe, M. Yagiura (Eds.), Metaheuristics: Progress as real problem solvers, Springer (2005), 29–36.

[220] Rinnoy Kan A.H.G, *Machine Scheduling Problems: Classification, Complexity and Computations*, Nijhoff, The Hague, (1976).

[221] Ribeiro C.C., Rosseti I., *A parallel GRASP for the 2-path network design problem*, Lecture Notes in Computer Science No. 2004 (2002), Springer, 922–926.

[222] Rinnoy Kan A.G.H., Lageweg B.J., Lenstra J.K., *Minimizing total cost one-machine scheduling*, Operations Research 26 (1975), 908–972.

[223] Robilliard D., Marion-Poty V., Fonlupt C., *Population parallel GP on the G80 GPU*, in: O'Neil M. et al. (Eds.), Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, Vol. 4971, Springer (2008).

[224] Rogalska M., Bożejko W., Hejducki Z., Wodecki M., *Harmonogramowanie robót budowlanych z zastosowaniem algorytmu Tabu Search z rozmytymi czasami wykonania zadań*, Przegląd Budowlany No. 7–8 (2009), 76–80.

[225] Rogalska M., Bożejko W., Hejducki Z., *Time/cost optimization using hybrid evolutionary algorithm in construction project scheduling*, Automation in Construction 18, Elsevier (2008), 24–31.

[226] Rogalska M., Bożejko W., Hejducki Z., *Scheduling of construction projects by means of evolutionary algorithms*, Proceedings of 9th International Conference Modern building materials, structures and techniques ISARC 2007, Vilnius, Lithuania (2007), 173–174.

[227] Roussel-Ragot P., Dreyfus G., *A problem-independent parallel implementation of simulated annealing: Models and experiments*, IEEE Transactions on Computer-Aided Design 9 (1990), 827–835.

[228] Savur V., *Parallel computer architecture*, http://sankofa.loc.edu/savur/web/Parallel.html

[229] Smith W.E., *Various optimizers for single-stage production*, Naval Research Logistic Quart 3 (1956), 59–66.

[230] Schütz M., Sprave J., *Application of Parallel Mixed-Integer Evolution Strategies with Mutation Rate Pooling*, in: L.J. Fogel, P.J., Angeline, T. Bäck (Eds.), Proceedings of the Fifth Annual Conference on Evolutionary Programming (EP'96), The MIT Press (1996), 345–354.

[231] Smutnicki C., Tyński A., *Job-shop scheduling by GA : A new crossover operator*, in: H.-D. Haasis, H. Kopfer, J. Schonberger (Eds.), Operations Research, Berlin, Springer (2006), 715–720.

[232] Smutnicki C., *Some results of the worst-case analysis for flow shop scheduling*, European Journal of Operational Research 109 (1998), 66–87.

[233] Smutnicki C., *Minimizing the mean completion time in a flow shop problem. The worst-case study*, in: M. Zaborowski (Ed.), Automatyzacja procesów dyskretnych, WNT, Warsaw (2004), 151–158.

[234] Smutnicki C., *Scheduling algorithms* (in Polish), EXIT, Warsaw (2002).

[235] Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J., *MPI: The Complete Reference Vol. 1, The MPI Core*, MIT Press, Boston (1998).

[236] Sprave J., *Linear neighborhood evolution strategies*, in: A.V. Sebald, L.J. Fogel (Eds.), Proceedings of the 3th AnnualConference on Evolutionary Porgramming, World Scientific, River Edge (1994), 42–51.

[237] Steinhöfel K., Albrecht A., Wong C.K., *Fast parallel heuristics for the job shop scheduling problem*, Computers & Operations Research 29 (2002), 151–169.

[238] Stockmeyer L., Vishkin U., *Simulation of parallel random access machines by circuits*, SIAM J. Comput. 13(2), (1984), 409–422.

[239] Storer J.A., *An ntroduction to data structures and algorithms*, Birkhäuser–Springer (2001).

[240] Storer R.H., Wu S.D., Vaccari R., *New search spaces for sequencing problems with application to job shop scheduling*, Management Science 38 (1992), 1495–1509.

[241] Stützle T., *Parallelization Strategies for Ant Colony Optimization*, in: R. De Leone, A. Murli, P. Pardalos, G. Toraldo (Eds.), High Performance Algorithms and Software in Nonlinear Optimization, Vol. 24 of Applied Optimization, Kluwer (1998), 87–100.

[242] Szwarc W., *Adjacent ordering in single machine scheduling with earliness and tardiness penalties*, Neval Research Logistics 40 (1993), 229–243.

[243] Taillard E., *Benchmarks for basic scheduling problems*, European Journal of Operational Research 64 (1993), 278–285.

[244] Taillard E., *Robust taboo search for the quadratic assignment problem*, Parallel Computing 17 (1991), 443-455.

[245] Taillard E., *Parallel taboo search techniques for the job shop scheduling problem*, ORSA Journal on Computing 6 (1994), 108-117.

[246] Taillard E., *Some efficient heuristic methods for the flow shop sequencing problem*, European Journal of Operational Research 47(1), (1990), 65–74.

[247] Talbi E.-G., *A taxonomy of hybrid metaheuristics*, Journal of Heuristics 8(5), (2002), 541–564.

[248] Talbi E.-G., Roux O., Fonlupt C., Robillard D., *Parallel Ant Colonies for Combinatorial Optimization Problems*, in: Feitelson, Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing: IPPS'95 Workshop, Lecture Notes in Computer Science No. 949(11), Springer (1999).

[249] Talbi E.-G., Hafidi Z., Geib J.M., *A parallel adaptive tabu search approach*, Parallel Computing 24 (1996), 2003–2019.

[250] Tan K.C., Narasimban R., Rubin P.A., Ragatz G.L., *A comparison of four methods for minimizing total tardiness on a single procesor with sequence dependent setup times*, OMEGA 28 (2000), 313–326.

[251] Tanese R., *Distribued genetic algorithms*, in: J.D. Schaffer (Ed.), Proc. of the Third Intern. Conf. on Genetic Algorithms, Morgan Kaufmann (1989), 434–439.

[252] Tang, J., Lim M.H., Ong Y.S., *Adaptation for parallel memetic algorithm based on population entropy*, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (Seattle, Washington, USA, July 8–12, 2006), GECCO '06, ACM, New York, NY (2006), 575–582.

[253] Toro F., Ortega J., Ros E., Mota B., Paechter B., Martín J.M., *PSFGA: Parallel processing and evolutionary computation for multi-objective optimization*, Parallel Computing 30 (2004), 721–739.

[254] Tsai C.-F., Tsai C.-W., Tseng C.-C., *A new hybrid heuristic approach for solving large traveling salesman problem*, Information Sciences 166 (2004), 67–81.

[255] Tsujimura Y., Park S.H., Change I.S., Gen M., *An effective method for solving flow shop problems with fuzzy processing times*, Computers & Industrial Engieering 25(1–4), (1993), 239–242.

[256] TSPLIB Web Page, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

[257] Vaessens R., Aarts E., Lenstra J., *Job shop scheduling by local search*, INFORMS Journal on Computing 8 (1996), 303–317.

[258] Valente J.M.S., Alves R.A.F.S., *Fildered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time*, Computers & Industrial Engineering 48(2), (2005), 363–375.

[259] Van Velhuizen D.A., Zydallis J.B., Lamont G.B., *Considerations in Engineering Parallel Multi-objective Evolutionary Algorithms*, IEEE Trans. Evolutionary Computation 7(2), (2003), 144–173.

[260] Verhoeven M.G.A., Aarts E.H.L., *Parallel Local Search*, Journal of Heuristics 1 (1995), 43–65.

[261] Voss S., *Tabu search: Applications and prospects*, in: D.Z. Du, P.M. Pardalos (Eds.), Network Optimization Problems, World Scientific (1993).

[262] Wan G., Yen B.P.C., *Tabu search for single machine scheduling with distinct due windows and weighted earliness/tardiness penalties*, European Journal of Operational Research 142 (2002), 271–281.

[263] Wang T.Y., Wu K.B., *An eficient configuration generation mechanism to solve job shop scheduling problems by the simulated annealing*, International Journal of Systems Science 30(5), (1999), 527–532.

[264] Wang C., Chu C., Proth J., *Heuristic approaches for $n/m/F/\Sigma C_i$ scheduling problems*, European Journal of Operational Research 96 (1997), 636–644.

[265] Weinert K., Mehnen J., Rudolph G., *Dynamic neighborhood structures in parallel evolution strategies*, Complex Systems 13(3), (2002), 227–244.

[266] Wrocław Centre of Networking and Supercomputing, www.wcss.wroc.pl

[267] WinTune98, http://www.winmag.com/WinTune98/

[268] Wodecki M., *A branch-and-bound parallel algorithm for single-machine total weighted tardiness problem*, The International Journal of Advanced Manufacturing Technology 37(9–10), (2008), 996–1004.

[269] Wodecki M., Bożejko W., *Solving the flow shop problem by parallel simulated annealing*, Lecture Notes in Computer Science No. 2328, Springer (2002), 236–247.

[270] Wodecki M., *Agregation methods in discrete optimization problems* (in Polish), Monographs series, Wroclaw University of Technology Publishing House, Wrocław 2009.

[271] Wolpert D.H., Macready W.G., *No Free Lunch Theorems for Optimization*, IEEE Trans. Evolutionary Computation 1(1), (1997), 67–82.

[272] Yamada T., Nakano R., *A genetic algorithm applicable to large-scale job shop problems*, in: R. Manner, B. Manderick (Eds.), Parallel problem solving from nature II. Amsterdam: North-Holland, (1992), 281–290.

[273] Yamada T., Reeves C.R., *Solving the C_{sum} Permutation Flowshop Scheduling Problem by Genetic Local Search*, IEEE International Conference on Evolutionary Computation (1998), 230–234.

[274] Yano C.A., Kim Y.D., *Algorithms for a class of single machine weighted tardiness and earliness problems*, European Journal of Operational Research 52 (1991), 167–178.

List of Tables

List of Figures

Index

Nowa klasa równoległych algorytmów szeregowania

Rozwój metod optymalizacji, szczególnie w zastosowaniu do rozwiązywania problemów szeregowania zadań produkcyjnych, sprowadzających się w ogromnej większości do zagadnień silnie NP-trudnych, przebiegał od początku istnienia tej dziedziny w latach 60–70-tych XX wieku w kierunku tworzenia coraz bardziej efektywnych algorytmów implementowanych w środowisku obliczeń sekwencyjnych (jednoprocesorowych). Pod koniec lat 70-tych XX wieku hitem wśród metod optymalizacji kombinatorycznej była metoda podziału i ograniczeń (B&B) uważana wtedy za remedium na prawie wszystkie kłopoty związane z NP-trudnością oraz rozmiarem problemów, których nie można było rozwiązać poprzez przegląd wyczerpujący rozwiązań. Szybko okazało się jednak, że metoda B&B jedynie przesunęła wzwyż praktyczną granicę rozmiaru rozwiązywalnych problemów, jak się faktycznie okazało jednak tylko nieznacznie (np. dla sumo-kosztowego problemu jednomaszynowego rozmiar ten zwiększył się z 20 zadań do 40–50). Co więcej, koszt obliczeń niezbędnych do uzyskania rozwiązania optymalnego okazał się ostatecznie nieuzasadnienie wysoki w porównaniu z zyskami ekonomicznymi i zasadnością jego wykorzystania w praktyce. Konkluzją z tych badań było precyzyjne ustalenie ograniczonego obszaru stosowalności schematu B&B. Począwszy od lat 80-tych XX wieku, nastąpił wyraźny zwrot w kierunku metod przybliżonych (aproksymacyjnych). Początkowo poszukiwanie algorytmów realizowanych w środowisku obliczeń sekwencyjnych, gwarantujących wysoką jakość rozwiązania kosztem zwiększonego czasu obliczeń (w tym także złożonych schematów aproksymacyjnych) zaowocowało szeregiem znaczących rezultatów teoretycznych, które jednak ostatecznie nie odegrały istotnej roli w praktyce. Ten znaczący teoretycznie kierunek w ostatnich latach zanika w sposób naturalny z powodu trudności w uzyskaniu istotnych oszacowań teoretycznych dla problemów występujących w realnych warunkach oraz małej praktycznej przydatności (zgrubności) oszacowań. Kolejnym przełomem było pojawienie się w latach 70–80-tych ubiegłego wieku zaawansowanych metod metaheurystycznych o bardzo dobrych cechach numerycznych. Najpierw – rozwiniętej teorii symulowanego wyżarzania, a następnie algorytmów genetycznych i poszukiwania z zabronieniami (tabu search). Entuzjazm dla tych podejść trwał znacznie dłużej. Do połowy pierwszego dziesięciolecia naszego wieku zaproponowano kilkadziesiąt typów metaheurystyk, realizowanych w środowiskach obliczeń sekwencyjnych. Mniej więcej po roku 2000 metody te osiągnęły kres swych możliwości; rozmiar efektywnie rozwiązywalnych problemów (tj. takich, dla których średni błąd w odniesieniu do rozwiązań optymalnych był np. mniejszy niż 1%) można było przesunąć do liczby idącej w tysiące, jednak w miliony czy setki milionów – już nie. Kropkę nad „i" postawiło twierdzenie „no-free-lunch" autorstwa Wolperta i Macready'ego, które w odniesieniu do metod przybliżonych można parafrazować jako: „bez użycia specjalnych

własności badanych problemów nie można uzyskać znaczącej przewagi jednej metaheurystyki nad drugą". Co ciekawe, Wolpert i Macready pokazali, że przewagę tę można uzyskać w metaheurystykach koewolucyjnych, wielokulturowych, a więc w naturalny sposób równoległych. Idąc za tą ideą, od połowy lat osiemdziesiątych XX wieku równocześnie rozwijały się równoległe, wielowątkowe metaheurystyki, najpierw jako proste zrównoleglenie najbardziej czasochłonnych elementów algorytmów sekwencyjnych (zwykle wyznaczanie funkcji celu), później, od końca lat dziewięćdziesiątych XX wieku, jako tzw. metody wielościeżkowe (tzn. poszukiwania wielowątkowe i rozproszone). Znaczny skok jakościowy projektowanych algorytmów pojawił się w chwili, gdy producenci powszechnego sprzętu komputerowego zorientowali się, że dalsze zwiększanie prędkości (częstotliwości taktowania zegara) w celu zwiększenia mocy obliczeniowej procesorów jest bardzo kosztowne i znacznie łatwiej można uzyskać zwiększenie mocy obliczeniowej stosując konstrukcje wielordzeniowe, stanowiące w naturalny sposób środowisko obliczeń równoległych (i w tym kontekście wśród producentów hardware'u także istnieje pojęcie „no-free-lunch"). Dziś popularne procesory takich producentów, jak Intel czy AMD mają po 4 rdzenie (niektóre procesory Intela – 9 rdzeni, a prototypy nawet 80 rdzeni), a procesory GPU (Graphic Processing Unit) służące początkowo jako procesory wyłącznie graficzne, a dziś już także stricte obliczeniowe, posiadają nawet 960 procesorów (jak np. produkty serii nVidia Tesla). Zderzenie dotychczasowych osiągnięć teorii szeregowania ze zwiększonymi możliwościami technologii obliczeniowej doprowadziło do uświadomienia sobie ograniczeń teorii wynikających głównie z sekwencyjnego charakteru obliczeń stosowanych dotychczas. Zwiększenie liczby rdzeni wymaga zastosowania specjalnie projektowanych algorytmów. Faktycznie, uruchomienie sekwencyjnego algorytmu metaheurystycznego na procesorze wielordzeniowym zwykle prowadzi do wykorzystania jednego rdzenia, a więc zaledwie cząstki potencjalnych możliwości sprzętu. Specyfika algorytmów optymalizacyjnych oraz procedur wyznaczania kluczowych elementów instancji problemu (np. wartości funkcji celu, która jest zwykle sformułowana w sposób rekurencyjny) powoduje, że automatyczne metody zrównoleglenia obliczeń zupełnie się nie sprawdzają. Potrzebne są algorytmy wyspecjalizowane, zaprojektowane specjalnie do uruchomienia w środowisku obliczeń równoległych dla konkretnych typów problemu, wykorzystujące zarówno specyficzne własności problemu, jak i środowiska obliczeń. Niniejsza monografia obejmuje zagadnienia projektowania algorytmów optymalizacji obciążenia maszyn oraz szeregowania zadań produkcyjnych w dyskretnych systemach wytwarzania, wykorzystujących równocześnie specyficzne własności problemu jak i środowiska obliczeń równoległych, w celu uzyskania metod o niespotykanych dotychczas, dobrych własnościach numerycznych.

Cel naukowy monografii. Postawione cele obejmują od strony metodologii następujące zagadnienia:

- Opracowanie nowych klas algorytmów metaheurystycznych poprzez zaproponowanie i zbadanie własności odpowiednich problemów szeregowania zadań i następnie wykorzystania tych własności w konstrukcji zarówno równoległych wersji znanych metaheurystyk, jak i nowych algorytmów typu populacyjnego.

- Implementacja zaproponowanych algorytmów równoległych i rozproszonych dla szerokiej klasy architektur programowania współbieżnego, m.in. GPGPU (General Purpose Graphic Processing Unit), procesorów wielordzeniowych oraz klastrów obliczeniowych.

- Zbadanie własności zaproponowanych algorytmów, a w szczególności ich efektywności w kontekście kosztowej optymalności (tj. osiągnięcia kosztu obliczeń tego samego rzędu co koszt wykonania algorytmu sekwencyjnego) oraz teoretycznego przyspieszenia.

- Opracowanie metod zrównoleglania algorytmów dokładnych na przykładzie metody podziału i ograniczeń (branch and bound, B&B). Szczegółowe cele wymienione powyżej należą do ogólniej sformułowanego zagadnienia, jakim jest opisanie nowej dziedziny, tzn. klasy algorytmów wielowątkowych (tak równoległych, jak i rozproszonych) rozwiązywania NP-trudnych problemów szeregowania zadań produkcyjnych.

Część z proponowanych metod można prawie bez zmian przenieść także na szerszą klasę bardzo trudnych zagadnień optymalizacji dyskretnej, takich jak np. problem komiwojażera (TSP), kwadratowy problem przydziału (QAP), czy problem rozmieszczenia blokowego. W szczególności, w przypadku wielowątkowych algorytmów poszukiwań jednościeżkowych (single-walk parallelization, o takiej samej trajektorii analizowanych rozwiązań jak trajektoria algorytmu sekwencyjnego) zaproponowane zostaną nowe oryginalne metody zrównoleglenia wyznaczania wartości funkcji celu oraz równoległego wyznaczania otoczenia. Rozpatrywane będą problemy: jednomaszynowe, przepływowe (flow shop), gniazdowe (job shop) oraz elastyczne problemy gniazdowe, z maszynami równoległymi (flexible job shop). Szczególnie ten ostatni przypadek, będący uogólnieniem klasycznego problemu gniazdowego, jest często spotykany w praktyce podczas modelowania zagadnień np. w budownictwie oraz organizacji produkcji. W zakresie wielowątkowych algorytmów wielościeżkowych (multiple-walk parallelization) zaproponowane zostaną oryginalne metody rozwiązywania jedno- i wielomaszynowych klas problemów szeregowania zadań poprzez wielowątkowe algorytmy

oparte na metodach: poszukiwania z zabronieniami (tabu search), symulowanego wyżarzania (simulated annealing), poszukiwania rozproszonego (scatter search), algorytmu ewolucyjnego (evolutionary algorithm) oraz populacyjnego (population-based metaheuristic), a także algorytmu genetycznego (genetic algorithm) i ich wielowątkowych wersji hybrydowych. W algorytmach hybrydowych wykorzystywana będzie wielowątkowość nisko- i wysokopoziomowa, tj. zarówno na poziomie najbardziej czasochłonnych elementów algorytmu, jak i na poziomie zwielokrotnienia instancji procesów poszukiwań (wątków). Opracowanie metod zrównoleglania algorytmów dokładnych (np. metody B&B) ma na celu nie tyle stworzenie narzędzia do rozwiązywania problemów szeregowania zadań, ile zaproponowanie algorytmu dokładnego wykorzystującego architekturę równoległą, mogącego służyć do porównywania wyników metaheurystyk dla małych instancji problemów z rozwiązaniami dokładnymi. Równoległe algorytmy dokładne mogą mieć także zastosowanie w rozwiązywaniu problemów cyklicznych szeregowania zadań.

Obecny stan wiedzy. Pomimo znaczącego w ostatnich latach rozwoju teorii algorytmów oraz teorii optymalizacji, algorytmy heurystyczne wciąż pozostają często jedyną drogą dla uzyskania rozwiązań, które są zadawalające z punktu widzenia praktyki, zarówno co do rozmiaru rozwiązywanych w rozsądnym czasie przykładów, jak i dobroci (odległości od rozwiązania optymalnego) otrzymanych wyników. Zdecydowanie krótszą historię mają metody obliczeniowe wykorzystujące komputery wieloprocesorowe, choć klasycznego już dziś podziału architektur tych komputerów dosyć dawno dokonał Flynn [108]. Faktycznie dopiero w latach osiemdziesiątych ubiegłego wieku pojawiły się konstrukcje szybkich algorytmów równoległych.

Metaheurystyki oparte na metodzie lokalnych poszukiwań mogą być przedstawione jako procesy przeszukiwania grafu, w którym wierzchołkami są punkty przestrzeni rozwiązań (np. permutacje), a łuki odpowiadają relacji sąsiedztwa – łączą wierzchołki będące rozwiązaniami sąsiednimi w tej przestrzeni. Poruszanie się po takim grafie wyznacza pewną drogę (trajektorię). Wielowątkowe algorytmy metaheurystyczne korzystają z wielu wątków, zwykle uruchomionych na oddzielnych procesorach bądź rdzeniach, do współbieżnego generowania lub przeglądania grafu.

Można wyróżnić dwa podejścia do zrównoleglania procesu lokalnego poszukiwania, w zależności od liczby trajektorii generowanych współbieżnie w grafie sąsiedztwa.

1. Pojedyncza trajektoria: algorytmy drobno- i średnioziarniste.
2. Wiele trajektorii: algorytmy średnio- i gruboziarniste.

Podejścia te stawiają przed algorytmem pewne wymagania dotyczące częstotliwości komunikacji, co implikuje rodzaj ziarnistości. Algorytmy drobnoziarniste odpowiadają podejściu z częstszą komunikacją, gruboziarniste – z rzadszą.

Algorytmy jednościeżkowe. Algorytmy jednościeżkowe generują pojedynczą trajektorię, jednak mogą to czynić współbieżnie poprzez podział procesu badania otoczenia na kilka procesorów, z których każdy bada pewną część otoczenia, szukając najlepszego elementu. Idea ta została zaproponowana najwcześniej dla sekwencyjnych algorytmów poszukiwań, patrz Nowicki i Smutnicki [196] pod nazwą metody reprezentantów (representatives). Pochodzenie nazwy jest ściśle związane z działaniem metody, bowiem z każdej części otoczenia zostaje wybrany reprezentant, a dopiero spośród nich najlepszy reprezentant jako następny punkt trajektorii poszukiwań. Odpowiedniki równoległe metody reprezentantów pojawiły się w literaturze później.

Algorytmy wielościeżkowe. Algorytmy, w których konstrukcji wykorzystano model wielościeżkowy badają współbieżnie przestrzeń rozwiązań za pomocą równolegle działających wątków poszukiwań. Algorytmy te można dodatkowo podzielić na podklasy ze względu na wymieniane informacje o aktualnym stanie poszukiwań:

1. Niezależne procesy poszukiwań.
2. Kooperujące procesy poszukiwań.

W przypadku, gdy współbieżnie działające procesy poszukiwań nie wymieniają pomiędzy sobą żadnych informacji, mówimy o niezależnych (independent) procesorach poszukiwań. Jeśli zaś informacja uzyskana w trakcie eksploracji trajektorii przez proces poszukiwań jest przekazywana innemu procesowi, a następnie wykorzystywana przez ten procesor, to można mówić o procesach kooperujących (cooperative). Spotykany jest także model mieszany, tzw. pół-niezależny (semi-independent) [9], wykonujący niezależne procesy poszukiwań przy zachowaniu pewnych wspólnych parametrów.

Równoległe obliczenia dla jednej trajektorii. Jest to metoda służąca do przyspieszenia przeszukiwania grafu sąsiedztwa poprzez zrównoleglenie najbardziej czasochłonnych operacji – czyli obliczania wartości funkcji celu, bądź zrównoleglenie procesu generowania sąsiadów. W przypadku zrównoleglania obliczania wartości funkcji celu przyspieszenie obliczeń może być uzyskane przy zachowaniu identycznej trajektorii przejścia przez graf, jak trajektoria algorytmu sekwencyjnego. W drugim przypadku – dekompozycji generowania otoczenia na procesory równoległe – zaistnieć może sytuacja, w której algorytm, sprawdzając równolegle większą liczbę sąsiadów niż to czyni wersja sekwencyjna (najczęściej zaopatrzona w mechanizm redukcji rozmiarów otoczenia), poruszać się będzie po trajektorii lepszej niż sekwencyjny odpowiednik, wyznaczając korzystniejszą trasę przejścia przez graf i tym samym dochodząc do lepszych rezultatów obliczeń (wartości

funkcji celu). Pierwsze aplikacje bazujące na opisywanym modelu pojawiły się w kontekście zrównoleglenia metody symulowanego wyżarzania i algorytmu genetycznego. Chociaż równoległa dekompozycja sąsiedztwa nie zawsze prowadzi do redukcji czasu obliczeń, jest jednak często stosowana do zwiększania rozpatrywanego sąsiedztwa. Tego typu algorytm równoległy tabu search dla problemu komiwojażera został zaproponowany przez Fiechtera [106]. Synchroniczny tabu serach był także badany przez Porto i Ribeiro [209]. Bożejko, Pempera i Smutnicki [39] zaprezentowali równoległe podejście jednościeżkowe w rozwiązywaniu problemu przepływowego. Aarts i Verhoeven [1, 260] różnicują klasę jednościeżkowych algorytmów równoległego przeszukiwania na dwie podklasy. Klasa jednokrokowa (single-step) obejmuje algorytmy, w których badanie otoczenia jest dzielone pomiędzy równoległe procesory, ale jako wynik wybierany jest jeden ruch. Z kolei w klasie wielokrokowej (multiple-step) sekwencja kolejnych ruchów w grafie sąsiedztwa jest wykonywana współbieżnie.

Równoległe obliczenia dla wielu trajektorii. Implementacje algorytmów opartych na równoległym wielościeżkowym przeszukiwaniu przestrzeni rozwiązań są aplikacjami gruboziarnistymi, czyli wymagającymi rzadkiej komunikacji. Są one łatwiejsze w zastosowaniu w systemach rozproszonych, jak na przykład klastrach komputerów klasy PC, dysponujących korzystnym wskaźnikiem ilorazu mocy obliczeniowej do ceny. Oprócz przyspieszenia obliczeń, uzyskać można także poprawę jakości otrzymywanych rozwiązań. Procesy poszukiwań mogą być niezależne lub kooperujące.

Niezależne procesy poszukiwań. W tej kategorii rozróżnić możemy dwa podstawowe podejścia:

1. Przeszukiwanie przestrzeni rozwiązań za pomocą wielu trajektorii. Każdy z procesorów startuje z innego rozwiązania początkowego (lub różnych populacji w przypadku algorytmu genetycznego). Wątki poszukiwań mogą stosować ten sam lub różne algorytmy lokalnego poszukiwania, z takimi samymi lub różnymi wartościami parametrów strojących (np. długość listy tabu, wielkość populacji, itp.). Trajektorie mogą się przecinać w jednym lub wielu miejscach grafu sąsiedztwa.
2. Równoległe badanie podgrafów grafu sąsiedztwa wyznaczonych przez dekompozycję problemu na kilka podproblemów (np. przez ustalenie pewnych zmiennych). Podgrafy grafu sąsiedztwa są badane współbieżnie bez przecinania się trajektorii. Otrzymujemy w tym przypadku całkowitą dekompozycję grafu sąsiedztwa na rozłączne podgrafy.

Pierwsza równoległa implementacja algorytmu tabu opartego na wielościeżkowym badaniu przestrzeni rozwiązań została zaproponowana przez Taillarda i dotyczyła kwadratowego zagadnienia przydziału (QAP) [244] oraz problemu

gniazdowego (job shop) [245]. Zrównoleglenie algorytmu genetycznego z użyciem niezależnych wątków poszukiwań nawiązuje do tak zwanego modelu wyspowego, bez komunikacji pomiędzy podpopulacjami zamieszkującymi poszczególne wyspy (Bubak i Sowa [68]). Chociaż zauważono pewne przyspieszenie, nie otrzymano poprawy wyznaczanych w ten sposób rozwiązań w stosunku do wyników sekwencyjnego algorytmu genetycznego z jedną dużą populacją. Fakt ten można wytłumaczyć szybką stagnacją podpopulacji (brakiem dalszej poprawy średniej wartości funkcji celu po pewnej liczbie wykonanych iteracji) na każdym z procesorów pozbawionym komunikacji z pozostałymi.
Kooperujące procesy poszukiwań. Model ten jest najogólniejszym i najbardziej obiecującym typem strategii przeszukiwania przestrzeni rozwiązań przez równoległy algorytm metaheurystyczny. Wymaga jednak większej wiedzy programistycznej i znajomości specyfiki rozwiązywanego problemu. Kooperacja oznacza w tym wypadku wymianę informacji – doświadczeń dotyczących dotychczasowego procesu przeszukiwania przestrzeni przez równoległe procesy. Wymieniać należy specyficzne informacje, charakterystyczne dla problemu i metody, np. najlepsze znalezione rozwiązania, rozwiązania elitarne (mało różniące się od najlepszych znanych), częstotliwości ruchów, listy tabu, podpopulacje i ich rozmiary i inne. Pierwszym tego typu algorytmem heurystycznym był asynchroniczny algorytm tabu przedstawiony przez Crainic'a, Toulouse i Gendreau [85]. Większość implementacji kooperującego algorytmu genetycznego bazuje na migracyjnym modelu wyspowym. Każdy z procesorów posiada swoją własną podpopulację wymieniając co pewną liczbę iteracji osobniki (zwykle najlepsze) z pozostałymi procesorami [38]. Bubak i Sowa [68] zastosowali migracyjny model wyspowy w równoległym algorytmie genetycznym dla problemu komiwojażera (TSP) uruchamianym na komputerze HP/Convex Exemplar SPP1600 z 16 procesorami oraz na klastrze heterogenicznym. Bożejko [26] zaproponował równoległy algorytm ścieżek łączących, bazujący na równoległej metodzie poszukiwania rozproszonego.

Metodyka badań. W zakresie poszukiwań jednowątkowych, dedykowanych dla jednorodnych systemów wieloprocesorowych takich jak GPU, zaproponowany zostanie szereg oryginalnych metod uwzględniających odmienne techniki projektowania algorytmów równoległych oraz różne potrzeby zgłaszane przez nowoczesne algorytmy optymalizacji dyskretnej (równoległe wyznaczanie wartości funkcji celu, analiza lokalnych otoczeń, itp.). Szczególna uwaga zostanie zwrócona na problemy efektywności, kosztu oraz przyspieszenia obliczeń w zależności od typu problemu, jego wielkości oraz zastosowanego środowiska obliczeń równoległych. Dla proponowanych algorytmów przeprowadzona zostanie analiza porównawcza korzyści wynikających z zastosowania odpowiednich podejść.

W zakresie poszukiwań wielowątkowych, dedykowanych zarówno dla jednorodnych, jak i niejednorodnych systemów wieloprocesorowych (takich jak duże komputery typu mainframe, klastry, gridy) zaprojektowane zostaną i przebadane eksperymentalnie warianty wielowątkowe najbardziej obiecujących aktualnie metod optymalizacji kombinatorycznej (poszukiwania tabu, symulowanego wyżarzanie, metod populacyjnych, poszukiwania rozproszonego, a także schematu B&B) w zastosowaniu do wybranych problemów szeregowania zadań. Szczególny nacisk zostanie położony na zbadanie zjawiska przyspieszenia ponadliniowego (superlinear speedup), którego pojawianie się zasygnalizowano wielokrotnie (m.in. w opracowaniu Alba [7] dotyczącym równoległych metaheurystyk).

The book contains a wealth of infor
readers, including advanced students
sionals working in the field of discrete optimization and management. A new methodology of solving strongly NP-hard real-world job scheduling problems is presented here. It allows us to design very efficient and fast approximate and exact algorithms for solving a wide class of discrete optimization problems, not only scheduling problems. Efficiency of the present research has been proved by comprehensive computational experiments conducted in parallel processing environments such as supercomputers, clusters of workstations, multi-core CPUs and GPUs.

Wydawnictwa Politechniki Wrocławskiej
są do nabycia w księgarni „Tech"
plac Grunwaldzki 13, 50-377 Wrocław
budynek D-1 PWr., tel. 71 320 29 35
Prowadzimy sprzedaż wysyłkową
zamawianie.ksiazek@pwr.wroc.pl

ISBN 978-83-7493-564-7