A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints

Stella C.S. Porto

(e-mail: stella@inf.puc-rio.br)

Celso C. Ribeiro

(e-mail: celso@inf.puc-rio.br)

Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
Rua Marquês de São Vicente 225
Rio de Janeiro 22453
Brazil

PUCRioInf-MCC03/93

July 1992

Revised January 1993, June 1994

Abstract

Parallel programs may be represented as a set of interrelated sequential tasks. When multiprocessors are used to execute such programs, the parallel portion of the application can be speeded up by an appropriate allocation of processors to the tasks of the application. Given a parallel application defined by a task precedence graph, the goal of task scheduling (or processor assignment) is thus the minimization of the makespan of the application. In a heterogeneous multiprocessor system, task scheduling consists in determining which tasks will be assigned to each processor, as well as the execution order of the tasks assigned to each processor. In this work, we apply the tabu search metaheuristic to the solution of the task scheduling problem on a heterogeneous multiprocessor environment under precedence constraints. The topology of the Mean Value Analysis solution package for product form queueing networks is used as the framework for performance evaluation. We show that tabu search obtains much better results, i.e. shorter completion times, improving from 20 to 30% the makespan obtained by the most appropriate algorithm previously published in the literature.

Keywords: Parallel processing, task scheduling, heterogenous processors, precedence constraints, makespan minimization, heuristics, tabu search.

Resumo

Programas paralelos podem ser representados por um conjunto de tarefas seqüenciais relacionadas. Quando multiprocessadores são utilizados para executar tais programas, a fração paralela da aplicação pode ser acelerada pela alocação apropriada de processadores às tarefas da aplicação. Dada uma aplicação paralela definida por um grafo de precedência, o objetivo do escalonamento de tarefas (ou alocação de processadores) é então a minimização do tempo total de sua execução. Em um sistema formado por processadores heterogêneos, o escalonamento de tarefas consiste em determinar quais tarefas devem ser alocadas a cada processador, assim como a odem de execução das tarefas alocadas ao mesmo processador. Neste trabalho aplica-se a metaheurística busca tabu à solução do problema de escalonamento de tarefas em um ambiente formado por multiprocessadores heterogêneos sob restrições de precedência. A topologia do grafo de tarefas associada ao algoritmo de análise do valor médio para redes de filas na forma produto é utilizada como plataforma para avaliação do desempenho da heurística. Mostra-se que a busca tabu obtém melhores resultados, isto é, tempos de execução menores, reduzindo de 20 a 30% os tempos de execução obtidos pelo algoritmo mais apropriado anteriormente publicado na literatura.

Palavras-chaves: Processamento paralelo, escalonamento de tarefas, processadores heterogêneos, restrições de precedência, minimização do tempo de execução, heurísticas, busca tabu.

1 Introduction

Parallel application programs can be represented as a set of interrelated tasks which are sequential units [8, 33]. When multiprocessors are used to execute such programs, the parallel portion of the application can be speeded up according to the number of processors allocated to the application. In a homogeneous architecture, where all processors are identical, the sequential portion of the application will have to be executed in one of the processors, degrading considerably the execution time of the application [2]. Menascé and Almeida [32] have proposed analytical models to improve the cost-effectiveness of a multiprocessor with heterogeneous architecture, where a larger processor tightly coupled to smaller ones is responsible for executing the serial portion of the parallel application, leading to higher performance. Recently, researchers at CMU carried out an experiment connecting a Cray YMP/832 to a 32K node Connection Machine CM-2 through a fast HIPPI data path. They were able to obtain a speedup of 10 in a distributed solution to the assignment problem, running the serial portions of the algorithm on the Cray while the parallel ones on the CM-2 [45].

In a homogeneous multiprocessor environment, one has to be able to determine the optimum number of processors to be allocated to an application (processor allocation), as well as which tasks are going to be assigned to each processor (processor assignment). In a heterogeneous setting, we not only have to determine how many, but also which processors should be allocated to an application, as well as which processors are going to be assigned to each task. Algorithms for processor assignment of parallel applications modeled by task precedence graphs in heterogeneous multiprocessor architectures have been proposed by Menascé and Porto [34]. The so-called greedy algorithms start from a partial solution and search to extend it until a complete assignment is achieved. At each step, one task assignment is done and this decision cannot be changed in the remaining steps. On the contrary, local search algorithms are initialized by a complete assignment and search to improve it by analyzing neighbor solutions.

Given a parallel application defined by a task precedence graph, task scheduling (or processor assignment) may be performed either statically (before execution) or dynamically (during execution). In the former case, there is no scheduling overhead to be considered during execution, but decisions are usually based on estimated values about the parallel application and the multiprocessor system. The work of each processor is defined at the time of compilation. More accurate information is used in a dynamic scheduling scheme. Each processor does not know *a priori* which task it will execute: processors are assigned to tasks during the execution of the application. To avoid overhead due to the scheduling procedure, processor assignment should be done very fast by a simple algorithm, eventually deteriorating the quality of the solution thus obtained. Contrarily, in the case of static scheduling, although less information is available, more sophisticated algorithms may be used since the compiler will be in charge of the assignment. The compilation time will certainly be longer, but the cost of task management should be smaller, since each processor will be ready in advance. Dynamic processor assignment is justified when the processors allocated to an application are not known beforehand, or when the execution times cannot be accurately estimated at the time of compilation. If the task precedence graph which characterizes the parallel application can be accurately estimated *a priori*, then a static approach is more attractive. Moreover, increasing compilation times is entirely justified for large scientific computation programs, where the execution times are much more relevant.

The scheduling problem involving the minimization of the maximum completion time on two uniform processors ($Q2 \mid\mid C_{max}$ in the notation of [30]) is already NP-hard [12, 13]. Approximate algorithms have been proposed for different versions of the problem studied in this work. The allocation problem in multiprogrammed homogeneous multiprocessors was studied by Sevcik [46] and Majumdar [31]. The issue is to determine how many processors should be allocated to the concurrent jobs (parallel applications). Each job is characterized through certain intrinsic parameters, namely serial fraction, average and maximum parallelism. The proposed heuristic allocation algorithms are based only on these parameters, which means that the internal structure of each job is not accurately known.

The problem of scheduling independent tasks to homogeneous parallel processors was studied by Kruskal and Weiss [27], with the goal of reducing the overall execution time. Adam *et al.* [1] also consider a homogeneous environment, but the parallel application is described by a precedence task graph. The scheduling problem considered by Hang *et al.* [26] is deterministic, non-preemptive and homogeneous. The system model, very similar to the one used by Porto and Menascé [38], can be used to model several types of systems such as a fully connected, a local area network, or a hypercube. To accomodate the deterministic scheduling approach, it is further assumed that the communication subsystem is contention-free. The algorithm adopts a simple greedy strategy: the earliest schedulable task is scheduled first. The starting time of each task is determined by several factors: when its preceding tasks are finished, how long the communication delays take, and where the task and its predecessors are allocated.

Automatic static parallelization schemes have been proposed e.g. in [37, 44, 49]. The approach of Sarkar and Hennessy [44] is appropriate to parallel programs described by task precedence graphs. Tasks are formed by the distribution of the iterations of parallel do-loops, or by the fusion of two parts of the sequential code in order to optimize data communication. Communication costs are taken into account explicitly and the method may then be used in distributed systems where the processors communicate by message passing, and not necessarily only in shared memory systems. A list algorithm is used for processor assignment. Polychronopoulos *et al.* [37] consider the parallelization of do-loops. The do-loop to be distributed is chosen among a set of nested do-loops. The choice criterion is an efficiency index associated with each do-loop. Tawbi [49] and Tawbi and Feautrier [50] consider shared memory architectures without interprocessor communication costs. Nested do-loops are automatically parallelized by a static approach. Simulated annealing and tabu search are compared for processor allocation. Again, a list algorithm is used for processor assignment. Implementation and computational results on the Encore Multimax machine are reported.

Similar to the scheduling problem, is the so-called mapping problem. The application is regarded as an undirected graph (the task interaction graph), where the nodes correspond to program tasks and their weights represent known or estimated computation costs. The edges indicate that the linked tasks interact during their lifetime, with edge weights reflecting the relative amounts of communication, without capturing any temporal execution dependencies. The parallel architecture is also seen as an undirected graph, with nodes representing processors and edge weights representing the cost of exchanging a unit message between them. A mapping aims at reducing the total interprocessor communication time and balancing the workload of the processors, thus attempting to find an allocation that minimizes the overall completion time. Sadayappan *et al.* [42] consider the task-processor mapping problem in the context of a local-memory multicomputer with a hypercube interconnection topology. Two heuristic cluster-based mapping strategies are compared: a nearest-neighbor approach and a recursive-clustering scheme. A hybrid approach is proposed, combining the characteristics of the heuristics and the use of a explicit cost function, which the authors claim to be the most attractive approach for the mapping problem. The nearest-neighbor mapping scheme explicitly attempts load balancing among clusters, whereas low communication costs are achieved implicitly through the use of a heuristic. In contrast, the recursive-clustering approach explicitly attempts to minimize communication costs, while load balancing is achieved implicitly by the search strategy. Following the same approach, Ercal *et al.* [10] also propose a task allocation scheme. The two phases of the recursive-clustering algorithm described earlier are merged. The essential idea is to make partial processor assignments to the nodes of the task graph during the recursive bipartitioning steps. The algorithm is compared with simulated annealing. A massively parallel genetic algorithm to the mapping problem and an implementation on a reconfigurable transputer network are proposed by Muntean and Talbi [36]. The population is mapped on a connected processor graph, one individual per processor. There is a bijection between the individual set and the processor set. The selection is done locally in a neighborhood of each individual. Tao, Narahari, and Zhao [48] considered the mapping problem in a heterogeneous parallel architecture. Three algorithms are proposed, based on different neighborhood search approaches, namely simulated annealing, tabu search, and stochastic probe. A candidate list with two different types of moves is used. The implementation of tabu search is very rudimentary and does not make use of all tools available, such as aspiration criteria, intensification and diversification phases. Some few numerical experiences are mentioned without details, indicating only slightly better results for the stochastic probe algorithm, with improvements never larger than 4% when compared to the cost of the solutions obtained by rudimentary implementations of the other two approaches.

Scheduling in a heterogeneous architecture is considered by Davis and Jaffe [9] and Horowitz and Sahni [25], but in these cases the tasks are independent. The task scheduling problem in a heterogeneous multiprocessor environment with applications represented by task precedence graphs was first considered by Porto and Menascé [34, 39]. A methodology for building heuristic static task scheduling algorithms was then proposed. Several algorithms were studied and compared based on simulation results. The focus of this work was on the processor assignment problem, assuming that processor allocation had already been determined. Communication demand between tasks and the costs due to interprocessor communication were not explicitly considered in that model. Recently, Porto and Menascé [38] extended the original model based on the assumptions of a loosely coupled multiprocessor architecture with a message passing communication scheme, now explicitly considering interprocessor communication. New algorithms were built and compared based on performance results obtained through a Markov chain model [51]. No other heuristic algorithms for this particular problem seem to be available in the literature [5, 6, 35].

In this work, we apply the tabu search metaheuristic to the static task scheduling problem in a heterogeneous multiprocessor environment under precedence constraints. The results obtained by tabu search are compared with those given by the most appropriate greedy algorithm in [34, 39]. We show that tabu search obtains better results, i.e. shorter completion times for parallel applications, using the schedule generated by such greedy algorithm as the initial solution to the search and systematically exploring a diversified solution set. The paper is organized as follows. In the next section, we formulate more accurately the task scheduling problem on heterogeneous processors under precedence constraints. In Section 3, the tabu search metaheuristic is viewed generically, while in Section 4 we present the resulting tabu search algorithm for the scheduling problem addressed in this work. Computational results are presented in Section 5. We first describe the performance evaluation framework, i.e., the general workload and system models, as well as the characteristics of the parallel applications used in the computational experiments and the criterion for comparing algorithm implementations and alternative solutions. The computational results show that tabu search obtains much better results, i.e. shorter completion times for parallel applications, improving from 20 to 30% the makespan obtained by the most appropriate algorithm previously published in the literature.

2 Problem Formulation

For the purpose of this paper, a heterogeneous parallel architecture is a set $P = \{p_1, \ldots, p_m\}$ of interconnected processors. Each processor has an instruction set of q execution time equivalent classes. The instruction execution times of a given processor $p_j \in P$ are represented by the vector $\Theta_j = (\theta_{1j}, \ldots, \theta_{qj})$, where θ_{ij} is the execution time of a type i instruction at processor p_j.

A parallel application Π is a set of partially ordered tasks. Let $T = \{t_1, \ldots, t_n\}$ be the set of tasks of Π and $G(\Pi)$ the (acyclic directed) task precedence graph associated with its tasks [4, 8, 33, 46]. Each node of this graph represents one of the tasks of the application. Arcs of the graph link a task to each of its immediate successors in the execution sequence. Associated with each task t_k we define a service demand vector $\Gamma_k = (\gamma_{1k}, \ldots, \gamma_{qk})$, where γ_{ik} is the average number of instructions of type i executed by task $t_k \in T$. Notice that in a homogeneous architecture, the service demand of a task can be measured in time units by a single scalar. In the heterogeneous environment it is not possible to measure the service demand in time units anymore, since the processors have different speeds. In a heterogeneous architecture, the execution time of a task depends on the processor that executes it. Hence, the execution time of task $t_k \in T$ at processor $p_j \in P$, denoted by $\mu(t_k, p_j)$, is given by $\Gamma_k \cdot \Theta_j$. Thus, a parallel application Π with n tasks and a heterogeneous multiprocessor system with m processors can be represented by a task precedence graph $G(\Pi)$ and an $n \times m$ matrix μ, with $\mu_{kj} = \mu(t_k, p_j)$ defined as above.

The problem of processor scheduling may be viewed as a two step process, namely processor allocation and processor assignment [52]. Processor allocation in a heterogeneous setting deals with the determination of which processors are to be allocated to a job. Processor assignment, or task scheduling, deals with the assignment of the already allocated processors to the tasks of the job. We consider in this paper only the problem of task scheduling in heterogeneous environments. Given a solution s for the scheduling problem, a processor assignment function is defined as the mapping $\mathcal{A}_s : T \rightarrow P$. A task $t_k \in T$ is said to be assigned to processor $p_j \in P$ in solution s if $\mathcal{A}_s(t_k) = p_j$. The task scheduling problem can then be formulated as the search for an optimal mapping of the set of tasks onto that of the processors, in terms of the overall makespan of the parallel application, i.e., the completion time of the last task being executed. At the end of the scheduling process, each processor ends up with an ordered list of tasks that will run on it as soon as they become executable.

A feasible solution s is characterized by a full assignment of processors to tasks, i.e., for every task $t_k \in T$, $\mathcal{A}_s(t_k) = p_j$ for some $p_j \in P$. A task $t_k \in T$ may be in one of the following states: *non-executable*, if at least one of its predecessor tasks is not executed; *executable*, if all its predecessor tasks are already executed but its own execution has not yet started; *executing*, if it is being executed (i.e., it is active); or *executed*, if it has already completed its execution in processor $\mathcal{A}_s(t_k)$. A processor $p_j \in P$ may be in one of the following states at a given time: *free*, if there is no active task allocated to it; or *busy*, if there is one active task allocated to it.

The maximum completion time (makespan) of a parallel application may be computed by an $\mathcal{O}(n^2)$ time labeling technique, using the precedence relations between tasks and average estimated execution times and service demand values given as characteristics of the application and system architecture. The procedure in Figure 1 describes the computation of the makespan of a parallel application. The *clock* variable measures the evolution of the execution. At the end of this procedure, $c(s) = clock$ is the cost of the current solution, i.e., the makespan of the parallel application given the task scheduling associated with solution s.

---

algorithm scheduler
begin
  Let $s = (\mathcal{A}_s(t_1), \ldots, \mathcal{A}_s(t_n))$ be a feasible solution for the scheduling problem, i.e.,
    for every $k = 1, \ldots, n$, $\mathcal{A}_s(t_k) = p_j$ for some $p_j \in P$
  $clock \leftarrow 0$
  $state(p_j) \leftarrow$ free $\forall p_j \in P$
  $start(t_k), finish(t_k) \leftarrow 0\ \forall t_k \in T$
  while $(\exists t_k \in T \mid state(t_k) \neq$ executed$)$ **do**
  begin
    for (each $t_k \in T \mid state(t_k) =$ executable) **do**
      if $(state(\mathcal{A}_s(t_k)) =$ free$)$ **then**
      begin
        $state(t_k) \leftarrow$ executing
        $state(\mathcal{A}_s(t_k)) \leftarrow$ busy
        $start(t_k) \leftarrow clock$
        $finish(t_k) \leftarrow start(t_k) + \mu(t_k, \mathcal{A}_s(t_k))$
      end
    Let i be such that $finish(t_i) = \min_{t_k \in T \mid state(t_k) = \text{executing}} \{finish(t_k)\}$
    $clock \leftarrow finish(t_i)$
    for (each $t_k \in T \mid state(t_k) =$ executing and $finish(t_k) = clock$) **do**
    begin
      $state(t_k) \leftarrow$ executed
      $state(\mathcal{A}_s(t_k)) \leftarrow$ free
    end
  end
  $c(s) \leftarrow clock$
end

Figure 1: Computation of the makespan of a given solution

3 Tabu Search

To describe the tabu search metaheuristic, we first consider a general combinatorial optimization problem (P) formulated as to

$$\begin{array}{ll} \text{minimize} & c(s) \\ \text{subject to} & s \in S, \end{array}$$

where S is a discrete set of feasible solutions. Local search approaches for solving problem (P) are based on search procedures in the solution space S starting from an initial solution $s_0 \in S$. At each iteration, a heuristic is used to obtain a new solution s' in the neighborhood $N(s)$ of the current solution s, through slight changes in s. Every feasible solution $\bar{s} \in N(s)$ is evaluated according to the cost function $c(.)$, which is eventually optimized. The current solution moves smoothly towards better neighbor solutions, enhancing the best obtained solution s^*. The basic local search approach corresponds to the so-called hill-descending algorithms, in which a monotone sequence of improving solutions is examined, until a local optimum is found.

Any hill-descending algorithm depends on two basic mechanisms: the initial solution heuristic and the neighbor search heuristic. The first should be capable of building from scratch an initial solution s_0. The neighbor search heuristic determines new neighbor solutions from a given current solution. In the most simple algorithm, it could be stated as a complete search for a neighbor solution with the lowest cost, without considering any criteria in the determination of which neighbor solutions would be effectively evaluated. In the case of the task scheduling problem, as defined in Section 2, the cost of a solution is given by its makespan, i.e., the overall execution time of the parallel application.

A move is an atomic change which transforms the current solution, s, into one of its neighbors, say $\bar{s}$. Thus, $movevalue = c(\bar{s}) - c(s)$ is the difference between the value of the cost function after the move, $c(\bar{s})$, and the value of the cost function before the move, $c(s)$. With these definitions, the description of a hill-descending algorithm in Figure 2 is straightforward.

---

algorithm hill-descending
begin
    Generate an initial solution s_0
    $s, s^* \leftarrow s_0$
    repeat
        $bestmovevalue \leftarrow \infty$
        for (all candidate moves) **do**
        begin
            Let $\bar{s}$ be the neighbor solution associated with the current candidate move
            $movevalue \leftarrow c(\bar{s}) - c(s)$
            if ($movevalue < bestmovevalue$) **then**
            begin
                $bestmovevalue \leftarrow movevalue$
                $s' \leftarrow \bar{s}$
            end
        end
        if ($bestmovevalue < 0$) **then** $s^* \leftarrow s'$
        $s \leftarrow s'$
    until ($bestmovevalue \geq 0$)
end

Figure 2: Basic hill-descending algorithm

It is clear through the description of the basic hill-descending algorithm, that this method always stops in the first local optimum. To avoid this drawback, several metaheuristics have been proposed in the literature, namely genetic algorithms, neural networks, simulated annealing, and tabu search [19]. They all have an essential common approach: the use of certain mechanisms which permit that the search for neighbor solutions take directions of increasing the cost of the current solution in a controlled way, as an attempt to escape from local optima. The current solution may not be the best solution so far encountered, which means that the best solution must be maintained separately throughout the execution of the algorithm. This class of techniques are called metaheuristics, because the process of finding a good solution (eventually the optimal one) consists of applying at each step a subordinate heuristic which has to be designed for each particular problem [14, 19, 24].

Among them, tabu search is an adaptive procedure for solving combinatorial optimization problems, which guides a hill-descending heuristic to continue exploration without becoming confounded by an absence of improving moves, and without falling back into a local optimum from which it previously emerged [15, 16, 17, 24, 29]. Briefly, the tabu search metaheuristic may be described as follows. At every iteration, an admissible move is applied to the current solution, transforming it into its neighbor with the smallest cost. Contrarily to a hill-descending scheme, moves towards a new solution that increase the cost function are permitted. In that case, the reverse move should be prohibited along some iterations, in order to avoid cycling. These restrictions are based on the maintenance of a short term memory function which determines how long a tabu restriction will be enforced or, alternatively, which moves are admissible at each iteration. Figure 3 gives a procedural description of the basic tabu search metaheuristic.

---

algorithm tabu-search
begin
Initialize the short term memory function
Generate the starting solution s_0
$s, s^* \leftarrow s_0$
while (moves without improvement $<$ $maxmoves$) **do**
begin
$bestmovevalue \leftarrow \infty$
for (all candidate moves) **do**
if (*candidate move* is admissible, i.e., if it does not belong to the tabu list) **then**
begin
Obtain the neighbor solution $\bar{s}$ by applying *candidate move* to the current solution s
$movevalue \leftarrow c(\bar{s}) - c(s)$
if ($movevalue < bestmovevalue$) **then**
begin
$bestmovevalue \leftarrow movevalue$
$s' \leftarrow \bar{s}$
end
end
end
if ($bestmovevalue > 0$) **then** update the short term memory function
if ($c(s') < c(s^*)$) **then** $s^* \leftarrow s'$
$s \leftarrow s'$
end
end

Figure 3: Basic description of the tabu search metaheuristic

The tabu tenure *nitertabu* is an important feature of the tabu search algorithm, because it determines how restrictive is the neighborhood search. The performance of an algorithm using the tabu search metaheuristic is intimately dependent on the basic characterizing parameters, namely the time that the short memory function enforces a certain move to be tabu, and the maximum number of iterations, $maxmoves$, during which there may be no improvement in the best solution. If the size of the tabu list is too small, the probability of cycling increases. If it is too large, there is a possibility that all moves from the current solution are tabu and the algorithm may be trapped. Sometimes the solution at this point is to reinitialize the short memory function, which means to get rid of the complete tabu list and to start the algorithm again with no restrictions.

However, it should be pointed out that cycle avoidance is not an ultimate goal of the search process. In some instances, a good search path will result in revisiting a solution encountered before. The broader objective is to continue to stimulate the discovery of new high quality solutions. One implication of choosing stronger or weaker tabu restrictions is to render smaller or longer tabu tenures appropriate [18]. For large problems, where $N(s)$ may have too many elements, or for problems where these elements may be costly to examine, the aggressive choice orientation of tabu search makes it highly important to isolate a candidate subset of the neighborhood, and to examine this subset instead of the entire neighborhood [18].

Succesfull applications of tabu search for combinatorial problems have been reported in the litterature, see e.g. [3, 11, 15, 21, 22, 23, 24, 28, 47, 53] among other references. Other advanced features, improvements and extensions to the basic tabu search procedure will be commented in the next section.

4 Task Scheduling by Tabu Search

The basic tabu search metaheuristic is now specialized into a specific algorithm for the task scheduling problem. This implies in turning the abstract concepts defined in Section 3, such as initial solution, solution space, and neighborhood, among others, into more concrete and implementable definitions.

As in Section 2, a solution s is here defined as any full assignment of tasks to processors, i.e., each task $t_k \in T$ is assigned to a certain processor $p_j \in P$ through the allocation function $\mathcal{A}_s(t_k) = p_j$. After completion of the scheduling process, there will be an ordered list of tasks associated with each processor. We assume that the tasks are numbered from 1 to n in a topological order in the beginning of the scheduling procedure, such that if t_i is a predecessor of t_j, then $t_i < t_j$.

Initial Solution. The initial solution s_0 is obtained through a greedy heuristic algorithm [34, 39], called (DES+MFT). This algorithm executes a deterministic simulation of the execution of the parallel application, very similarly to the process of obtaining the makespan of the application described in Figure 1. At each iteration, an executable task $t_k \in T$ is selected to be scheduled, taking the precedence constraints into account. The processor $p_j \in P$ which will be designated to execute this task is that which will presumably finish its execution first. Algorithm (DES+MFT) is a slight variant of algorithm (DES+MFTPO) mentioned in [34], both presenting practically identical performance results.

This heuristic benefits from the heterogeneity since it is able to perform a look-ahead during the deterministic simulation and decide whether it is advantageous to wait for a fast processor to become available, even though there might be some free slower processors to which the task could be assigned. We notice that after the construction of the initial solution, the lists of tasks associated with each processor may be not sorted following the original topological order. This would be the case if, for instance, two independent tasks $t_a < t_b$ (i.e., neither of them precedes the other), were scheduled in the initial solution to the same processor with the execution of t_b preceding that of t_a.

Neighborhood. A neighbor solution $\bar{s} \in N(s)$ is obtained by taking a single task $t_i \in T$ from the task list of processor $\mathcal{A}_s(t_i)$ and transfering it to the task list of another processor $p_l \in P$, with

$p_l \neq \mathcal{A}_s(t_i)$. The whole neighborhood is obtained by going through every task and then building a new solution by placing this task into every position of the task list of every other processor in the system. The cardinality of the neighborhood is clearly $O(n^2)$. Processors $\mathcal{A}_s(t_i)$ and p_l will be sometimes referred as p_{source} and p_{target} respectively.

In other words, the neighborhood $N(s)$ of the current solution s is the set of all solutions differing from s by only a single assignment. If $\bar{s} \in N(s)$, there is only one task $t_i \in T$ for which $\mathcal{A}_s(t_i) \neq \mathcal{A}_{\bar{s}}(t_i)$. A move is then the single change in the assignment function that transforms a solution s into one of its neighbors. Each move is characterized by a vector $(\mathcal{A}_s(t_i), t_i, p_l, pos)$, associated with the task $t_i \in T$ which will be taken out from the task list of processor $\mathcal{A}_s(t_i)$ and transferred to that of $p_l \in P$ in position pos.

Candidate lists. We have stressed the importance of procedures to isolate a candidate subset of moves from a large neighborhood, to avoid the computational expense of evaluating moves from the entire neighborhood. Candidate list strategies [18] implicitly have a diversifying influence by causing different parts of the neighborhood space to be examined on different iterations Candidate lists may be implemented by several strategies, as described by Glover *et al.* [20]: neighborhood decomposition, elite evaluation candidate lists, preferred attribute candidate lists, and sequential fan candidate lists. We used a preferred attribute candidate list approach, based on considering only a promising subset of the whole set of admissible moves.

A move has been characterized as the exchange of a task $t_i \in T$ from the processor $p_{source} = \mathcal{A}_s(t_i)$ where it is currently scheduled to a certain position pos of the task list of a different processor $p_{target} \in P$. Transferring this same task to a different position in the task list of processor p_{target} may generate a different execution order and, consequently, a different makespan. The number of neighbor solutions to be examined may be reduced by investigating only a few moves to some positions which most likely will lead to the best neighbor solution.

Consider a dynamic task enumeration scheme defined as follows. For the current solution s, consider the task list of each processor. The positions of the tasks in these lists define the order in which they will be offered to the scheduler, not necessarily the order in which they will be executed. The order of execution may be obtained by the makespan computation algorithm **scheduler** given in Figure 1. Renumber all tasks in a topological order according to the task precedence graph in such a way that if two tasks run on the same processor, the first one to be executed receives a smaller identification.

If a new task $t_i \in T$ is transferred to processor p_{target}, it is more likely that it should be placed in the task list of this processor (i.e., offered to the **scheduler** algorithm) after the last predecessor task and before the first successor task of t_i which are assigned to this same processor. Other positions in the task list of processor p_{target} are unlikely to be appropriate, since the task list could not be processed in its natural order. One possible candidate list strategy would consider then all these positions in the task list of processor p_{target} as possible moves for task t_i. In a more restrictive scheme implemented in this work, only one move is investigated, corresponding to placing t_i in between the only two consecutive tasks t_a and t_b assigned to p_{target} with current identifications satisfying $t_a < t_i < t_b$.

Tabu list. A chief mechanism for exploiting memory in tabu search is to classify a subset of the moves in a neighborhood as forbidden (tabu). The classification depends on the history of the search, particularly manisfested in the *recency* or *frequency* that certain move or solution components, called *attributes*, have participated in generating past solutions. Some choices of attributes may be better than others [18].

An attribute is defined to be *tabu-active* when its associated reverse (or duplicate) attribute has ocurred within a stipulated interval of recency or frequency in past moves. An attribute that is not tabu-active is said to be *tabu-inactive*. The condition of being tabu-active or tabu-inactive is called *tabu status* of an attribute [18]. A tabu attribute does not necessarily correspond to a tabu move. A move may contain tabu-active attributes, but still may not be tabu if these attributes are not sufficient to activate a tabu restriction. A move can be determined to be tabu by a restriction defined over any set of conditions on its attributes, provided these attributes are currently tabu-active.

The short memory function is represented by a finite list of tabu moves. If the best move $(p_{source} = \mathcal{A}_s(t_k), t_k, p_{target} = p_j)$ from the current solution deteriorates the cost function, the reverse move $(p_{target}, t_k, p_{source})$ should be prohibited along some iterations. The attribute which must be made tabu-active is defined as the pair (t_k, p_{source}), thus prohibiting task t_k to be scheduled again to processor p_{source} during a certain number of iterations. An $n \times m$ matrix *tabu* could be used to implement this short term memory. This matrix is initialized with zeroes. Whenever a move (t_k, p_{source}) is made tabu, we set $tabu(t_k, p_{source})$ to the current iteration counter plus the number of iterations *nitertabu* along which the move will be non-admissible, i.e., considered as a tabu move. Matrix *tabu* may then be used to keep track of the tabu status of every move.

Extended Tabu Lists. As discussed in the previous paragraph, the tabu attribute (PA) which is made active at each non-improving iteration has been defined until now as the pair (t_k, p_{source}) (task t_k is prohibited to be scheduled again to processor p_{source} during a certain tabu tenure). In some situations, it may be desirable to increase the number of available moves that receive a tabu classification. This may be achieved either by increasing the tabu tenure or by changing the tabu restriction [18]. Several other applications of tabu search (see e.g. [21, 28]) have shown that frequently it may be interesting to turn tabu-active certain attributes that not only avoid the reversal move towards the original solution, but also avoid a great variety of other moves towards other solutions which resemble the original one.

Using this approach, we may define other more restrictive tabu attributes following a move $(p_{source} = \mathcal{A}_s(t_k), t_k, p_{target} = p_j)$, such as: (i) prohibiting task t_k from leaving processor p_{target} (T); (ii) prohibiting any task from leaving processor p_{target} (PT); (iii) prohibiting any task to be scheduled to processor p_{source} (PS); and (iv) both constraints (ii) and (iii) are enforced (PTPS).

Aspiration criteria. Tabu conditions based on the activation of some move attributes may be too restrictive and result in forbidding a whole set of unvisited solutions which might be attractive. Tabu restrictions should not be inviolable under all circumstances. Aspiration criteria are introduced in the basic tabu search algorithm to identify tabu restrictions which may be override, thus removing the tabu status otherwise applied to a move [18]. A type of aspiration criterion consists in removing the tabu classification from a trial move when it leads to a solution better than that which was in the origin of the move which activated the corresponding tabu attribute.

A detailed description of the **tabu-schedule** algorithm is given in Figure 4. At each iteration the algorithm calls the procedure **obtain-best-move** described in Figure 5, which computes the best admissible move (t_k, p_j) from the current solution and handles the short term memory function.

---

algorithm tabu-schedule
begin
  Obtain the initial solution s_0
  Let *nitertabu* be the number of iterations during which a move is considered tabu
  Let *nmaxmoves* be the maximum number of iterations without improvement in the best solution
  Let *tabu* be a matrix which keeps track of the tabu status of every move
  { initialization }
  $s, s^* \leftarrow s_0$
  Evaluate $c(s_0)$
  $iter \leftarrow 1$
  $nmoves \leftarrow 0$
  for (all $t_i \in T$ and all $p_l \in P$) **do** $tabu(t_i, p_l) \leftarrow 0$
  { perform a new iteration as long as the best solution was improved in the last *maxmoves* iterations }
  while $(nmoves < maxmoves)$ **do**
  begin
    { search for the best solution in the neighborhood }
    obtain-best-move (t_k, p_j)
    { move to the best neighbor }
    Move to the neighbor solution s' by applying move (t_k, p_j) to the current solution s:
      set $\mathcal{A}_{s'}(t_k) \leftarrow p_j$ and $\mathcal{A}_{s'}(t_i) \leftarrow \mathcal{A}_s(t_i)\ \forall i = 1, \ldots, n$ with $i \neq k$
    $c(s') \leftarrow c(s) + bestmovevalue$
    { update the best solution }
    if $(c(s') < c(s^*))$ **then**
    begin
      $s^* \leftarrow s'$
      $nmoves \leftarrow 0$
    end
    { otherwise, update the number of moves without improvement }
    else $nmoves \leftarrow nmoves + 1$
    $s \leftarrow s'$
    $iter \leftarrow iter + 1$
  end
end

Figure 4: Algorithm **tabu-schedule** for the task scheduling problem

5 Computational Results

This section presents the computational results obtained by applying the tabu search algorithm to the scheduling problem in different situations. We first describe the basic framework for the workload and the multiprocessor system models, as well as the characteristics of the parallel applications used in the computational experiments and the criterion for comparing algorithm implementations and alternative solutions. The results of the tuning process for the tabu parameters are also presented. This process determines a certain *tabu configuration pattern*, which fully defines the tabu algorithm. Next, we present computational experiments for the evaluation of the **tabu-schedule** algorithm under several workloads and system configurations.

```
procedure obtain-best-move (t_k, p_j)
begin
    bestmovevalue ← ∞
    { scan all tasks }
    for (all t_i ∈ T) do
        for (all p_l ∈ P | p_l ≠ A_s(t_i)) do
            { check whether the move is admissible or not }
            if (tabu(t_i, p_l) < iter) then
            begin
                Obtain the neighbor solution s̄ by applying move (t_i, p_l) to the current solution s:
                    set A_s̄(t_i) ← p_l and A_s̄(t_r) ← A_s(t_r) ∀r = 1, ..., n with r ≠ i
                movevalue ← c(s̄) − c(s)
                { update the best move }
                if (movevalue < bestmovevalue) then
                begin
                    bestmovevalue ← movevalue
                    k ← i
                    j ← l
                end
            end
        end
    end
    { update the short term memory function }
    if (bestmovevalue ≥ 0) then tabu(t_k, A_s(t_k)) ← iter + nitertabu
end
```

Figure 5: Procedure **obtain-best-move**

5.1 Performance Evaluation Framework

We now describe the framework for evaluating the performance of the tabu search metaheuristic for the scheduling problem. This framework consists of some simplifying assumptions for the general workload and system models, as well as the characteristics of the parallel applications considered in the computational experiments.

- A deterministic model is used. In a deterministic model, the precedence relations between the tasks and the execution time needed by each task are fixed and known before a schedule (i.e., an assignment of tasks to processors) is devised. Although deterministic models are unrealistic, since they ignore e.g. variances in execution times of tasks due to interrupts and contention for shared memory, they make possible the static assignment of tasks to processors [40].

- Any processor is able to execute any task, i.e., they all have the same instruction set.

- There is only one *heterogeneous* or *serial processor* in P, which has the highest processing capacity. The remaining $m-1$ processors are called *homogeneous* or *parallel processors*.

- The processors are considered to be *uniform*, which means that the ratio between the execution time of any two tasks in any two processors is constant: $\frac{\mu(t_i,p_k)}{\mu(t_i,p_l)} = \frac{\mu(t_j,p_k)}{\mu(t_j,p_l)}$, $\forall k,l \in \{1,\ldots,m\}$, $\forall i,j \in \{1,\ldots,n\}$. With this assumption it is possible to consider that the instruction set has only one type of instruction.

- Given the previous assumption, let PPR be the *Processor Power Ratio* defined in [32], which measures the ratio between the execution time of any instruction in the homogeneous processor and its execution time in the fastest processor. The heterogeneity of the architecture is measured by the processor power ratio.
- The heterogeneity of the application is measured by the intrinsic serial fraction of the application, F_s, which is obtained through the procedure proposed in [46].

As described in Section 2, a parallel application may be seen as a set of tasks, characterized by their service demands and precedence relationships. We consider one unique topology for the precedence graph [55], typical of the Mean Value Analysis (MVA) solution package for product form queueing networks containing two classes of N customers each [41]. Figure 6 depicts an example of a $n = 25$ task graph associated with the MVA algorithm for $N = 4$. This dynamic programming scheme represents several computations qualified as *wave front*, because the topology is characterized by a regular mesh divided in two distinct phases. The first phase of the computations shows a slowly increasing parallelism, which slowly decreases during the second one. According to Zahorjan[54], the tasks at the border of the graph have half the size of the tasks in the inner part of the graph, since the latter have two parents and the former only one.

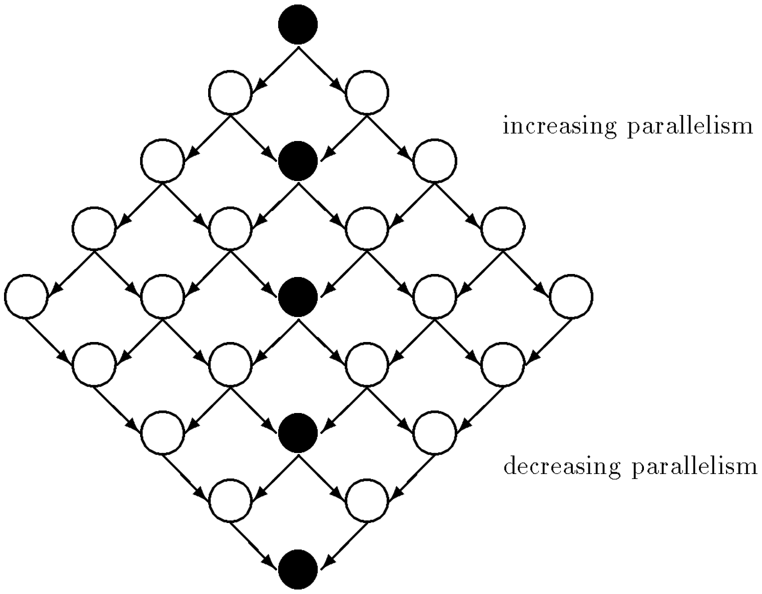

Figure 6: Task graph for an application of the MVA algorithm with two classes of $N = 4$ customers ($n = 25$ tasks)

The task precedence graph for a typical application of the MVA algorithm is shown in Figure 6. The dark nodes define the *vertical central axis* of the graph. The *horizontal central axis* is the middle horizontal axis, with the largest number of tasks. The number of tasks in the horizontal central axis equals $N+1$, where N is the number of customers in each class. For this same topology pattern we define different precedence graphs (i.e., different applications) by varying the size of the graph or the number of tasks, which is equal to the square of the number of tasks in the horizontal central axis.

For the same task precedence graph, different parallel applications may be obtained by changing the service demands of the tasks. Each parallel application has a different serial fraction, which depends on both its topology and the service demands of its tasks.

The main goal of the scheduling algorithm is to minimize the overall execution time of the parallel application. We recall that $c(s^*)$ and $c(s_0)$ represent, respectively, the makespan of the solution obtained by the tabu search approach and that of the initial solution given by algorithm (DES+MFT). Then, the most useful measure for the evaluation of the tabu search algorithm is the relative reduction provided by the tabu search algorithm with respect to the initial solution, i.e., the makespan relative reduction $\eta = (c(s_0) - c(s^*))/c(s_0)$. The performance of the tabu search scheduling algorithm will thus be evaluated through the makespan reductions obtained for different values of the application and system parameters: the number n of tasks, the serial fraction F_s, the number of processors m, and the processor power ratio PPR.

5.2 Tabu Parameters and Algorithm Robustness

The **tabu-schedule** algorithm for the task scheduling problem strongly depends on two numerical parameters, namely the tabu tenure *nitertabu* and the maximum number *maxmoves* of iterations without improvement. Its behavior also depends on the strategy implemented for the type and restrictiveness of the tabu list, as well as on the candidate set and aspiration criteria strategies.

Let a certain tabu configuration pattern be the set of tabu parameters and implementation strategies fully determined. Several tests were made in order to obtain the best tabu configuration pattern, which would provide the best performance for the **tabu-schedule** algorithm. This study was performed based on an application with the MVA topology, with the number of tasks in the horizontal axis ranging from 4 to 20 (accordingly, the number of tasks ranges from 16 to 400).

We have observed that the tabu search algorithm is quite robust. The quality of the solutions obtained does not seem to be much affected by different choices of parameter values and implementation strategies. The main reason for this behavior seems to be the efficiency of the candidate list strategy, which for every tabu configuration pattern discards most of the admissible moves and keeps only those leading to good solutions. Also, the characteristics of the system architecture (i.e., only one heterogeneous processor) and of the structure of the service demand (i.e., the task execution times) are such that many solutions have the same makespan and, consequently, many ties may be arbitrarily broken during the search without loosing the path to a good solution. As a result of this study, we have chosen to implement the **tabu-schedule** algorithm using the basic tabu list of type PA, $nitertabu = 20$ and $maxmoves = 100$. The candidate set and aspiration criterion strategies are those described in Section 4. This is the tabu configuration pattern used in the computational experiments reported below.

5.3 Computational Experiments

We present in this section the numerical results obtained with the application of the tabu search metaheuristic in the solution of the task scheduling problem under precedence constraints. We investigate the behavior of the **tabu-schedule** algorithm with the variation of the following parameters characterizing either the application or the architecture: the number of tasks in the graph, the serial fraction, the processor power ratio, and the number of processors. Bar graphs are used to illustrate this behavior, plotting the relative reduction in the makespan (i.e. η) of the best solution found with respect to the makespan of the initial solution given by the (DES+MFT) algorithm, against each of the parameters above.

5.3.1 Number of Tasks

As described in section 5.1, the number n of tasks in the MVA application is equal to the square of the number n_h of tasks in the horizontal axis, i.e., $n = n_h^2$. We have taken nine different graph sizes ranging from 16 to 400, corresponding to taking n_h equal to 4, 6, 8, 10, 12, 14, 16, 18, and 20. The service demand (size) of each task follows the standard characteristic of the MVA topology [54], i.e., the service demand of the tasks at the border of the graph is 1, while that of the inner tasks is 2. The processor power ratio was fixed at 5, while the number m of processors was made equal to one half of the number of tasks in the horizontal axis, i.e. $m = n_h/2$. The characteristics of the nine test applications are given in Table 1.

Application	Number n_h of tasks in the horizontal axis	Number n of tasks	Serial fraction	Number m of processors
P-01	4	16	0.416	2
P-02	6	36	0.250	3
P-03	8	64	0.178	4
P-04	10	100	0.139	5
P-05	12	144	0.114	6
P-06	14	196	0.096	7
P-07	16	256	0.083	8
P-08	18	324	0.073	9
P-09	20	400	0.066	10

Table 1: Characteristics of the application tests used in the analysis of the number of tasks

Figure 7 depicts the behavior of the makespan relative reduction obtained through the use of the **tabu-schedule** algorithm, with respect to the variation of the number n_h of tasks in the horizontal axis of the application. Significative makespan relative reductions ranging from 20 to 30% with respect to the (DES+MFT) algorithm [34, 39] may be observed. We may see that the relative reduction in the makespan seems to diminish only for very large task graphs. However, this behavior seems to be more a result of the characteristics of the test applications, as it will be described in section 5.4.

5.3.2 Serial Fraction

For the investigation of the behavior of the algorithm as a function of the serial fraction, we considered an MVA topology with $n_h = 10$ tasks in the horizontal central axis. Increasing values of the serial fraction were obtained by (i) taking the application test P-04 in the previous section with standard service demand (i.e., the tasks at the border of the graph have service demand equal to 1 and the inner tasks equal to 2) as a basic reference, and (ii) progressively increasing the

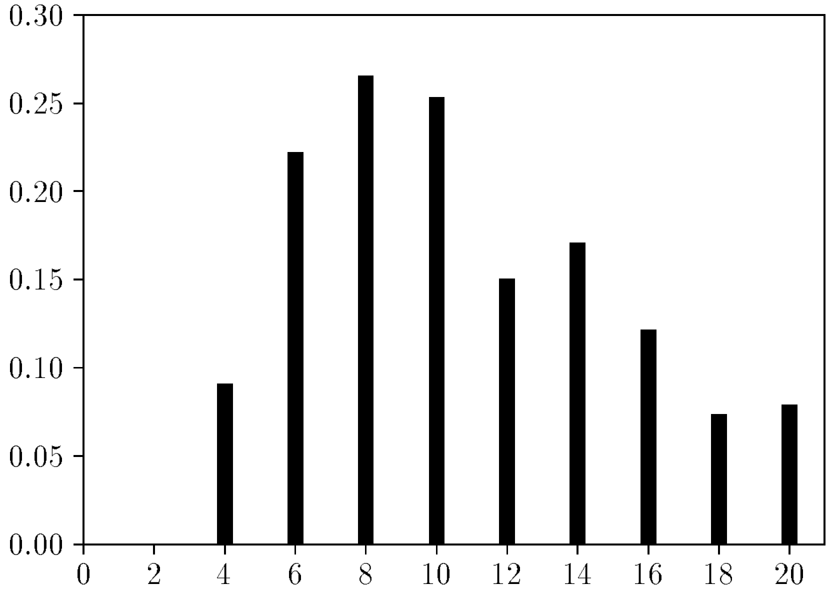

Figure 7: Relative reduction in the makespan versus the number of tasks in the horizontal axis (n_h)

service demands of the tasks in the vertical central axis. Keeping both the processor power ratio and the number of processors equal to 5, we considered the nine applications described in Table 2.

Application	Service demand of the tasks in the vertical axis	Serial fraction
P-04	2	0.139
P-10	4	0.214
P-11	8	0.333
P-12	12	0.471
P-13	16	0.585
P-14	32	0.783
P-15	64	0.889
P-16	100	0.928
P-17	200	0.964

Table 2: Characteristics of the application tests used in the analysis of the serial fraction

Figure 8 illustrates the gain in performance achieved with the **tabu schedule** algorithm according to the variation of the serial fraction. Best values for the makespan relative reductions, ranging from 30 to 40%, are obtained for values of the serial fraction between 0.20 to 0.60. The differences observed in the makespan relative reduction are due to the effects of the *serialization* phenomenon, further explained in section 5.4.

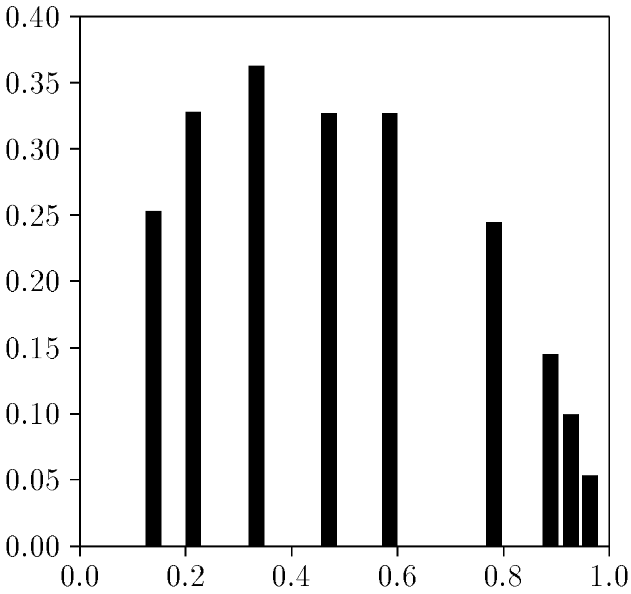

Figure 8: Relative reduction in the makespan versus the serial fraction (F_s)

5.3.3 Processor Power Ratio

The behavior of the scheduling algorithm is affected not only by the characteristics of the application, but also by those of the architecture. The system heterogeneity is characterized by the processor power ratio. Again, we have taken the basic application P-04 with $n = 100$ tasks and serial fraction 0.139 for the generation of additional problems, each of which with $m = 5$ processors and with the processor power ratio ranging from 2 to 30. The makespan relative reductions obtained are plotted against the processor power ratio in Figure 9.

5.3.4 Number of Processors

Finally, we investigated the behavior of the tabu search algorithm proposed in this paper with the variation of the number of processors. The same application P-04 was used as the basis for the evaluation. We take the processor power ratio equal to 5. The behavior of the makespan relative reduction with the variation of the number of processors from 2 to 10 is shown in Figure 10. The results in this figure show that makespan relative reductions ranging from 20 to 25%, i.e., of the same order of those reported in the previous sections, are again obtained by the tabu search metaheuristic.

5.4 Evaluation of the Relative Reduction in the Makespan

We say that an application is *serialized* by a processor assignment algorithm when all of its tasks are scheduled to one unique processor. When the serial fraction and/or the processor power ratio are very high, the best solution is usually obtained through the *serialization* of the application

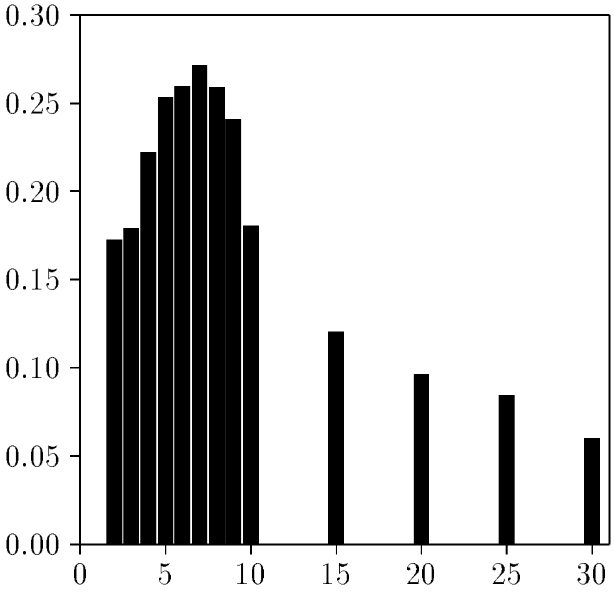

Figure 9: Relative reduction in the makespan versus the processor power ratio (PPR)

over the heterogenous processor, which has greater processing capacity. This seems to be clear if we imagine two extreme cases: $F_s = 1$ or $PPR \longrightarrow \infty$. In the first case, we face a totally serial application, and obviously it must be executed on a single processor, which will necessarily be the heterogeneous one. In the latter case, the heterogeneous processor is able to execute any task in an infinitesimal time, consequently serialization determines again the best performance.

Serialization is responsible for the shape of the curves describing the variation of the makespan relative reduction with both the serial fraction and the processor power ratio. This effect can be explained as follows. For very high serial fraction values or very high processor power ratio values, both the initial solution algorithm (DES+MFT) and the tabu search method tend to overload the heterogeneous processor through serialization, which in turn determines a low makespan relative reduction. By the same token, only for very small serial fractions and/or processor power ratios (DES+MFT) is able to make use of the parallelism offered by the application. Thus, also for low values of these parameters the initial solution resembles the one found by the tabu search and the makespan relative reduction is small. In the middle range of the F_s and PPR values, tabu search demonstrates a much better ability in distributing tasks across the processors, benefiting from the existing parallelism and system heterogeneity, and attaining very significant makespan relative reduction with respect to the (DES+MFT) algorithm.

As we may see from Table 1, the pattern of service demand of the test applications leads to decreasing serial fractions when the size of the task graph increases. Thus, according to the serialization phenomenon, for large task graph sizes both (DES+MFT) and tabu search will assign a very large number of tasks to the fastest processor, reducing the makespan relative reduction.

The effect of increasing the number of processors is the reduction in system resource contention. A very low resource contention demands very accurate resource management from the scheduling algorithm, so that the heterogenous processor does not become subutilized, reducing the benefit provided by heterogeneity. There is always a twofold commitment: profiting from system paral-

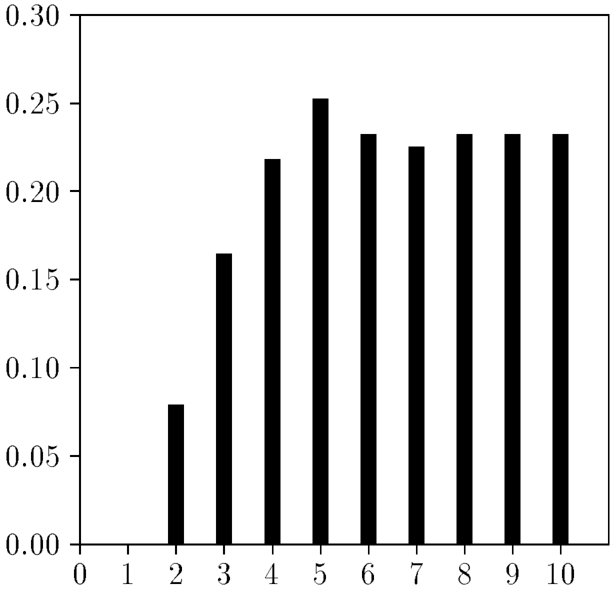

Figure 10: Relative reduction in the makespan versus the number of processors (m)

lelism, which means spreading tasks over all processors; and profiting from system heterogeneity, which means to concentrate tasks on the heterogeneous processor. Thus, for very high resource contention (i.e., a small number of processors), the initial solution and the final solution found by tabu search achieve similar performance. In this case, the available parallelism is small and the number of possibilities for rescheduling is still incipient. With very low resource contention (i.e, a large number of processors), the makespan relative reduction is reduced and stabilizes thereof. Neither algorithm is able to make use of extra processors, because the application has attained its maximum parallelism, beyond which there is no benefit in providing more processors.

6 Concluding Remarks

We have provided a new algorithm based on the tabu search metaheuristic for the task assignment problem on multiprocessor heterogeneous systems under precedence constraints. The topology of the Mean Value Analysis solution package for product form queueing networks was used as the framework for performance evaluation. We have shown that the tabu search algorithm obtains much better results, i.e. shorter completion times for parallel applications, improving from 20 to 30% the makespan obtained by the most appropriate algorithm previously published in the literature.

The quality of the solutions obtained by the tabu search algorithm does not seem to be much affected by different choices of parameter values and implementation strategies. The robustness of the algorithm seems to be due mainly to the efficiency of the candidate list strategy, which for every tabu configuration pattern discards most of the admissible moves and keeps only those leading to good solutions.

Further extensions of this work consist in the application of the tabu search algorithm to other parallel programs with different topologies for the task precedence graph, as well as in its application to task scheduling on message-passing architectures where interprocessor communication times are relevant.

References

[1] T.L. Adam, K.M. Chandy and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems", *Communications of the ACM* 17 (1974), 685-690.

[2] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capability", *Proceedings of the AFIPS Spring Joint Computer Conference* 30, 483-485, Atlantic City, 1967.

[3] S.G. de Amorim, J.-P. Barthelémy, and C.C. Ribeiro, "Clustering and Clique Partitioning: Simulated Anealing and Tabu Search Approaches", *Journal of Clasification* 9 (1992), 17–41.

[4] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, Englewood Cliffs, 1989.

[5] J. Blazewicz, *personal communication*, 1993.

[6] J. Blazewicz, K. Ecker, G. Schmidt, and J. Werglarz, *Scheduling in Computers and Manufacturing Systems*, Springer Verlag, Berlin, 1992.

[7] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[8] E.G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall Inc., New Jersey, 1973.

[9] E. Davis and J.M. Jaffe, "Algorithms for Scheduling Tasks on Unrelated Processors", *Journal of the ACM* 28 (1981), 721–736.

[10] F. Ercal, J. Ramajuan, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning", *Journal of Parallel and Distributed Computing* 10 (1990), 35–44.

[11] C. Friden, A. Hertz, and D. de Werra, "STABULUS: A Technique for Finding Stable Sets in Large Graphs with Tabu Search", *Computing* 42 (1989), 35–44.

[12] M.R. Garey and D.S. Johnson, "Strong NP-Completeness Results: Motivation, Examples and Implications", *Journal of the ACM* 25 (1978), 499–508.

[13] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[14] F. Glover, "Future Paths for Integer Programming and Links with Artificial Intelligence", *Computers and Operations Research* 13 (1986), 533–549.

[15] F. Glover, "Tabu Search – Part I", *ORSA Journal on Computing* 1 (1989), 190–206.

[16] F. Glover, "Tabu Search – Part II", *ORSA Journal on Computing* 2 (1990), 4–32.

[17] F. Glover, "Tabu Search: A Tutorial", *Interfaces* 20 (1990), 74–94.

[18] F. Glover and Manuel Laguna, "Tabu Search", to appear in *Modern Heuristic Techniques for Combinatorial Problems*, 1992.

[19] F. Glover and H.J. Greenberg, "New Approaches for Heuristic Search: A Bilateral Linkage with Artificial Intelligence", *European Journal of Operational Research* 39 (1989), 119–130.

[20] F. Glover, E. Taillard, and D. de Werra, "A User's Guide to Tabu Search", Working paper, 1991.

[21] P. Hansen, E.L. Pedrosa Filho, and C.C. Ribeiro, "Location and Sizing of Off-Shore Platforms for Oil Exploration", *European Journal of Operational Research* 58 (1992), 202–214.

[22] P. Hansen, M.V. Poggi de Aragão, and C.C. Ribeiro, "Boolean Query Optimization and the 0-1 Hyperbolic Sum Problem", *Annals of Mathematics and Artificial Intelligence* 1 (1990), 97–109.

[23] A. Hertz and D. de Werra, "Using Tabu Search Techniques for Graph Coloring", *Computing* 29 (1987), 345–351.

[24] A. Hertz and D. de Werra, "The Tabu Search Metaheuristic: How We Used It", *Annals of Mathematics and Artificial Intelligence* 1 (1990), 111–121.

[25] E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors", *Journal of the ACM* 23 (1976), 317–327.

[26] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times", *SIAM Journal of Computing* 18 (1989), 244–257.

[27] C.P. Kruskal and A. Weiss, "Allocating Subtasks on Parallel Processors", *IEEE Transactions on Software Engineering* 11 (1985), 1001–1009.

[28] M. Laguna, J.W. Barnes, and F. Glover, "Scheduling Jobs with Linear Delay Penalties and Sequence Dependent Setup Costs and Times Using Tabu Search", submitted to *Applied Intelligence*, 1990.

[29] M. Laguna, "Tabu Search Primer", Research report, University of Colorado at Boulder, Graduate School of Business and Administration, Boulder, 1992.

[30] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity", Report NFI 11.89/03, Eindhoven Institute of Technology, Department of Mathematics and Computer Science, Eindhoven, 1989.

[31] S. Majumdar, D.L. Eager, and R.B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proceedings of the International Conference on Parallel Processing*, 104–113, 1988.

[32] D.A. Menascé and V. Almeida, "Cost-Performance Analysis of Heterogeneity in Supercomputer Architectures", *Proceedings of the Supercomputing'90 Conference*, New York, 1990.

[33] D.A. Menascé and L.A. Barroso, "A Methodology for Performance Evaluation of Parallel Applications in Shared Memory Multiprocessors", *Journal of Parallel and Distributed Computing* 14 (1992), 1–14.

[34] D.A. Menascé and S.C.S. Porto, "Processor Assignment in Heterogeneous Parallel Architectures", *Proceedings of the IEEE International Parallel Processing Symposium*, 186–191, Beverly Hills, 1992.

[35] T.E. Morton and D.W. Pentico, *Heuristic Scheduling Systems with Applications to Production Engineering and Project Management*, Wiley, New York, 1993.

[36] T. Muntean and E.-G. Talbi, "A Parallel Genetic Algorithm for Process-Processors Mapping", *Proceedings of the Second Symposyum on High Performance Computing*, 71–82, Montpellier, 1991.

[37] C.D. Polychronopoulos, D.J. Kuck, and A.P. Padua, "Utilizing Multidimensional Loop Parallelism on Large-Scale Parallel Processor Systems", *IEEE Transactions on Computers* 38 (1989), 1285–1296.

[38] S.C. Porto and D.A. Menascé, "Processor Assignment in Heterogeneous Message Passing Parallel Architectures", to appear in *Proceedings of the Hawaii International Conference on System Science*, Kauai, 1993.

[39] S.C. Porto, *Heuristic Task Scheduling Algorithms in Multiprocessors with Heterogeneous Architectures: a Systematic Construction and Performance Evaluation* (in Portuguese), M.Sc. dissertation, Catholic University of Rio de Janeiro, Department of Computer Science, Rio de Janeiro, 1991.

[40] M.J. Quinn, *Designing Efficient Algorithms for Parallel Processors*, McGraw-Hill, New York, 1987.

[41] M. Reiser and S.S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks", *Journal of the Association for Computing Machinery* 27 (1980), 313–322.

[42] P. Sadayappan, F. Ercal, and J. Ramajuan, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube", *Parallel Computing* 13 (1990), 1–16.

[43] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, The MIT Press, Cambridge, 1989.

[44] V. Sarkar and J. Hennessy, "Compile-Time Partitioning and Scheduling of Parallel Programs", *ACM Sigplan Notices* 21 (1986), 17–26.

[45] M. Schneider, "Tying the Knot Between Serial and Massively Parallel Supercomputing: Pittsburgh's not-so-odd Couple", *Supercomputing Review* 4 (1991).

[46] K.C. Sevcik, "Characterization of Parallelism in Adaptation and Their Use in Scheduling", *Performance Evaluation Review* 17 (1989), 171–180

[47] J. Skorin-Kapov, "Tabu Search Applied to the Quadratic Assignment Problem", *ORSA Journal on Computing* 2 (1990), 33–45.

[48] L. Tao, B. Narahari, and Y.C. Zhao, "Heuristics for Mapping Parallel Computations to Heterogeneous Parallel Architectures", *Proceedings of the Workshop on Heterogeneous Processing*, 36–41, IEEE Computer Society Press, 1993.

[49] N. Tawbi, *Parallélisation automatique: estimation des durées d'exécution et allocation statique des processeurs*, Doctorate dissertation, Université Paris VI, Laboratoire MASI, Paris, 1991.

[50] N. Tawbi and P. Feautrier, "Processor Allocation and Loop Scheduling on Multiprocessor Computers", to appear in *Proceedings of ICS'92*, 1992.

[51] A. Thomasian and P. Bay, "Analytical Queueing Network Models for Parallel Processing of Task Systems", *IEEE Transactions on Computers* 35 (1986), 1045–1054.

[52] S.K. Tripathi and D. Ghosal, "Processor Scheduling in Multiprocessor Systems", *Proceedings of the First International Conference of the Austrian Center for Parallel Computation*, Springer Verlag, 1991.

[53] M. Widmer and A. Hertz, "A New Approach for Solving the Flow Shop Sequencing Problem", *European Journal of Operational Research* 41 (1989), 186–193.

[54] J. Zahorjan, *personal communication*, 1992.

[55] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", Technical Report 89-09-17, Department of Computer Science and Engineering, University of Washington, 1989.

n_h	η
4	0.091
6	0.222
8	0.265
10	0.254
12	0.151
14	0.171
16	0.121
18	0.074
20	0.079

F_s	η
0.139	0.254
0.214	0.328
0.333	0.363
0.471	0.327
0.585	0.327
0.783	0.245
0.889	0.145
0.928	0.099
0.964	0.053

PPR	η
2	0.172
3	0.179
4	0.222
5	0.254
6	0.260
7	0.272
8	0.259
9	0.241
10	0.181
15	0.121
20	0.096
25	0.084
30	0.060

m	η
2	0.079
3	0.164
4	0.218
5	0.253
6	0.232
7	0.225
8	0.232
9	0.232
10	0.232