CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

Parallel Local Search to Schedule Communicating Tasks on Identical Processors

Tatjana Davidović
Teodor Gabriel Crainic

August 2013

CIRRELT-2013-54

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
2325, de la Terrasse, bureau 2642
Québec (Québec)
Canada G1V 0A6
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

Université Laval
UQÀM
HEC MONTRÉAL
École Polytechnique Montréal
Université de Montréal

Parallel Local Search to Schedule Communicating Tasks on Identical Processors

Tatjana Davidović[1], Teodor Gabriel Crainic[2,*]

[1] Serbian Academy of Science and Arts, Kneza Mihaila 36, P.O. Box 367, 11001 Belgrade, Serbia

[2] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT) and Department of Management and Technology, Université du Québec à Montréal, P.O. Box 8888, Station Centre-Ville, Montréal, Canada H3C 3P8

Abstract. The paper reports on the analysis of parallelization strategies for Local Search (LS) when the neighborhood size varies throughout the search, the Multiprocessor Scheduling Problem with Communication Delays (MSPCD) illustrating the methodology and results. The dynamic load distribution strategy implemented within a master-slave framework is shown to offer the best performance. Experimental results on several sets of instances with up to 500 tasks show excellent speedups (super-linear in most cases) while preserving the quality of the final solution. The proposed parallel LS is incorporated into Multistart Local Search and Variable Neighborhood Search meta-heuristic frameworks to analyze its efficiency in a more complex environment. The comparison between the sequential and parallel versions of each meta-heuristic, using various numbers of processors, shows improvement in the solution quality within proportionally smaller CPU time.

Keywords: Distributed memory, multiprocessors, load balancing, neighborhood decomposition, directed acyclic graph, meta-heuristics.

Acknowledgements. This work has been partially supported by the Serbian Ministry of Science, Grant nos. 174010 and 174033. Funding was also been provided by the Natural Sciences and Engineering Council of Canada (NSERC), through its Discovery Grant program. The authors would like to thank Mr. François Guertin for his unlimited help with parallel programming and executions and Calcul Québec and Compute Canada for the computational resources.

Results and views expressed in this publication are the sole responsibility of the authors and do not necessarily reflect those of CIRRELT.

Les résultats et opinions contenus dans cette publication ne reflètent pas nécessairement la position du CIRRELT et n'engagent pas sa responsabilité.

* Corresponding author: Teodor-Gabriel.Crainic@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2013

© Copyright Davidović, Crainic and CIRRELT, 2013

1 Introduction

Local Search (*LS*) is a well-known methodological tool for finding good suboptimal solutions for combinatorial optimization problems. It is widely used to improve the quality of solutions obtained by constructive heuristics. More importantly, it serves as a basic building block for a wide variety of meta-heuristic methods, neighborhood-based ones in particular. Although meta-heuristics have proven efficient for many combinatorial optimization problems and real-life applications, there are still problems that cannot be treated appropriately in a reasonable amount of time due to their complexity or the large size of the practical instances, or both. As the recent literature shows [6], the parallelization of search procedures offers a promising approach to increase the efficiency of heuristic and meta-heuristic methods. LS is generally computationally intensive and one wants it to be very efficient, especially within a meta-heuristic framework where it will be called upon very frequently. Parallel strategies are very promising in this respect and, yet, there is a little work dedicated to the parallelization of LS procedures. The main objective of this paper is to propose efficient parallelization strategies for the LS procedure.

We address the parallelization of LS operating in a solution environment comprised of neighborhoods of variable sizes. To the best of our knowledge, there are no papers dealing with this problem in the recent literature. We show that any static parallelization of such a highly dynamical search process yields an inefficient utilization of available resources. Our approach, involving the dynamic fine-grained partition of the neighborhood, leads to an efficient load balanced parallel execution of LS. Moreover, we show that the proposed parallel LS highly increases the performance of the corresponding neighborhood-based meta-heuristics.

We illustrate these strategies and show results through an application of parallel LS to the *Multiprocessor Scheduling Problem with Communication Delays* (*MSPCD*). The MSPCD is an essential problem not only in computer science, but also in various applications involving scheduling of multiple resources (robotics, aircraft control, etc.). It was shown in [11] how hard may be to find good solutions for this problem, even with meta-heuristic methods. Therefore, we aim to provide an efficient method for improving the existing scheduling results for MSPCD. As also shown in [11], LS takes the main part of the computational burden in meta-heuristic search, 99% of the execution time was devoted to constructing and evaluating neighbors. We show in this paper the benefits of varying the neighborhood size in MSPCD and experimentally verify that fixed neighborhood partitions perform badly. The experimental evaluation underlines the usefulness of our approach in addressing the MSPCD.

The main contributions of this paper are 1) identifying and analyzing the issues related to variable neighborhood dimensions, for which static decomposition is very inefficient as a parallelization strategy of the corresponding LS; 2) the development of appropriate

parallelization strategies for this type of LS procedure; 3) the experimental evaluation of various parallel LS strategies; and 4) improving the best known results for the MSPCD. Note however that, the proposed parallelization strategy does not depend on the particular application and can be easily adapted to different combinatorial optimization problems. Moreover, our efficient implementation of the selected parallelization strategy on a distributed-memory multiprocessor architecture using the MPI communication library supports the conclusion of Jin *et al.* [17] stating that “MPI is a de facto standard for parallel programming (especially on distributed memory systems) assuring achievable performance and portability”.

The paper is organized as follows. Section 2 recalls the formulation of the MSPCD and the sequential, permutation-based LS procedure. The proposed parallelization strategies are described in Section 3, while the implementation details are given in Section 4. Section 5 contains the results of extensive experimentation with the proposed parallel LS procedures. The experimental evaluations of VNS and MLS with the selected parallel LS procedure are described in Section 6. We conclude in Section 7.

2 MSPCD Definition and Sequential LS

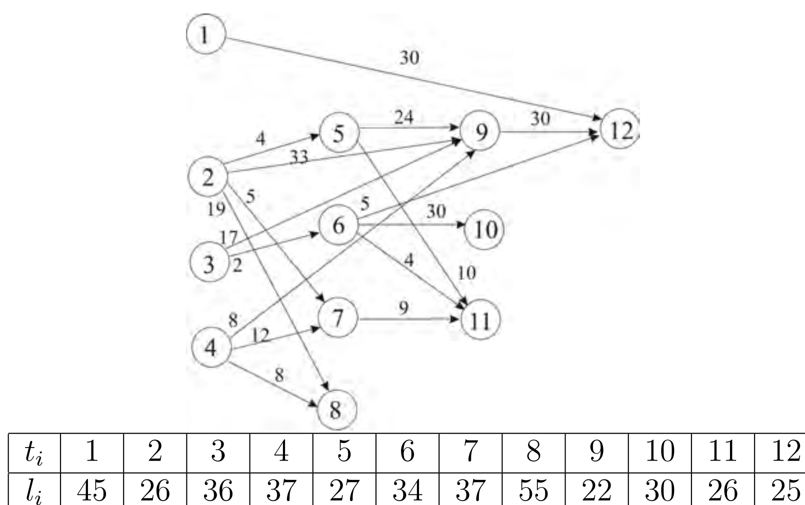
The Multiprocessor Scheduling Problem with Communication Delays can be described as follows: tasks (or jobs) have to be executed on a multiprocessor system containing several identical processors; we have to decide where and when each task will be executed, such that the total completion time (makespan) is minimum. The duration of each task is known as well as the precedence relations among tasks, i.e., what tasks should be completed before some other could begin. In addition, if dependent tasks are executed on different processors, the data transferring time (or communication delay) is also known. Task preemption and duplication (redundant executions) are not allowed.

In the next subsection, we recall the formulation of the MSPCD for which the sequential LS procedure was proposed. The two mathematical programming formulations, the first one based on task ordering and the second one inspired by rectangles packing are detailed in [10]. The second subsection is dedicated to a brief overview of the sequential LS proposed in [11], including the data structures and neighborhood definitions. We are thus introducing the definitions used to present the parallel LS strategies we propose.

2.1 Multiprocessor Scheduling with Communication Delays

The tasks to be scheduled are represented by a directed acyclic graph (DAG) [8, 20, 31], called *Task Graph* (TG), defined as a tuple $\mathcal{G} = (T,\ L,\ E,\ C)$. Here $T = \{t_1, \ldots, t_n\}$

stands for the set of tasks; $L = \{l_1, \ldots, l_n\}$ represents the computation times of the tasks (execution times, lengths, durations); $E = \{e_{ij} \mid t_i, t_j \in T\}$ describes the set of communication edges; and $C = \{c_{ij} \mid e_{ij} \in E\}$ is the set of edge communication costs. The set E defines precedence relations between tasks. A task cannot start its execution unless all its predecessors are completed and all relevant data is available. The communication cost $c_{ij} \in C$ represents the amount of data transferred between tasks t_i and t_j when they are executed on different processors. When both tasks are scheduled to the same processor no communication is required. An example of the task graph containing 12 nodes (tasks) is given in Fig. 1. The numbers within the circles represent the task indexes. Weights on the task-graph edges represent the communication cost. Task durations are listed in the table presented below the task graph.



t_i	1	2	3	4	5	6	7	8	9	10	11	12
l_i	45	26	36	37	27	34	37	55	22	30	26	25

Figure 1: Task graph example

The multiprocessor architecture $\mathcal{M}$ is assumed to contain p identical processors with their own local memories, communicating by exchanging messages through bidirectional links of the same capacity. This architecture is modeled by a *distance matrix* [8, 12]. The element $(i,\ j)$ of the distance matrix $D = [d_{ij}]_{p \times p}$ is equal to the minimum distance between the processors i and j calculated as the number of links along the shortest path between the two processors. The distance matrix is therefore symmetric with zero diagonal elements. The multiprocessor system containing 4 processors connected in a ring architecture and the corresponding distance matrix are presented in Figure 2.

The scheduling of TG $\mathcal{G}$ onto $\mathcal{M}$ consists in determining the index of the associated processor and starting time instant for each of the tasks from TG in such a way as to minimize some objective function. The usual objective function (that we use in this paper as well) is the completion time C_{max} [4] of the scheduled task graph (referred to as *makespan* or schedule length) $C_{max} = \max_{i \in T}\{S_i + l_i\}$. The starting time S_i of a task t_i depends on the completion times of its predecessors and the amount of required

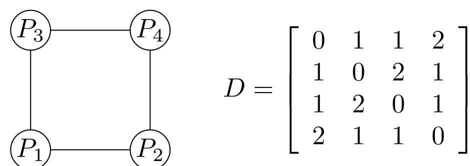


Figure 2: 2-dimensional hypercube multiprocessor architecture

communication. The communication time between tasks t_i assigned to processor p_k and t_j assigned to processor p_l is calculated as $\gamma_{ij}^{kl} = c_{ij} \cdot d_{kl} \cdot ipc$, where ipc represents the inter-processor-communication speed (the communication-to-computation-ratio (CCR) from [23]). If $l = k$, then $d_{lk} = 0$ implying $\gamma_{ij}^{kl} = 0$.

Since MSPCD is known to be NP-hard, only a few researchers have developed exact algorithms [23, 32] for solving moderate-size instances. The classical approach to this problem involves developing heuristic methods, which can give suboptimal solution of good quality [12, 20, 21, 31]. These heuristic methods are constructive, building the solution from a particular problem representation. Most of of these heuristics are based on the list-scheduling technique, which usually consists of two steps [see 24, 30, for scheduling techniques using different approaches] The first step orders the tasks in a priority list, and the second one selects the most suitable processor for each task following the order of the list. The most widespread task ordering rule is based on the *Critical Path* (CP) [8, 18, 21]. Regarding the processor selection strategy, we point out the *Earliest Start* (ES) selecting the processor with minimum value of S_i [27]. The two-step heuristics offer a very suitable environment for LS and meta-heuristic search procedures. Different lists of tasks enable the definition of transformation rules from one list to another, which is the initial step in defining neighborhoods and search strategies based on this solution representation.

Different meta-heuristics were proposed to improve the heuristic solutions, sequential [28] and parallel [29] Tabu Search (TS), parallel Genetic Algorithm (GA) [22], sequential permutation-based GA, TS, MLS and VNS [11]. The last paper performed an in-depth analysis of MSPCD and showed that, for the benchmark instances with known optimal solutions from [9], even the best performing meta-heuristic (VNS) produced solutions that were sometimes more than ten percent worse then the optimum. This is an indication that improvements can be obtained with parallelization.

2.2 Solution representation, neighborhoods and sequential local search

We develop parallel LS procedures starting from the sequential method proposed in [11], using the same solution representation and neighborhood definition. The solution space S is defined as the set of all *feasible permutations* of tasks with respect to the precedence relations. An example of an unfeasible permutation the task graph from Figure 1 is

$$2 \quad 5 \quad 9 \quad 1 \quad 3 \quad 4 \quad 6 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12$$

where task 9 appears before its predecessor 3. Considering such an unfeasible permutation can lead to schedules that will not execute correctly on the target multiprocessor architecture. The same solution representation was used in [8, 22, 19]. Changing the permutations,one obtains different lists of tasks for scheduling. We use the ES scheduling rule [8] to compute the makespan objective function value, which is to be minimized, as it provided the best performance in previous works [11, 9].

Given the solution representation, there is an indirect connection only between the solution space and the value of the objective function. Therefore, it is not straightforward to update the objective function value following a solution modification. It is actually generally necessary to perform the complete rescheduling of tasks to processors when a permutation is changed. The complexity of ES is $O(n^2p)$, as one has to calculate the starting time on each processor p for each task n, given the allocation of all its predecessors ($O(n)$). Two tricks were proposed in [11] to minimize the needed calculations and to improve the efficiency of LS. First, the changed part of the solution is identified and only that part is rescheduled. Second, the induced symmetry between solutions (the fact that several different feasible permutations can yield the same final schedule) is taken into account. If there is no change in the schedule for the first task belonging to the changed part, the corresponding neighbor is not considered, since obviously it is leading to the same final schedule.

Representing a MSPCD solution as a feasible permutation of tasks, provides the means to explore well-known several neighborhood structures used in addressing the traveling salesman problem, while imposing the restriction to keep the feasibility of the generated neighbors [11]. In this paper, we use the *Swap-1* neighborhood to define the proposed parallel LS strategies.

A Swap-1 neighbor of a feasible solution x is obtained by moving a task from one position to another (Figure 3). The worst case cardinality of the Swap-1 neighborhood is $O(n^2)$, but the feasibility constraints make it usually much smaller. To further reduce the neighborhood cardinality,citeDHM04 introduced a new search parameter, the *search direction*. The neighborhood is thus searched by moving tasks only to the right (FORWARD

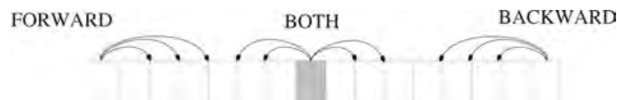


Figure 3: Swap-1 neighborhood search directions

search) or only to the left (BACKWARD *search*) as illustrated in Figure 3.

The LS procedure usually represents a systematic search in the given neighborhood of an initial solution for the "better" solutions. The pseudo-code for the sequential LS is given in Figure 4. The initial solution can be selected randomly or by applying some constructive heuristic. At the beginning, it also represents the current best solution. The LS of Figure 4 adopts the *Best Improvement* (*BI*) search principle according to which all neighbors are visited and evaluated. The LS can be reduced by imposing that only some specified part of the neighborhood is searched. Alternatively, one can perform a *First Improvement* (*FI*) search, i.e., stop the search in the given neighborhood of the current solution as soon as the first improving solution is found.

1. *Initialization.* Choose initial solution x; Set $x_{min} = x$; $f_{min} = f(x)$.
2. **repeat**
    IMPROVEMENT = 0;
    for all $x' \in \mathcal{N}(x_{min})$
        if $(f(x') < f_{min})$ **then**
            $x_{min} = x'$; $f_{min} = f(x')$; IMPROVEMENT = 1;
        endif
   until IMPROVEMENT == 0;
return (x_{min}, f_{min})

Figure 4: Pseudo-code for the sequential local search procedure

It is obvious that the restrictions described above, feasibility constraints, savings in the computations of the schedule length, search direction, and improvement strategy, highly influence the neighborhood size of each particular solution. One must, therefore, account for the fact that the number of neighbors varies throughout the execution of the LS procedure, when designing parallel strategies.

3 Parallel LS Procedures

A significant amount of work was performed regarding the development and analysis of various parallelization strategies for meta-heuristics. A number of main ideas can

be identified in the literature [see the survey papers 6, 7, 33] starting from the low level parallelization realized by distributing neighborhoods among processors, up to the cooperative multi-thread parallel search [6, 5]. In the traditional, classical approach, the main goal of parallelization is to speedup the computations needed to solve a particular problem by engaging several processors and dividing the total amount of work between them. When meta-heuristics are considered, the goals are richer [6]. Parallelization is changing the original algorithm and, therefore, a new class of meta-heuristics is obtained [1]. The non-determinism and the design of parallel search procedures involving several computations proceeding simultaneously interacting according to various mechanisms, yield more robust algorithms that perform a better exploration of the solution space, achieving the desired gains in generating better-quality solution within reduced amounts of running time. The parallelization of LS was rarely studied in details, however. It is, therefore, the object of our work reported in this paper.

Notice that, even when the sequential LS proceeds deterministically, parallelization often transforms it into stochastic search, since the final result generally depends on the number of processors, the distribution of computations, and the communication mechanisms. Two main issues have to be considered therefore when analyzing the efficiency of parallelization strategies: the division of computations and the communication mechanisms. The former is key to a good computation-load balancing among processors and significant overall speedup. With regards to communications, the goal is to minimize them while still exchanging information meaningful for the search. Three main aspects should be considered 1) *what* is communicated; 2) *when* the communications take place; and 3) *where* the information is sent or, more generally, between which tasks/processors communications are taking place. Communication can be synchronous or asynchronous. According to [6], parallel LS with synchronous communication is more similar to the sequential one with respect to directing the search process. The issue with synchronization is the occurrence of idle time intervals if the load of processors is not well balanced. Asynchronous inter-processor communication can mitigate this impact and minimize the idle time intervals, but then the search is performed non-deterministically, yielding a parallel LS that is quite different from the original, sequential one. On the other hand, asynchronous communications generally enable different regions of solution space to be explored, which may result in a better quality of the final solution.

In the rest of this section, we describe several strategies for parallel LS for MSPCD. These strategies can be classified according to [5] as 1C/RS/SPSS and 1C/KS/SPSS (which correspond to the *neighborhood decomposition* class from [7]). A number of papers in the recent literature that applied 1C/RS/SPSS strategies to LS, for example, within Tabu Search [3, 29], Simulated Annealing [34], and Variable Neighborhood Search [13]. To the best of our knowledge, none of the works dealing with parallel LS considered the case when the size of the neighborhood can vary during the search process. This case requires detailed analysis and different approaches to parallelization, and we aim to contribute to address this issue.

3.1 Neighborhood decomposition parallelization strategies

Neighborhood decomposition means the partition of the neighborhood and the exploration of the resulting regions in parallel, each region by a different processor, as illustrated by the block-diagram presented in Figure 5. Each processor examines the neighbors from the associated region within a single iteration of LS. In the general case, this strategy does not change the original sequential algorithm, and aims just to speedup the search process.

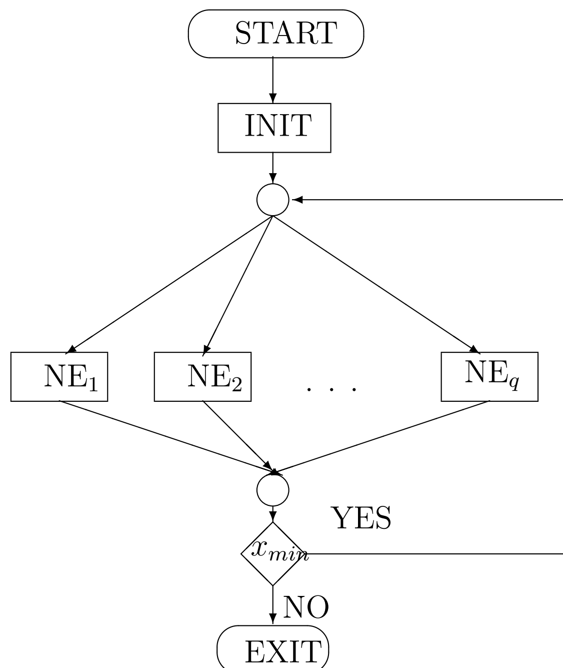


Figure 5: PNE - Neighborhood decomposition parallel LS

We refer to this parallelization as *parallel neighborhood exploration* (*PNE*). From a computational point of view, linear speedup should be expected since these computations are independent. Regarding the communication issues, all the improved solutions (local minima with respect to each region of the neighborhood) should be exchanged among processors at the end of the current iteration and the best one propagated to the next iteration for further exploration. The described PNE falls into the synchronous category, since the communications are performed at strictly defined execution points among all processors. According to the taxonomy proposed in [5], PNE can be classified as 1C/RS/SPSS.

Several PNE strategies may be devised according, in particular, to how the partition is performed. In an *uniform partition* strategy, tasks are equally distributed among

processors. Applying this strategy to the Swap-1 neighborhood (Section 2.2), means to divided the n tasks, ordered within a feasible permutation, into q parts, where q represents the number of available processors involved in the parallelization (Figure 6). The partition depends on the task position (index) in the current feasible permutation. A subset of indexes is associated to each processor that then searches the region of the neighborhood defined by this subset.

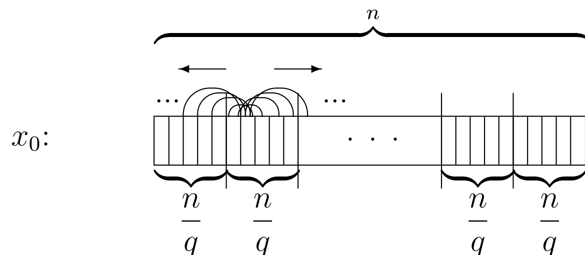


Figure 6: Uniform partition of Swap-1 neighborhood

During the parallel search, each processor moves tasks that are initially in its search range. The new position of a task can be anywhere in the permutation, however, it is not limited by the search range of the particular processor (Figure 6). Therefore, each processor operates on a complete feasible solution, generating and evaluating the new permutations obtained by moving tasks from the associated part only. According to the uniform-partition idea, each region should contain approximately n/q succeeding tasks and, thus, processor r $(r = 1, 2, \ldots, q)$ has to search the interval

$$\left[(r-1) * \left\lceil \frac{n}{q} \right\rceil + 1, r * \left\lceil \frac{n}{q} \right\rceil\right],$$

where $\lceil x \rceil$ indicates the minimal integer grater than or equal to x.

Uniform-partition PNE was used within TS for scheduling dependent tasks onto heterogeneous multiprocessor systems in [29]. The authors noticed an equal distribution of computations among the neighbors of a current solution, each solution had the same number of neighbors, and the complexity of the objective function value calculation was also fixed. Therefore, the neighborhood could be partitioned into equal parts (up to the modulo $n\%q$ factor). With a master-slave multiprocessor architecture containing up to 16 processors, almost linear speedup was reported.

Contrary to [13, 29], the equal partition of neighborhoods is not appropriate in our case. First, the number of neighbors is not the same for all solutions and depends on the improvement-rate strategy (FI or BI). In our experiments, for example, when $n = 200$, the number of neighbors varied between 1860 and 2060 for the BI search.

The deviations were larger for FI where, sometimes, the first visited neighbor yielded the solution improvement and thus its evaluation completed the current LS iteration. The second reason against equal partition is the variable duration of neighbor evaluation (Section 2.2). Moreover, the number of neighbors within each region of the neighborhood does not have to be the same due to the feasibility constraints. Therefore, the appropriate load-balanced partition of the permutation among processors has to be determined. An attempt to generate such a partition is described in the next subsection.

3.2 Variable size neighborhood decomposition

Experiments with the uniform-partition PNE for parallel LS showed that the expected load imbalance took indeed place. The fact that the load imbalance was the consequence of an inappropriate distribution of calculations among processors appeared clearly since we were always able to identify at least one processor where the values of the measured total execution time (t_{tot}) and computation time (t_{CPU}) were almost the same, while for the others the difference between these was significant. This difference contains both the communication and idle times produced by load imbalance. Since the communication time is the same for all processors, it follows that computations were imbalanced. The curves illustrating Load imbalances of FI and BI searches performed by four processors in BACKWARD in a single LS run are illustrated in Figure 7.

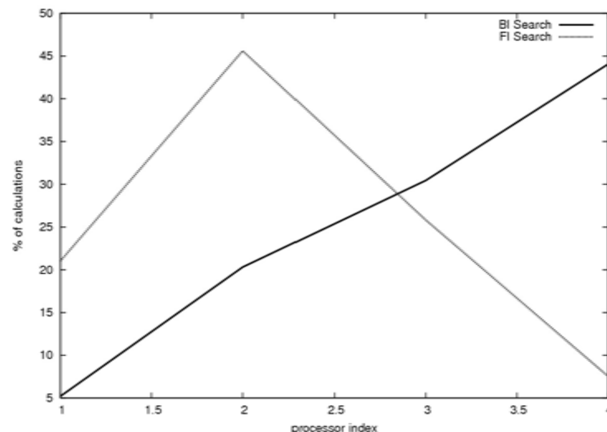


Figure 7: Load imbalance, 4 processors searching in BACKWARD direction

We therefore define the fixed *coarse, variable-size, load-balanced* neighborhood partition, identified as PNEC in the following, which we expect to yield better load balances between processors. Our initial experimental results showed that the task parti-

Table 1: Percentages of tasks allocated to each processor (BACKWARD search)

q	BI search % of tasks										FI search % of tasks									
1	100										100									
2	75	25									50	50								
3	65	18	17								20	20	60							
4	55	20	12	13							17	15	15	53						
5	53	22	11	7	7						10	10	15	15	50					
6	53	21	10	6	5	5					12	8	10	10	15	45				
7	41	18	11	10	8	6	6				12	7	7	10	10	19	35			
8	35	17	13	10	6	8	5	6			7	5	6	7	7	14	21	33		
9	35	12	11	9	6	6	6	8	7		5	3	5	8	9	10	14	18	28	
10	32	12	10	10	6	6	5	7	7	5	5	5	4	5	7	9	9	12	19	25

tion should be different for the cases when neighborhood is explored in FORWARD and BACKWARD directions. An example of the experimentally determined distribution for BACKWARD search for $q \leq 10$ is given in Table 1. The main issue with this distribution is that the percentage of tasks to be handled by each processor has to be determined all over again when q increases. We therefore propose a dynamic PNE strategy in the next subsection.

3.3 Dynamic fine-grained neighborhood decomposition

We propose the *fine-grained, dynamic* partitioning of the permutation and neighborhood, where each processor is given a single task at a time, in the order defined by the feasible permutation, until all the tasks are explored or the FI criterion is satisfied. We identify this strategy as *PNEF* in the following.

We expect PNEF to offer the "ideal" strategy. On the negative side, such a partitioning may result in an significant increase in communications and, therefore, its performance should be measured when used on an arbitrary multiprocessor architecture. Yet, multiprocessor modern machines with highly integrated processors very enable fast communications. This is true even for the somewhat older machine used for the experimentation reported in this paper, a SUN Enterprise 10000, where we observed practically negligible communication times (equal t_{tot} and t_{CPU} in the initial unbalanced tests). For such high-speed communication architectures, the PNEF strategy becomes highly relevant.

The main characteristic of PNEF is its nondeterminism, resulting in differences in solutions and performance between the BI sequential and parallel versions, as well as between parallel executions involving different numbers of processors. Recall that, since there is no unique correspondence between the feasible permutation and the resulting schedule [11], several feasible permutations can yield the same schedule (or at least schedule with the same length) of tasks among processors . In an sequential execution, the neighborhood is searched always in the same direction (FORWARD or BACKWARD, Figure 3) until the first occurrence of a solution leading to the desired improvement, which is then used to direct the next search iteration. When the neighborhood is explored in parallel, the definition of "the first occurrence" is no longer the same. The starting point for further exploration is the first reported solution, which may occur anywhere in the permutation, by any processor. We hope these variations may induce the exploration of different regions of the search space and attain the objective of improving the quality of the final solution without increasing the parallel execution time. According to the described characteristics, the PNEF strategy can be classified as 1C/KS/SPSS.

4 Implementation details

In this section, we describe the implementation environment and the ideas we applied in order to ensure an efficient exploitation of the available parallel architecture for both variants of parallel LS.

4.1 Experimentation environment

The multiprocessor architecture is assumed to contain q identical processors organized in a master-slave architecture (Figure 8). We define Processor 0 as the supervising processor (master) with the main role of performing I/O, communication, and coordination tasks. In the case of PNEC, the master performs its part of calculations too. The other $q-1$ are working processors (slaves), which perform the computations needed for the execution of the LS procedure. Our multiprocessor system is based on SUN computers and Message Passing Interface (MPI) communication protocol (library for C programming language) [14, 15].

The master-slave architecture has been selected for several reasons. First, it ensures the minimization of communications. Having a small diameter (the maximum distance between any two processors is 2), it allows us to minimize the number of messages required to distribute all necessary data among the processors. Second, its simplicity allows us to perform the detailed analysis of the performance of the proposed parallelization strategies given multiprocessor characteristics (defined by *ipc*).

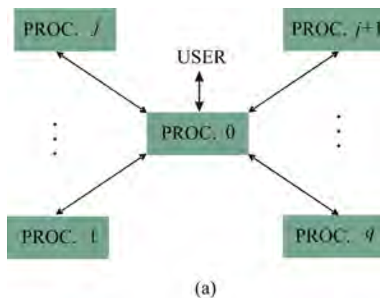


Figure 8: Multiprocessor architecture for parallel LS experimentation

It is very important to distinguish between the multiprocessor architecture for the parallel execution of LS (the machine on which experiments are run) and the target multiprocessor architecture of the scheduling problem. For the execution of parallel LS, we always use a master-slave architecture with q processors as explained above. The target multiprocessor architecture for scheduling task graphs consists of p processors connected in an arbitrary way. For our experiments we used hypercubes of various sizes, as suggested in [11, 9].

We performed several optimization steps in our code to minimize the amount of data to be exchanged between processors and to balance the computational load among them. The data types were appropriately chosen to reduce the quantity of required data transfers. Moreover, we exchanged only the necessary information (the solution quality, i.e., schedule length) not the entire feasible permutation, unless improvement was observe.

To determine the performance of a parallelization strategy, it is important to measure the communication and computation times properly. Each worker measures the time between receiving two important messages from the supervisor, the START and STOP commands. This is its total execution time (t_{tot}). Each processor is also calculating the computation time (t_{CPU}) as a sum of all time intervals that do not involve communications. This means that the difference $t_{tot} - t_{CPU}$ includes the communication time and the processor idling time. Based on the obtained time measuring results, we were able to find the best balance between computation and communication amounts, described in Section 5.

Finally, we use both the speedup factor S [2], calculated as the ratio between execution time of the best performing sequential algorithm and the the greatest t_{tot} among all the processors, and the solution quality to evaluate the efficiency of the proposed parallel LS procedures.

4.2 Implementation of the PNE strategies

The initial solution x_0 is created by each processor applying the CPES heuristic [11]. This ensures savings in the communications and reduces the idle time intervals of the processors. At the end of the parallel LS execution, be it either the PNEC or PNEF variant, the workers send to the supervisor the time measurement data (t_{tot} and t_{CPU}) to be reported.

4.2.1 Implementation of the PNEC strategy

The PNEC parallelization strategy followed the load balanced partition described in Section 3.2. Starting and ending task indexes defining each region are determined by the corresponding processor at the beginning of the program execution. Once the partition is fixed, each of the q processors starts exploring its part of the neighborhood by moving tasks with indexes in the associated range. Upon completion of the parallel neighborhood exploration, all processors are synchronized to communicate the obtained results. Working processors send their minima to the supervisor and wait for the next instruction. The supervisor collects all schedule lengths and decides on the next step. If the current best solution is improved, the new one (obtained upon request from the corresponding processor) is sent to all workers as the initial solution for the next iteration. In the case when the current incumbent is not improved, the STOP message is sent to all workers and the best found solution is reported.

4.2.2 Implementation of the PNEF strategy

PNEF was implemented with $q-1$ processors performing LS in parallel, while the supervisor performs the dynamic distribution of tasks and verifying the quality of each obtained improved solution (potentially n of them). It also determined the initial solution for the next iteration by selecting one of these solutions according to the predefined criterion.

At the beginning of each iteration the first $q-1$ tasks are taken by each worker in a deterministic manner: the first working processor performs movements of the first task, and so on. Once a worker completes the search connected to the currently associated task, it sends the results (new feasible permutation and the corresponding makespan) to the supervisor. The worker then receives from the supervisor either the index of the next task to be processed or an indication that the new iteration is about to start and a new solution to arrive. When the iteration is completed, and the current best solution is improved, the supervisor broadcasts the new solution among the workers. When the current best solution is not improved, the STOP message is sent to all of the workers.

5 Experimentations with parallel LS

Our programs are developed in the C programming language, and the MPI communication library is used for the data exchange between processors. The experimental evaluation is performed on a SUN Enterprise 10000 multiprocessor system containing 64 processors, each with 400MHz clock and 64Gb of RAM. The parallel LS procedures were tested on up to 30 processors.

5.1 Parameter calibration

The first phase of any experimental evaluation is certainly the decision on parameter values. In our case, we have to reason about both the parameters of the LS procedure and the parameters for parallelization. The parameters for LS are (see Section 2) the type if improvement rate (FI or BI) and the search direction (FORWARD or BACKWARD). Regarding the parallelization, we have to select the type of neighborhood partition and to determine the most suitable value for the number of processors q.

For parameter selection and the quality evaluation of our procedures, we experimented with "hard" test instances with known optimal solutions [9]. This set contains 10 instances, which are sparse task graphs similar to the one given in Figure 1 with optimal solutions of the type presented in Figure 9. The number of tasks in these graphs ranges from 50 to 500, with the increment of 50, while the edge density is around 30% of the maximum allowed density (calculated from the corresponding optimum solution). In this phase, we assumed that the target multiprocessor architecture for scheduling is the 2-dimensional hypercube given in Figure 2.

P_4	t_4	t_7	t_{11}	
P_3	t_3	t_6	t_{10}	
P_2	t_2	t_5	t_9	t_{12}
P_1	t_1	t_8		

Figure 9: Optimal schedule for task graph with 12 nodes

The results of the experiments with the PNEC and PNEF variants are are summarized in Table 2 for all LS parameter combinations. Detailed results are available from the authors.

The third and the sixteenth rows of Table 2 contain the sequential execution information, the average schedule length and execution time. The data we present for the

Table 2: Comparison of PNEC and PNEF variants, instances with known optimal solutions

FORWARD search									
	BI search					FI search			
1	2781.4	131.79	2781.4.	131.79		2772.5	59.38	2772.5	59.38
	PNEC		PNEF			PNEC		PNEF	
q	% imp.	S	% imp.	S		% imp.	S	% imp.	S
2	0.00	1.83	0.00	0.97		-0.003	1.56	0.00	0.96
3	0.00	2.46	-0.001	2.50		-0.005	1.69	0.001	1.72
4	0.00	3.04	-0.001	3.74		-0.005	1.65	0.00	3.43
5	0.00	3.56	-0.001	4.98		-0.004	2.01	0.001	4.51
6	0.00	3.58	-0.001	6.20		-0.003	2.37	0.00	4.98
7	0.00	5.61	-0.001	7.41		-0.004	1.97	-0.003	5.75
8	0.00	5.33	-0.001	8.66		-0.002	1.62	-0.003	6.58
9	0.00	6.52	-0.001	9.91		-0.004	2.35	-0.003	7.46
10	0.00	6.92	-0.0007	11.44		-0.004	2.57	-0.004	11.20

BACKWARD search									
1	2737.9	104.64	2737.9	104.64		2771.6	65.14	2771.6	65.14
	PNEC		PNEF			PNEC		PNEF	
q	% imp.	S	% imp.	S		% imp.	S	% imp.	S
2	0.00	1.82	0.00	0.96		-0.003	1.06	0.00	0.95
3	0.00	2.15	0.00	1.90		-0.001	1.26	0.019	1.52
4	0.00	3.51	0.00	2.82		-0.001	1.56	0.019	2.25
5	0.00	3.40	0.00	3.78		0.002	2.25	0.016	3.30
6	0.00	3.63	0.00	4.68		0.013	2.27	0.018	4.35
7	0.00	4.86	0.00	5.51		0.001	2.04	0.019	4.85
8	0.00	5.21	0.00	6.68		-0.001	2.38	0.017	5.39
9	0.00	6.26	0.00	7.65		-0.002	2.48	0.017	6.12
10	0.00	6.50	0.00	8.89		-0.002	2.38	0.016	6.99

parallel executions (rows 6 to 14 and 18 to 26) are average improvements with respect to the sequentially obtained schedule length and the speedup of the corresponding parallel PNE. A negative figure means that the parallel solution is worse then the sequential one. It is obvious that the BI parallel execution of PNEC is the same as the sequential one, while the execution of PNEF is completely nondeterministic for all the combinations of parameters.

The main conclusion regarding both variants of PNE is that the proposed decomposition provides significant speedups, although sometimes yielding a small degradation in solution quality. Neither degradation nor improvement is significant, however. Hence, we do not consider this factor relevant. Moreover, one notices the systematic increase in speedup with the number of processors growing.

Examining the results in Table 2, we can conclude that PNEF performs better then PNEC for the given multiprocessor architecture. This is a consequence of the ideal load

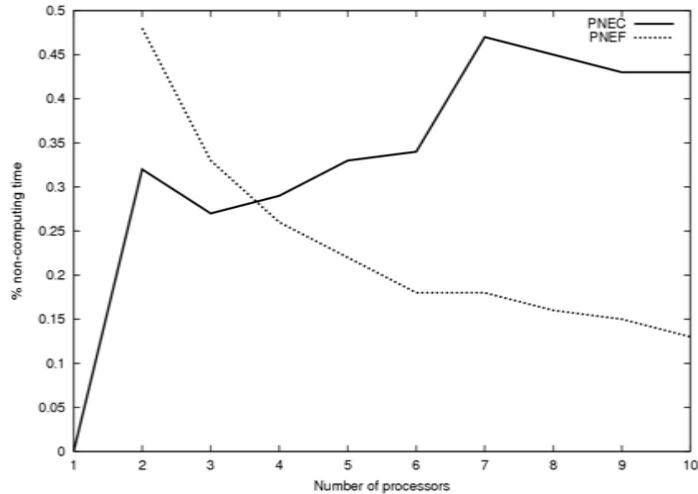


Figure 10: Communication and idling time of processors in PNEC and PNEF

balancing achieved by the fine grained decomposition, which is powerful enough to cancel the increase in the amount of communications. To support this conclusion, we present in Figure 10 the normalized value of the percentage of non-computing time (the time spent for communication and idling of processors) t_{NC} as a function of q for the two PNE strategies. For PNEC, t_{NC} is calculated as

$$t_{NC}^{PNEC} = \frac{\sum_{i=1}^{q}(t_{tot}(i) - t_{CPU}(i))}{\sum_{i=1}^{q} t_{tot}(i)}.$$

The formula in the PNEF case is slightly different since one processor (the supervisor) is not performing computations. We considered its total execution time as the non-computing part, and therefore,

$$t_{NC}^{PNEF} = \frac{t_{tot}(1) + \sum_{i=2}^{q}(t_{tot}(i) - t_{CPU}(i))}{\sum_{i=1}^{q} t_{tot}(i)}.$$

We used the results from the FI-FORWARD search for completely random instances to generate Figure 10. The conclusion also holds for the other instances, the non-computing time for PNEF decreases with an increase in q.

Since we obtained a constant increase in speedup by adding new processors (up to 10), we could conclude that scalability is achieved at least for $q \leq 10$. In the subsection to follow, we experiment with architectures containing more processors (10, 15, ...). We use only PNEF for further evaluations since, in our case, it obviously outperforms PNEC.

5.2 Scheduling results for PNEF

In this section, we describe the results obtained by the proposed PNEF variant of the parallel LS to the extended set of instances. We use representative subsets of the two benchmark sets proposed in [9]. The first consists of 36 sparse task graphs ($\rho = 20\%$), for which $n = 50, 100, 200, 300, 400, 500$ and 6 graphs for each n. Graphs were generated randomly as described in [9]. We varied the target multiprocessor architecture (by changing $D_{p\times p}$) and compare the local minima obtained by the sequential and parallel variants. For the target multiprocessor systems we selected 1D-, 2D-, and 3D-hypercubes. We varied the target multiprocessors architecture in order to examine how the structure of the problem influences the performance of the parallel LS. The second subset consists of 100 task graphs with known optimal solutions for 2D-hypercube multiprocessor networks (with the number of task varying from 50 to 500 with the increment 50 and ten different densities for the each number of tasks).

Table 3: Scheduling results for random task graphs on $p = 2$ processors

$n\backslash q$	1	2	4	6	8	10	15	20	25	30
	SL	% improvement								
50	405.83	0.00	0.00	-0.0004	0.00	0.00	0.00	-0.0004	-0.0004	0.00
100	894.17	0.00	0.00	0.00	-0.0002	-0.0002	-0.0009	-0.0033	-0.0009	-0.0002
200	1876.50	0.00	-0.0003	0.0003	0.0003	-0.0004	-0.0005	0.0001	-0.0004	-0.0004
300	2812.17	0.00	0.0002	-0.0004	-0.0001	-0.0005	-0.0004	-0.0010	-0.0012	-0.0015
400	3686.83	0.00	-0.0001	-0.0003	-0.0006	-0.0007	0.00	0.0001	-0.0005	-0.0013
500	4609.00	0.00	-0.0005	0.0003	0.0004	-0.0002	-0.0008	-0.0008	-0.0001	-0.0003
av.	2380.75	0.00	-0.0001	-0.0001	-0.0004	-0.0003	-0.0004	-0.0009	-0.0006	-0.0006
	CPU time	Speedup								
50	0.14	0.87	2.33	4.67	4.67	7.00	7.00	14.00	7.00	4.67
100	2.07	0.98	2.69	4.31	5.59	7.14	10.35	14.79	8.28	10.35
200	36.02	0.99	3.07	5.67	6.91	10.12	16.15	20.35	12.29	16.15
300	189.27	1.00	3.37	6.21	9.22	12.27	17.94	26.92	18.78	22.35
400	763.42	1.00	3.25	5.48	8.37	11.56	17.62	24.67	18.56	21.18
500	2137.03	0.63	3.84	6.40	8.86	13.35	20.76	31.81	20.34	25.79
av.	521.32	0.91	3.09	5.46	7.27	10.24	14.97	22.09	14.21	16.75

The scheduling results for random task graphs are presented in Tables 3, 4, and 5. We present each row of the tables the average results for the six instances of the same size when q was varied up to 30 processors. The search parameter combination is FI-BACKWARDS.

The tables are divided into two parts, the upper part containing solution quality measures, and the lower one presenting speedup factors. The number of tasks is given in the first column, the sequential execution characteristics (average schedule length for the upper part and wall-clock execution time for the lower one) are in the second column.

Table 4: Scheduling results for random task graphs on $p = 4$ processors

$n \backslash q$	1	2	4	6	8	10	15	20	25	30
	SL	% improvement								
50	292.33	0.00	0.0011	0.0011	0.0040	0.0040	0.0040	0.0040	0.0040	0.040
100	617.33	0.00	0.0016	0.0011	0.0011	0.0016	0.0041	0.0019	0.0019	0.0013
200	1273.33	0.00	-0.0015	0.0006	0.0012	0.0016	0.0006	-0.0001	-0.0008	-0.0007
300	1913.00	0.00	-0.0031	-0.0027	-0.0031	-0.0031	-0.0029	-0.0011	0.0011	-0.0022
400	2555.17	0.00	-0.0001	0.0013	0.0011	-0.0006	-0.0008	-0.0014	-0.0012	-0.0010
500	3225.17	0.00	0.0001	-0.0033	-0.0032	0.0008	0.0005	-0.0033	-0.0034	-0.0030
av.	1646.06	0.00	-0.0003	-0.0003	0.0002	0.0007	0.0009	0.0000	0.0003	0.0057
	CPU time	Speedup								
50	0.08	0.89	2.00	2.67	4.00	4.00	8.00	8.00	4.00	4.00
100	1.18	0.72	2.11	3.19	4.37	5.62	7.87	9.08	6.55	7.87
200	16.69	0.99	2.89	4.64	6.54	8.65	12.45	14.02	10.30	12.09
300	109.93	1.01	3.43	5.57	7.44	9.23	15.42	19.18	11.94	15.27
400	383.43	0.99	3.09	4.79	7.18	9.97	16.91	21.38	14.14	15.50
500	1004.65	0.99	3.21	5.70	7.87	10.45	15.56	23.92	15.09	17.72
av.	252.66	0.93	2.79	4.43	6.23	7.99	12.70	15.93	10.34	12.06

The remaining 9 columns of the upper part contain percentages of the improvement with respect to the sequential solution using different numbers of processors for PNEF.

Note that PNEF on two processors ($q = 2$) performs the same as the sequential LS, only the communication between supervisor and worker is added. Therefore, the percentage of improvement in this case is always zero, while the speedup factor is a little bit less than 1, since some time is spent on communications. The real gains with parallel LS are obtained by engaging more processors. On the other hand, as can be seen from Tables 3–5, too many processors do not provide an increase in solution quality. Moreover, by adding new processors, we increase communications and this results in the decrease of the speedup factor when using more than 20 processors. It should be noted that the dependence between the amount of communications and the speedup is not quite straightforward, as speedup is also influenced by the non-determinism of the search process.

A very interesting observation is that the sequential LS for $p = 4$ requires only half the time needed for the sequential scheduling for $p = 2$. This may look strange since the problem size is increased. However, in our case, it just reflects the problem characteristic that it is easier to improve the current solution when there is more space to search (more processors to transfer tasks to). Moreover, the FI strategy makes the increase in sequential search speed quite natural. On the other hand, scheduling onto 3D-hypercube ($p = 8$ processors) is not so fast, as no significant improvement could be obtained while we still have twice more processors to check (let us remind here that the complexity of

Table 5: Scheduling results for random task graphs on $p = 8$ processors

$n \backslash q$	1	2	4	6	8	10	15	20	25	30
	SL	% improvement								
50	291.17	0.00	0.00	0.00	0.00	-0.0023	-0.0023	-0.0023	-0.0011	-0.0011
100	609.50	0.00	0.0003	0.0003	-0.0019	-0.0019	-0.0049	-0.0081	-0.0073	-0.0024
200	1260.67	0.00	0.00	-0.0005	-0.0007	-0.0010	-0.0032	-0.0029	-0.0029	-0.0030
300	1900.83	0.00	0.00	0.0005	0.0002	0.0002	-0.0028	-0.0028	-0.0026	-0.0042
400	2551.67	0.00	0.0001	0.0012	0.0012	0.0012	0.0017	0.0019	0.0033	0.0020
500	3228.83	0.00	0.0007	0.0009	0.0008	0.0019	0.0020	0.0014	0.0009	0.0004
av.	1640.44	0.00	0.0002	0.0004	-0.0001	-0.0003	-0.0016	-0.0021	-0.0016	-0.0014
	CPU time	Speedup								
50	0.12	0.92	3.00	4.00	6.00	6.00	12.00	12.00	6.00	6.00
100	2.14	0.97	2.97	4.55	6.29	7.64	14.26	15.28	10.19	10.70
200	22.98	0.83	2.73	4.48	6.02	8.24	12.84	15.53	10.54	11.55
300	99.23	0.99	3.11	4.89	6.60	8.27	13.02	16.76	10.93	12.56
400	438.47	0.99	3.09	5.15	7.10	9.40	14.14	18.77	11.69	14.02
500	1253.67	1.00	2.83	4.75	6.80	9.01	13.78	19.46	12.56	14.65
av.	302.77	0.95	2.95	4.64	6.47	8.09	13.34	16.30	10.32	11.58

the ES scheduling algorithm is $O(n^2p)$).

Another interesting fact is that scheduling on $p = 4$ processors seems to be easier for the parallel execution. We obtain an improvement in solution quality in most cases. On the other hand, the experiments show that this improvement does not increase with the number of engaged processors q (sometimes one even observes a degradation in the solution quality). We therefore conclude that for the problem and instances addressed, PNEF performs best on a modest number of processors (around 10). Regarding the scheduling for $p = 8$, it is not possible to increase the solution quality, as the strong data dependency characteristic of the problem studied reduces the inherent parallelism. Moreover, adding new processors for parallel execution yields a degradation in solution quality. The main reason seems to be that, increasing the number of processors, increases the number of permutation partitions while decreasing their size. This results in a rather large, potentially overlapping number of solutions that may be discovered more or less simultaneously, the first improving one not being necessarily the one that would give the best trajectory. A small improvement in solution quality is however observed for large instances ($n > 300$), as the size of each region is also relatively larger mitigating the issue.

The results for the second part of our experimental evaluation are presented in Table 6. We present the average (over 10 values) percentage of deviation from the known optimal solution (including for the sequential execution) in the upper part of the table. The lower part, illustrating the speedup of PNEF, contains the same type of data as the previous

tables.

Table 6: Results for random task graphs with known optimal solution on $p = 4$ processors

$n \backslash q$	1	2	4	6	8	10	15	20	25	30
	% deviation									
50	34.50	34.50	33.91	28.39	28.39	27.24	28.72	28.72	28.72	28.72
100	49.21	49.21	45.60	52.00	52.15	52.15	48.17	45.79	52.69	48.49
150	59.10	59.10	59.61	59.54	59.59	60.01	59.17	59.33	58.93	59.01
200	59.78	59.78	65.05	65.73	68.67	71.46	69.94	72.71	71.48	72.10
250	68.79	68.79	68.35	74.93	70.16	70.28	70.55	70.69	70.32	73.80
300	73.51	73.51	83.64	79.75	82.32	71.14	71.47	75.69	75.85	75.85
350	67.93	67.93	74.41	74.50	74.56	73.98	70.77	70.47	74.54	74.53
400	89.94	89.94	90.00	90.05	80.10	79.34	89.89	86.19	75.40	75.85
450	65.39	65.39	51.66	51.35	51.21	51.45	51.10	50.88	51.10	51.10
500	42.88	42.88	45.39	53.57	50.04	50.07	49.68	48.93	48.69	48.47
av.	61.10	61.10	61.76	62.98	61.72	60.07	60.95	60.94	60.77	60.79
	CPU time	Speedup								
50	0.22	0.92	2.75	5.50	7.33	7.33	11.00	11.00	11.00	11.00
100	1.93	0.97	2.97	5.68	7.15	9.19	10.72	13.79	9.19	9.19
150	4.02	0.97	2.26	4.02	4.79	6.38	8.55	9.57	5.58	6.00
200	15.47	0.98	2.72	3.91	4.96	6.31	8.41	11.54	21.49	8.19
250	23.78	0.99	2.96	3.90	5.75	7.79	11.98	15.26	9.65	13.93
300	67.22	0.99	3.07	3.35	6.61	4.54	6.64	11.22	7.34	8.74
350	161.99	1.00	4.48	7.92	9.87	12.49	11.77	15.40	15.43	19.24
400	80.43	1.00	2.46	4.13	5.30	5.86	9.27	6.90	3.34	5.20
450	369.57	1.00	3.58	6.56	9.37	12.45	15.63	20.44	13.70	15.65
500	840.61	1.00	2.32	4.70	8.36	11.19	17.49	25.05	16.51	17.79
av.	156.52	0.98	2.96	4.97	6.95	8.35	11.15	14.02	11.32	11.49

As can be seen from Table 6, the deviations of the local minima from the known optimal solutions are quite large. This is consistent with the results presented in [11] for the sequential case. We can also note that nondeterminism of PNEF sometimes improves and sometimes degrades the quality of the obtained local minimum. Regarding the speedup of the parallel search, conclusion is almost the same as in the previous case; parallelization is always good, and it appears most efficient for $q \leq 10$. The feneral conclusion about the proposed LS parallelization strategies is that the gain is mostly in significantly speeding up the search process, while the solution quality cannot be improved much.

The question then is, what is the impact of parallel LS on the performance of meta-heuristics that call upon it, always with respect to solution quality and computational efficiency. We address this question in the next subsection.

6 Meta-heuristics with Parallel LS

We analyze the impact of parallel LS on the performance of meta-heuristics through two solution approaches, Multistart Local Search and Variable Neighborhood Search. Both methods rely on the efficiency of LS but are quite different in the sophistication of their search mechanisms.

Multistart Local Search (MLS) is the simplest meta-heuristic method. It consists in restarting the LS procedure from different (usually randomly selected) initial solutions until some predefined stopping criterion is satisfied. The usual stopping criteria are maximum number of iterations, maximum number of iterations without improvement of the current best solution, maximum allowed CPU time, etc.

Variable Neighborhood Search (VNS) meta-heuristic was proposed by Mladenović and Hansen [25]. It uses multiple neighborhoods $\mathcal{N}_k$, $(k=1,\ldots,k_{max})$ to avoid being trapped in a local optimum. Usually, the initial solution is determined by some constructive scheduling heuristic and then improved by LS before the beginning of the actual VNS procedure. The main loop of VNS consists of four steps: shaking, improving, moving and stopping-criterion checking. In *shaking*, the diversification step, a random point x' in the kth neighborhood of the current best solution x is generated. Then, some LS procedure is performed starting from x' within the *improving* step. The improved solution, that is, the local optimum, x'' is used in the *moving* step to guide further the search: when it is the new incumbent, the search is concentrated around this solution, otherwise the next neighborhood for shaking is selected, i.e., the value for k is incremented. The final step is used to verify if the stopping criterion is met. Recent developments and applications of VNS may be found in [26, 16].

To implement MLS or VNS with parallel LS, we need only to incorporate the selected parallel LS procedure (PNEF) into the respective meta-heuristic. We run the resulting parallelized version of the two meta-heuristics on different number of processors with the maximum allowed execution time as the stopping criterion. We put the same time limit (regardless the number of processors used for the parallel execution) for all task graphs of the same size in order to check for the improvement of the final solution. We thus analyzed the number of iterations performed by each meta-heuristic method as the speedup indicator. The scheduling results for the sparse task graphs used in Section 5.1 are presented in Table 7.

The results displayed in Table 7 show that, except in a few isolated cases, the final solution quality improves when the number of processors increases, and the improvement is significant. The corresponding increase in the number of performed iterations is also evident, showing an almost linear growth (speedup) for both methods. The difference between the number of iterations observed for the two tested methods is due to the differences in the definitions of the term iteration: for MLS, a single iteration involves

Table 7: MLS and VNS results on instances with known optimal solution for $p = 4$

$q =$	1	2	3	4	5	6	7	8	9
	Average schedule length								
MLS	2725.3	2714.7	2714.6	2563.9	2561.0	2541.5	2542.1	2532.1	2539.4
VNS	2255.8	2153.8	2212.6	2179.2	2020.4	1987.7	1981.3	1800.7	1786.6
	Average number of iterations								
MLS	154.8	303.3	452.6	593.9	692.2	870.5	1015.0	1107.9	1286.6
VNS	10.9	18.2	32.1	39.7	49.0	55.2	69.9	75.8	78.8

the execution of LS from a random initial solution, while in the case of VNS, the number of iterations represents how many times neighborhood k_{max} was visited.

Figures 11 to 18 illustrate the behavior of the two meta-heuristics with PNEF parallel LS on one sparse and one dense graph from each set of task graphs of size $n = 300$, scheduled onto 2D-hypercubes (with time limit set to $t_{max} = 1200s$). The figures represent improvement in time of the schedule length for different q. Vertical lines correspond to the proportionally shorter execution time, i.e., when we use q processors for the parallel execution, we are interested in the solution quality starting from point $t = t_{max}/q$. The illustrations emphasize the quality of the solutions obtained and the speedup estimations. In most of the cases, the sequential solution is significantly improved by parallelization and this improvement is obtained in proportionally shorter time. The gain due to parallel LS is evident for both meta-heuristics, especially for the task graphs with known optimal solution, which were proven to be very hard.

These results are impressive. They clearly show the importance of parallel LS for the performance of the meta-heuristics for hard combinatorial optimization problems.

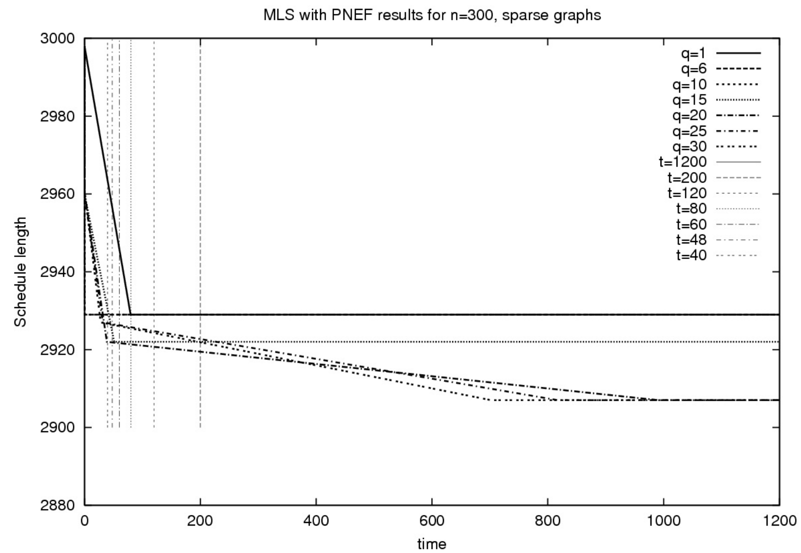


Figure 11: Scheduling results for MLS with PNEF for sparse random task graph ($\rho = 20$) with known optimal solution $SL_{opt} = 1600$

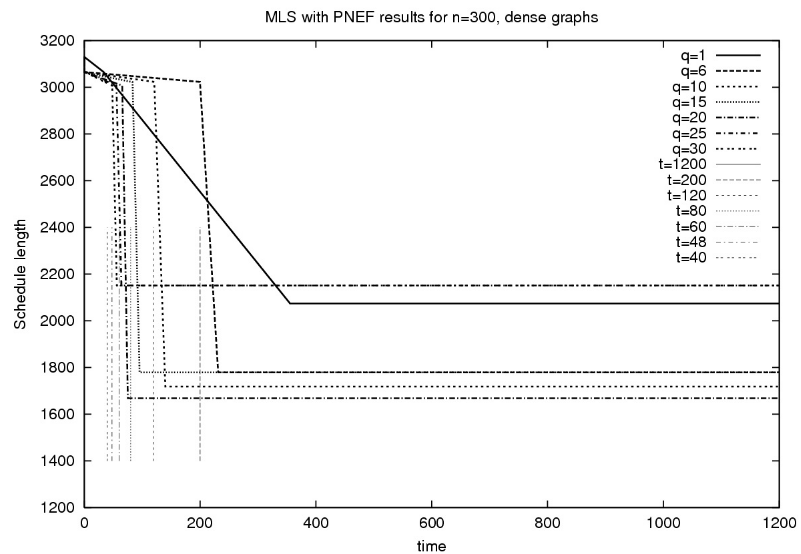


Figure 12: Scheduling results for MLS with PNEF for dense random task graph ($\rho = 60$) with known optimal solution $SL_{opt} = 1600$

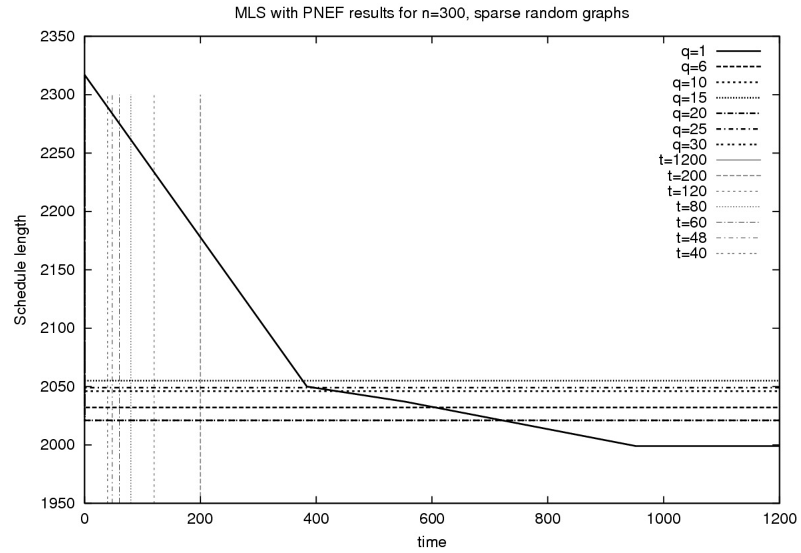


Figure 13: Scheduling results for MLS with PNEF for sparse random task graph ($\rho = 20$)

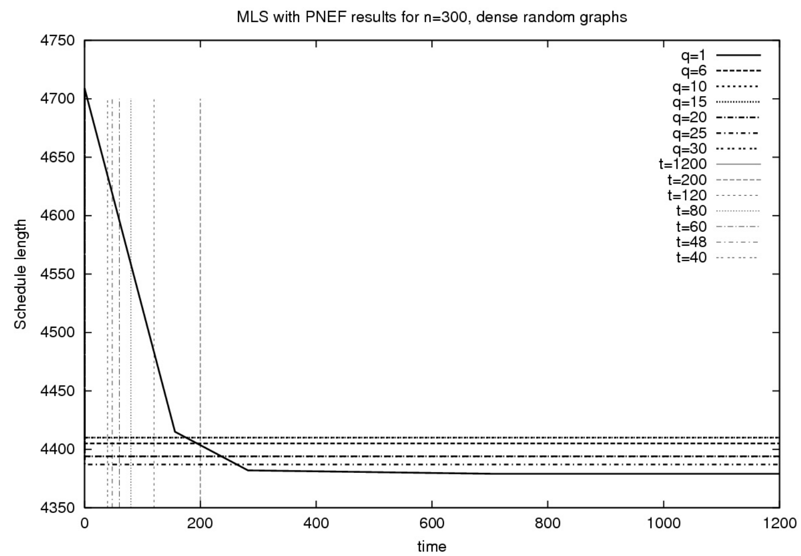


Figure 14: Scheduling results for MLS with PNEF for dense random task graph ($\rho = 60$)

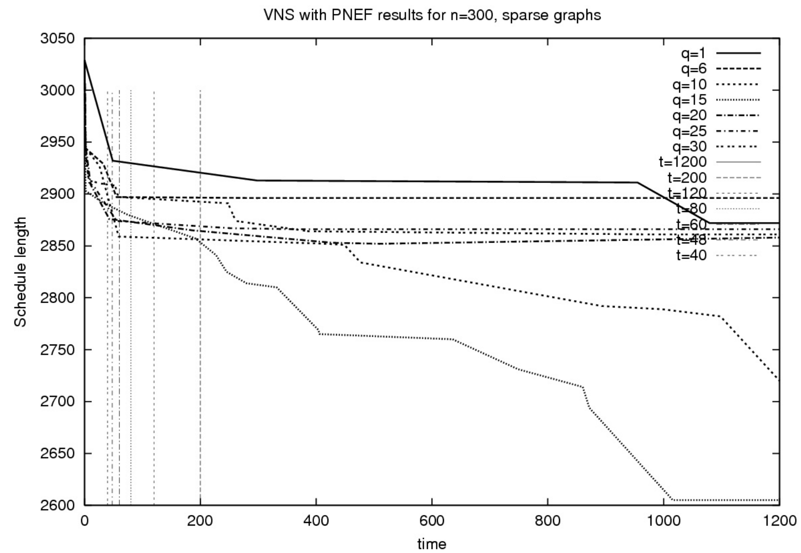


Figure 15: Scheduling results for VNS with PNEF for sparse random task graph ($\rho = 20$) with known optimal solution $SL_{opt} = 1600$

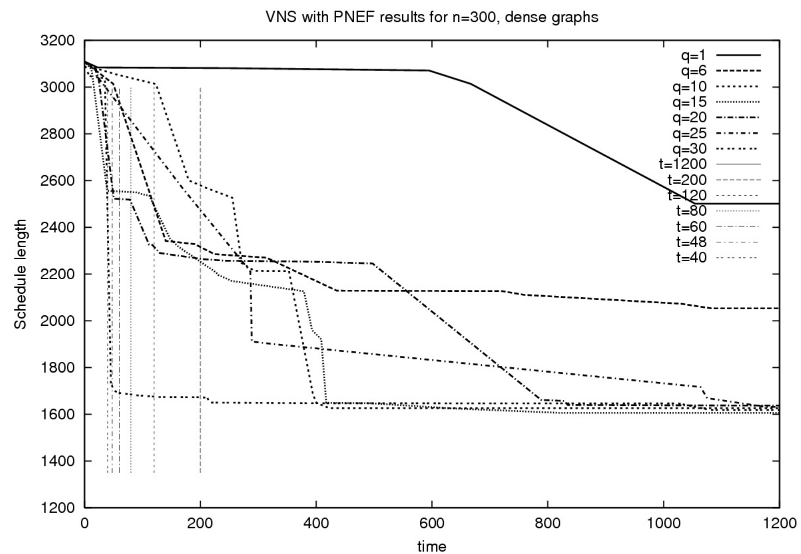


Figure 16: Scheduling results for VNS with PNEF for dense random task graph ($\rho = 60$) with known optimal solution $SL_{opt} = 1600$

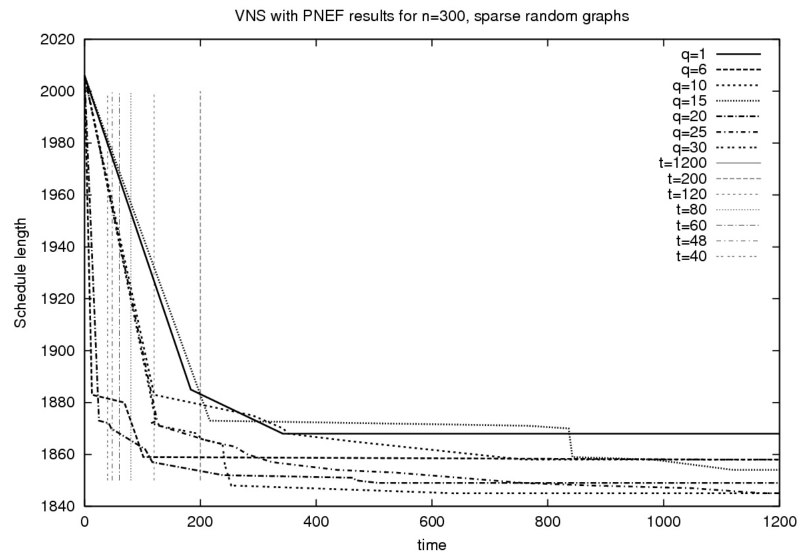


Figure 17: Scheduling results for VNS with PNEF for sparse random task graph ($\rho = 20$)

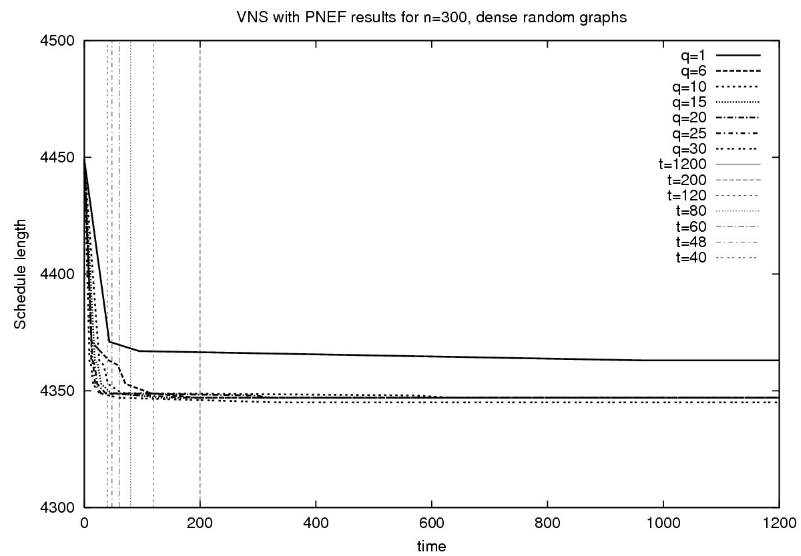


Figure 18: Scheduling results for VNS with PNEF for dense random task graph ($\rho = 60$)

7 Conclusions

We addressed the parallelization of Local Search operating on neighborhoods of variable sizes. LS is a basic building block of many meta-heuristic methods. At the same time, it is usually computationally very intensive and it is called upon intensively during a meta-heuristic search. It is thus important to enhance its efficiency and parallel algorithmic design offers a very promising avenue in this respect.

We therefore proposed several parallel LS strategies, using the successful sequential LS developed for the MSPCD as our very challenging illustrative environment. We analyzed fixed and dynamic neighborhood-partition strategies and identified the fine-grained dynamic partition approach as the most appropriate, as it provides an ideal load-balancing of the processors involved in the parallel LS. Extensive numerical experiments have shown that this parallel LS strategy yields solutions of better quality significantly faster than the sequential procedure.

We also analyzed the performance of the proposed parallel LS through its impact on the behavior and performance of meta-heuristics, which is, after all, the ultimate goal of developing parallel LS. We used two very different meta-heuristics, the Multi-start Search and the Variable Neighborhood Search. Experimental results show remarkable improvements in solution quality and significant speedups, emphasizing the value of the proposed parallel LS strategy.

In all experiments, the results showed that best results were obtained when a modest number of processors was used. For the instances tested, both solution quality and speedup were the best for $q = 10$. Yet, the experiments also hinted that a larger number of processors may be beneficial when instance dimensions grow, pointing to the scalability of the method.

An interesting and challenging research avenue is now to combine the fine-grained parallel LS method and coarse-grained cooperative search. We hope to report results on this topics in the near future.

Acknowledgments

This work has been partially supported by the Serbian Ministry of Science, Grant nos. 174010 and 174033. Funding was also provided by the Natural Sciences and Engineering Council of Canada, through its Discovery Grant program. The authors would like to thank Mr. François Guertin for his unlimited help with parallel programming and executions and Calcul Québec and Compute Canada for the computational resources.

References

[1] E. Alba, editor. *Parallel metaheuristics: a new class of algorithms.* Wiley-Interscience, 2005.

[2] J. Blazewicz, M. Drozdowski, and K. Ecker. Management of resources in parallel systems. In J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, editors, *Handbook on Parallel and Distributed Processing*, pages 263–341. Springer, 2000.

[3] W. Bozejko and M. Wodecki. Parallel tabu search method approach for very difficult permutation scheduling problems. In *Proc. Int. Conf. Parallel Computing in Electrical Engineering (PARELEC'04)*, pages 156–161, Dresden, Germany, Sept. 07–10 2004.

[4] P. Brucker. *Scheduling Algorithms.* (fifth eidtion) Springer Verlag, 2007.

[5] T. G. Crainic and N. Hail. Parallel meta-heuristics applications. In E. Alba, editor, *Parallel Metaheuristics*, pages 447–494. John Wiley & Sons, Hoboken, NJ., 2005.

[6] T.G. Crainic and M. Toulouse. Parallel Meta-heuristics. In M. Gendreau and J.Y. Potvin, editors, *Handbook of metaheuristics*, pages 497–541. Springer, 2010.

[7] V.-D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol. Strategies for the parallel implementations of metaheuristics. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer Academic Publishers, Norwell, MA, 2002.

[8] T. Davidović. Exhaustive list–scheduling heuristic for dense task graphs. *YUJOR*, 10(1):123–136, 2000.

[9] T. Davidović and T. G. Crainic. Benchmark problem instances for static task scheduling of task graphs with communication delays on homogeneous multiprocessor systems. *Comput. Oper. Res.*, 33(8):2155–2177, Aug. 2006.

[10] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović. Towards the optimal solution of the multiprocessor scheduling problem with communication delays. In *Proc. 3rd Multidisciplinary Int. Conf. on Scheduling: Theory and Application*, Paris, France.

[11] T. Davidović, P. Hansen, and N. Mladenović. Permutation based genetic, tabu and variable neighborhood search heuristics for multiprocessor scheduling with communication delays. *Asia Pac. J. Oper. Res.*, 22(3):297–326, Sept. 2005.

[12] G. Djordjević and M. Tošić. A compile-time scheduling heuristic for multiprocessor architectures. *The Computer Journal*, 39(8):663–674, 1996.

[13] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez, and J. M. Moreno-Vega. The parallel variable neighborhood search for the p-median problem. *J. Heur.*, 8(3): 375–388, May 2002.

[14] W. Gropp and E. Lusk. *Users Guide for* `mpich` *a Portable Implementation of MPI.* University of Chicago, Argonne National Laboratory, 1996.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* The MIT Press, 1994.

[16] P. Hansen, N. Mladenović, J. Brimberg, and J. A. Moreno Pérez. Variable neighbourhood search. In M. Gendreau and J-Y. Potvin, editors, *Handbook of Metaheuristics*, pages 61–86. (second edition) Springer, New York Dordrecht Heidelberg London, 2010.

[17] H. Jin, D. Jespersen, P. Mehrotra, and R. Biswas. High performance computing using mpi and openmp on multi-core parallel systems. *Parallel Computing*, 37(9): 562–575, 2011.

[18] W. H. Kohler. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. Computers*, pages 1235–1238, Dec. 1975.

[19] X. Kong, W. Xu, and J. Liu. A permutation-based differential evolution algorithm incorporating simulated annealing for multiprocessor scheduling with communication delays. *LNCS: Computational Science–ICCS 2006*, 3991:514–521, 2006.

[20] Y.-K. Kwok and I. Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proc. 7th IEEE Symposium of Parallel and Distributed Processing (SPDP'95)*, pages 36–43, Dallas, Texas, USA, Oct. 1995.

[21] Y.-K. Kwok and I. Ahmad. Dynamic critical path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[22] Y.-K. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *J. Parallel and Distributed Computing*, 47:58–77, 1997.

[23] Y-K. Kwok and I. Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65(12): 1515–1532, 2005.

[24] B. A. Malloy, E. L. Lloyd, and M. L. Soffa. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Trans. Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[25] N. Mladenović and P. Hansen. Variable neighborhood search. *Comput. & OR*, 24 (11):1097–1100, 1997.

[26] J. A. Moreno Pérez, P. Hansen, and N. Mladenović. Parallel variable neighborhood search. In E. Alba, editor, *Parallel Metaheuristics*, pages 247–266. John Wiley & Sons, Hoboken, NJ., 2005.

[27] M. Pinedo. *Scheduling Theory, Algorithms and Systems.* 2dn edition, Prentice Hall, 2008.

[28] S.C. Porto and C.C. Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *Int. J. High-Speed Computing*, 7: 45–71, 1995.

[29] S.C. Porto and C.C. Ribeiro. Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. *J. Heur.*, 1(2):207–225, 1995.

[30] A. K. Sarje and G. Sagar. Heuristic model for task allocation in distributed computer systems. *IEE Proceedings-E*, 138(5):313–318, Sept. 1991.

[31] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel and Distributed Systems*, 4(2):175–187, February 1993.

[32] S. Venugopalan and O. Sinnen. Optimal linear programming solutions for multiprocessor scheduling with communication delays. *LNCS: Algorithms and Architectures for Parallel Processing*, 7439:129–138, 2012.

[33] M. G. A. Verhoeven and E. H. L. Aarts. Parallel local search. *J. Heur.*, 1:43–65, 1995.

[34] M. Wodecki and W. Bozejko. Solving the flow shop problem by parallel simulated annealing. *LNCS*, (2328):236–247, 2002.