Parallel Tabu Search Message-Passing Synchronous Strategies for Task Scheduling under Precedence Constraints

Stella C.S. Porto*
Dept. of Telecommunication Eng.
Universidade Federal Fluminense
Rua Passos da Pátria 156
Niterói 24210, RJ
Brazil
e-mail: stella@inf.puc-rio.br

Celso C. Ribeiro †
Department of Computer Science
Pontifícia Universidade Católica
Rua Marquês de São Vicente 225
Rio de Janeiro 22453-970, RJ
Brazil
e-mail: celso@inf.puc-rio.br

Abstract

This paper presents parallelization strategies for a tabu search algorithm for the task scheduling problem on heterogeneous processors under task precedence constraints. Parallelization relies exclusively on the decomposition of the solution space exploration. Four different parallel strategies are proposed and implemented on an asynchronous parallel machine under PVM: the master-slave model, with two different schemes for improved load balancing, and the single-program-multiple-data model, with single-token and multiple-token message passing schemes. The comparative analysis of these strategies shows that the tabu search approach for this problem is very suitable to the parallelization of the neighborhood search, with efficiency results almost always close to one for problems over a certain size.

Keywords: Task scheduling, tabu search, parallel algorithms, master-slave, SPMD.

1 Introduction

When parallel application programs are executed on MIMD machines, the parallel portion of the application can be speeded up according to the number of processors allocated to it. 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, considerably degrading the execution time. A faster processor tightly coupled to smaller ones, responsible for executing the serial portion of the parallel application, may lead to higher performance [13]. In a homogeneous multiprocessor environment, one has to be able to determine the optimum number of

*Work of this author was sponsored by the Brazilian Ministry of Education, through a CAPES scolarship in the framework of the PDEE program.

†Work of this author was sponsored by the CAPES/COFECUB Brazil-France agreement, in the framework of project 128/91.

processors to be allocated to an application (processor allocation), as well as which tasks will be assigned to each processor (processor assignment). In a heterogeneous setting, one has to determine not only 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. Deciding the processor on which a certain task will be executed is a more complex procedure in the homogenous case, where all processors have distinct processing speeds.

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 tasks it will execute, because processors are assigned to tasks during the execution of the application. To avoid the overhead due to the scheduling procedure, processor assignment should be done very fast by a simple algorithm, although this has the evident disadvantage of causing the quality of the resulting solution eventually to deteriorate. By contrast, 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 e.g. for large scientific programs, where the execution times are much more relevant. These applications present reliable estimates for task execution times, due to their known behavior and regularity. Thus, scheduling will be performed only once, and the program itself will be repeatedly executed, relying on few parameters whose management will not interfere with its processing time. Consequently, even if the scheduling effort is costly, its cost will be amortized over the many times the schedule will be re-applied.

The task scheduling problem in a heterogeneous multiprocessor environment with applications represented by task precedence graphs was first considered by Porto and Menascé in [14, 15]. The focus of this work was on the processor assignment problem, assuming that processor allocation had already been performed. Greedy algorithms for processor assignment of parallel applications modeled by task precedence graphs in heterogeneous multiprocessor architectures have been proposed and compared. More recently, Porto and Ribeiro [16] applied the *tabu search* metaheuristic to the static task scheduling problem under precedence constraints in a heterogeneous multiprocessor environment. The results obtained by tabu search improved by approximately 25% the makespan (i.e., the completion time) of the parallel applications, with respect to the schedule generated by the best greedy algorithm.

Tabu search is a local search 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 local optima from which it previously emerged [16]. In the case of the task scheduling problem considered in this work, where the precedence constraints determine a cost function whose evaluation takes $O(n^2)$ time complexity (where n stands for the number of tasks to be scheduled), the computational times required by any method that relies on such evaluations may be very large. This motivates the development of a parallel tabu search implementation, considering the promising results of applying this approach to other combinatorial problems already reported in the literature [1, 2, 4, 5, 6, 7].

In this paper we consider different parallelization strategies for the tabu search algorithm applied to the static task scheduling problem in a heterogeneous multiprocessor architecture. In Section 2 we present the formulation of the task scheduling problem. In Section 3 we briefly describe the tabu search approach for this problem. In Section 4 we describe the proposed parallel strategies. In Section 5 we present the test framework used for the computational experiments, followed by the numerical results and a comparative analysis of the different strategies. In Section 6 some concluding remarks are presented.

2 Problem Formulation

According to the same formulation used in [14, 16], a parallel application Π with a set of n tasks $T = \{t_1, \ldots, t_n\}$ and a heterogeneous multiprocessor system composed by a set of m interconnected processors $P = \{p_1, \ldots, p_m\}$ can be represented by a task precedence graph $G(\Pi)$ and an $n \times m$ matrix μ, where $\mu_{kj} = \mu(t_k, p_j)$ is the execution time of task $t_k \in T$ at processor $p_j \in P$. 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 makespan $c(s)$ 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 then characterized by a full assignment of processors to tasks. At any time instant, a task $t_k \in T$ may be in one of the following four states: *non-executable*, if at least one of its predecessor tasks was not yet executed; *executable*, if all its predecessor tasks were 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 a labeling technique [16]. Algorithm **makespan** in Figure 1 describes the computation of the makespan of a parallel application in $O(n^2)$ time complexity. 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 schedule associated with solution s.

algorithm makespan
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 $(\exists t_k \in T \mid state(t_k) \neq \text{executed})$ **do**
  begin
    for (each $t_k \in T \mid state(t_k) = \text{executable}$) **do**
      if $(state(\mathcal{A}_s(t_k)) = \text{free})$ **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 $t_k \in T \mid state(t_k) = \text{executing}$ 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 makespan

Figure 1: Computation of the makespan of a given solution

3 Tabu Search Applied to the Scheduling Problem

To describe the tabu search metaheuristic, we first consider a general combinatorial optimization problem formulated as follows

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

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. 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, $c(s)$. Every feasible solution $\bar{s} \in N(s)$ is evaluated according to the cost function $c(.)$, which is eventually optimized. The current solution moves progressively towards better neighbor solutions, enhancing the best obtained solution s^*. The basic local search approach corresponds to the so-called hill-descending algorithm, in which a monotone sequence of improving solutions is examined, until a local optimum (a solution whose cost value is better, or no worse, than that of each of its neighbors) is found.

Tabu search [8, 9, 10, 11] may be described as a higher level heuristic for solving combinatorial problems, designed to guide other hill-descending heuristics in order to escape from local optima. Thus, tabu search is an adaptive search technique that aims to intelligently explore the solution space in search of good, hopefully optimal, solutions. Broadly speaking [3], two mechanisms are used to direct the search trajectory. The first is intended to avoid cycling through the use of short term memories (tabu lists) that keep track of recently examined solutions. The second mechanism makes use of one or several memories, which may be referred to as long term memories, to direct the search either into a promising neighborhood (intensification), or towards previously unexplored regions of the solution space (diversification). It is noteworthy that these memory mechanisms may be viewed as learning capabilities that gradually build up images of good or promising solutions.

In the case of the task scheduling problem, the cost of a solution is given by its makespan. The neighborhood $N(s)$ of the current solution s is the set of all solutions differing from s by only a single assignment. If $\bar{s} \in N(s)$, then 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 taking out task $t_i \in T$ from the task list of processor $\mathcal{A}_s(t_i)$ and transferring it to that of $p_l \in P$ in position pos. However, 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 (candidate list). The most likely position is obtained through a dynamic task enumeration scheme, which is repeatedly applied each time the makespan of the solution is calculated [16]. If the best move takes the current solution s to a best neighbor solution s' degenerating its cost function, i.e. $c(s') \geq c(s)$, then the reverse move must be made tabu during a certain number of iterations in order to avoid cycling.

In our previous work [16], we proposed the tabu search algorithms described in Figures 2 and 3 to solve the scheduling problem defined in Section 2. A tabu configuration pattern was then defined to be a fully determined set of tabu parameters and implementation strategies. The proposed algorithm makes use of a simple tabu scheme and explores different tabu configuration patterns, each of which involves *(i)* a different type of candidate list, *(ii)* a different type of tabu list, *(iii)* a different value for $maxmoves$, which determines the maximum number of moves without improvement allowed during the search, *(iv)* a different value for $nitertabu$, which determines the number of iterations along which a move will be considered as tabu (i.e, prohibited), and *(v)* a different aspiration criterion, i.e. a special condition that, when satisfied by certain tabu moves, allow them to be accepted by disabling their tabu status. A series of different tabu configuration patterns were studied side-by-side with a variety of task precedence graphs (topology, number of tasks, serial fraction, service demand of each task) and system configurations (number of processors, architecture heterogeneity measured by the processor power ratio). The algorithm showed itself to be very robust and obtained very good results, systematically improving by approximately 25% the makespan of the solutions obtained by the best greedy algorithm used to provide an initial solution.

4 Parallel Strategies for the Task Scheduling Problem

According to Crainic *et al.* [3], a great number of tabu search procedures may be derived using different strategies when parallel implementations are contemplated. Parallel architectures

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 tabu-schedule

Figure 2: Tabu search algorithm **tabu-schedule** for the task scheduling problem

allow more efficient exploration of the solution space. Generally, this extra efficiency may be achieved by accelerating some phases of the algorithms, or by redesigning them. Several implementations have been proposed in recent literature for the parallelization of tabu search [1, 2, 4, 5, 6, 7, 18].

As mentioned by Fiechter [6], synchronous parallelization schemes generally require extensive communication and therefore are only worth applying to problems in which the computations performed at each iteration are complex and time consuming. This is exactly the case for our scheduling problem, in which the search for the best move at each iteration is a computationally intensive task. Due to the presence of the precedence constraints, the calculation of the cost of each solution implies a time complexity of $O(n^2)$ to compute each makespan evaluation, similar to a deterministic event simulation of the execution of the parallel application. Thus, from this point of view, the sequential tabu search algorithm for this problem is suitable for parallelization. Moreover, the size of the problems amenable to be solved in reasonable computational time by sequential search is rather limited and may certainly be increased by the use of a parallel scheme on a faster, more powerful parallel or distributed computer.

```
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 obtain-best-move
```

Figure 3: Procedure **obtain-best-move**

The design of parallel implementations for tabu search may use some basic ideas derived from the taxonomy presented in the work of Crainic *et al.* [3], whose main interest is to take into account the differences in control and communication strategies which are important when designing parallel algorithms. The proposed taxonomy has a twofold basis: first, concerning how the search space is partitioned; and second, concerning the control and communication strategies used by parallel tabu search procedures. The parallelization schemes proposed in this work are synchronized at the end of each iteration of the search. The search for the best neighbor during each iteration is performed in parallel and different sets of neighbor solutions are analysed by each task. This typically characterizes a strict domain decomposition parallelization scheme.

Two basic models are used, namely Master-Slave (MS) and Single-Program-Multiple-Data (SPMD), which mainly differ in the way information is exchanged between parallel tasks at the end of each iteration of the tabu search. Four different strategies derived from these two models are described with some detail in what follows. In order to refer to the tasks that compose the parallel program (described by the task precedence graph) associated with the task scheduling problem to be solved, we refer to them hereafter as *problem-tasks*. The tasks which effectively compose the parallel implementation of the tabu search algorithm, and which are distributed to the different processors, are given different designations depending on their role in each strategy, namely: slave-task, master-task, parent-task, child-task or simply tasks.

4.1 Master-Slave Model

The neighborhood of each current solution s is partitioned and a set of neighbor solutions to be evaluated is assigned to each slave-task. The most balanced way of dividing the neighborhood is to assign an equal number of problem-tasks to each slave-task. The slave-task will then only consider neighbor solutions obtained through moves of these pre-determined problem-tasks. The search for the best neighbor performed by each slave over a partition of the neighborhood is then called a best neighbor partial search. Moreover, as will be further explained, there is a choice in some cases of also giving a partition to the master, so that it can also be doing a best neighbor partial search, instead of being idle while slave-tasks are executing their search procedures. The number and size of partitions still offer different approaches to this strategy:

• **Single Partitioning (MS-SP):** The neighborhood is partitioned only once in approximately equal size partitions, depending on the number of slaves. The partitions of the neighborhood are distributed only once to all the slave-tasks, before the search starts. The master also receives one of the partitions and executes as if it had an embedded slave-task. The master initially distributes the problem and tabu configuration patterns. All tasks (master and slaves) use the same method to obtain the same initial solution. The master initializes the current and best solutions as the initial solution, so that the slaves do not need to keep track of the best solution. During the search, at each time a slave-task finishes its partial search, it sends to the master-task its best local non-tabu move, the cost of the corresponding best local neighbor, and a flag indicating if it eventually did not find any non-tabu move. The master compares its own partial best move with the values received from the slaves and proceeds as in the sequential version: it selects the move corresponding to the best neighbor solution, updates the best solution found so far and its cost, and verifies if the reverse move should be made tabu. If the termination condition is not verified, then it broadcasts the selected move to the slave-tasks, the tabu status of the reverse move and the cost of the eventually new best solution. The master and the slaves update their tabu lists, and restart the search with the same partitions. However, if any non-tabu move has not been found, the master sends to the slave-tasks a tabu list reinitialization message and another search within this same iteration will take place. In case of termination, the master sends to its slaves a final message, so that they can exit. Although the search in each task (master or slave) is done partially, their tabu lists are complete because they receive at each iteration the best move selected by the master with its corresponding reverse move tabu status.

• **Multiple Partitioning (MS-MP):** In order to meet load balancing requirements, when there are noticeable computational power differences between processors due to machine heterogeneity, load and contention discrepancies, the partition distribution may be done on a work-demand basis. Initially, the neighborhood is partitioned into equal parts as in the former case, but the number of partitions is sufficiently larger than the number of slave-tasks. The initial partitions are distributed (one per slave) by the master-task and kept by its slaves. Other initializations are done in the same way as before. During the search, each time a slave-task finishes its work, it sends to the master its partial search results. When the master receives results from a slave and there are still more partitions to be distributed, it sends this slave a new partition. When all partitions have been searched, the master already has the value of the best neighbor. It then proceeds as in the former case. In this way, more work is given to slave-tasks which are less loaded. On the other hand, it increases the communication between master and slaves, which may in turn reduce the gain obtained from the improved load balancing. It should be noticed that the size of the partitions is fixed and established before the search starts. In this case, as the master must always be available to send new partitions to the slave-tasks, it is not interesting to have it also doing best neighbor partial search as in the MS-SP strategy.

4.2 Single-Program-Multiple-Data Model

In this model, tasks work also in rigid synchronization, but there is no master-slave relationship as in the former strategies. All tasks execute the same code (SPMD model) following a token-ring communication structure, and the differentiation in the code stands for the task which spawns the others and makes the initialization of the token ring, which we call the parent-task. Tasks communicate pairwise, not strictly between parent and child. The principle here lies on the communication scheme. Tasks are organized in a logical ring and communicate according to this logical circular order established by the parent-task during the spawning procedure. This model can be subdivided into two different classifications, namely single token and multiple token, due to their resemblence to the correspondingly named access level of local network protocols:

• **Single Token (SPMD-ST):** Initialization is done in a manner similar to that of the former strategies. The parent-task reads problem and tabu configuration pattern files, spawns the other tasks, and broadcasts the necessary information. After the partitions have been sent, each task starts its best neighbor partial search. This strategy is very similar to that of the master-slave with single partitioning, because also here, all tasks work in parallel in the best neighbor partial search and receive identical size partitions. The difference relies mainly in the comparison step, which in this case is performed in a decentralized way, as explained in what follows. To complete each iteration, the parent-task takes the initiative of sending its successor the best move value it obtained. This initiative is responsible for starting the communication between tasks along the ring. All tasks wait to receive the best partial move from its predecessor in the logical ring. As each task receives the best move computed by its predecessor, it compares this move with its own result and passes on the best one to its successor. So, when the parent-task finally receives its predecessor message, closing the logical ring cycle, it knows that this is the best move, because it already passed by all other tasks around the ring. Then, it sends forward this best move to its successor, and the other tasks subsequently do the same, completing a second cycle of message passing around the logical ring. At the end of these two cycles, all tasks have the best move for that iteration. After receiving the global best move, each task proceeds independently updating the best solution found and its cost, the tabu list, the current solution and its cost, the number of iterations without improvement and the iteration counter. Each one then verifies the termination condition. If the search is to proceed, the next iteration is initiated immediately after the updating.

• **Multiple Token (SPMD-MT):** The initialization procedure is the same as for the single-token approach. The difference occurs at the end of the best neighbor partial search. Instead of waiting for the parent-task to initiate the best partial move message passing along the logical ring, each task sends its own result to its successor. When a task receives a result from its predecessor, it compares this information with its own result and sends the best one to its successor. Suppose there is a total of M tasks in the ring. Then, after $M-1$ messages have been received by each task, each of them necessarily has the best global move for this iteration. The updating phase follows, exactly in the same way as for the single token scheme.

The basic difference between the single token and the multiple token schemes is that the delay for waiting the token to pass around the ring in the former strategy is substituted by a greater number of simultaneous point-to-point messages between different task pairs in the latter. Let the tasks in the ring be numbered from 0 to $M-1$, with the parent being the 0-tagged task. In the single token approach, a total of $2M-1$ messages are sent around the ring. Tasks $0, 1, \ldots, M-2$ send two messages each: first, the result of the comparison between the move it found during its own best neighbor partial search and the move received from its predecessor; second, the best global move found during the iteration, which circulates through the ring. Task $M-1$ will send only one message, which necessarily is the best move, amounting to the total of $2M-1$ messages. In the multiple token scheme, there are $M(M-1)$ messages: each task among the M existing ones sends one message around the ring $M-1$ times. Now, suppose d is a constant delay taken by a message to leave from its source and to reach its destination. In the single token scheme there will be a total communication delay of $(2M-1)d$, since all messages are rigidly synchronized. However, in the multiple token model the communication delay may attain a best possible minimum of $(M-1)d$, due to the possible simultaneity of at most M messages each time.

5 Computational Experiments

In this section, we describe the framework for the computational experiments presented further. The hardware platform used for the implementation and performance evaluation of the parallel implementations of tabu search for the task scheduling problem is the IBM 9076 Scalable POWERparallel 1 (IBM SP1) machine working under PVM. The PVM package was used as the communication platform, due to its portability and flexibility. In terms of performance, as a tradeoff to portability, generally PVM does not profit from the hardware communication facilities available on specific parallel machines. However, there is a trend towards the design of parallel programs using more general platforms, in order to follow the speed of change in the computer system market. In this sense, PVM has been conforming itself as one possible standard in parallel programming development environments.

5.1 Configuration of the Test Problems

The characteristics which fully describe the scheduling problem are the same as those in Porto and Ribeiro [16]. An instance of our scheduling problem is characterized by the workload model and the system model. A deterministic model is used, in which the precedence relations between the tasks and the execution time needed by each task are fixed and known beforehand (i.e., before an assignment of tasks to processors is devised). Although deterministic models are unrealistic, since they ignore concerns such as deviations in task execution times due to interrupts and contention for shared memory, they make possible the static assignment of tasks to processors [17]. There is only one heterogeneous or serial processor, which has the highest processing capacity. The remaining $m-1$ processors are called homogeneous or parallel processors. Any processor is able to execute any task, i.e., they all have the same instruction set. The *processor power ratio* defined in [13] measures the ratio between the execution time of any instruction in the homogeneous processor and its execution time in the fastest processor. Then, the heterogeneity of the architecture is measured by the processor power ratio.

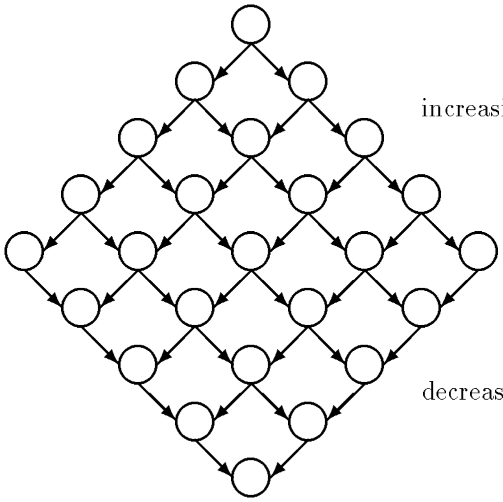

Figure 4: Task graph for an application of the MVA algorithm with $n = 25$ tasks

For the computational experiments, we have considered parallel applications with precedence graphs following the typical topology of the Mean Value Analysis (MVA) solution package for product form queueing networks[12, 19]. Figure 4 depicts an example of a task graph associated with the MVA algorithm for $n = 25$ tasks. The *horizontal central axis* is formed by the n_h tasks in the middle horizontal axis of the graph. For this same topology pattern, we define different precedence graphs (i.e., different applications) by varying the size of the graph, given by the number of tasks $n = n_h^2$. We have taken nine different graph sizes with the number n of tasks ranging from 16 to 400, corresponding to taking n_h ranging from 4 to 20, by steps of 2. The service demand of the tasks at the border of the graph is equal to one, while that of the inner tasks equals two [20]. The processor power ratio of the heterogeneous multiprocessor system considered in the scheduling problem was supposed to be equal to 5, while the number m of processors was made equal to one half of the number of tasks in the horizontal axis, i.e. $m = n_h/2$.

5.2 Tabu Configuration Pattern

The **tabu-schedule** algorithm for the task scheduling problem strongly depends on two 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 list and aspiration criteria strategies.

In Porto and Ribeiro [16], several experiments 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 6 to 14 (accordingly, the number of tasks ranges from 36 to 196). We have retained the following tabu configuration pattern as the outcome of these experiments: *(i)* the candidate list is built through a dynamic enumeration technique determining only a single position for the moving task in the task list of each target processor, *(ii)* the tabu list is organized as matrix, in which each element $[i, j]$ holds the last iteration value until which the move of task $t_i \in T$ to processor $p_j \in P$ is prohibited (i.e., tabu), *(iii)* $maxmoves = 100$, *(iv)* $nitertabu = 20$, and *(v)* the aspiration criterion establishes that a certain tabu move drops its tabu classification if it takes the current solution s to a neighbor solution which improves upon the best solution s^* found so far.

5.3 Parallel Strategies

Let q be the number of processors which work together throughout the execution of the parallel tabu search algorithm. In the case of the master-slave strategies $q = 1 + nslaves$, where $nslaves$ stands for the number of slave-tasks. For the SPMD strategies $q = 1 + nchildren$, where $nchildren$ stands for the number of child-tasks which compose the logical ring together with the single parent-task. The computational experiments have been performed for different number of processors ($4 \leq q \leq 16$).

The MS-MP strategy is also characterized by the partition size. The smaller the partition size, the smaller the granularity of the work dispatched to the slaves by the master. Load balance may be done more accurately, although more communication will take place during each iteration until the whole neighborhood is searched. Consequently, there will be a threshold for the partition size, beyond which good performance will not be attained. This threshold may depend not only on the problem size and the number of processors, but also on external factors such as the current system load. We have partitioned the neighborhood in $2 \cdot nslaves$ partitions in the case of MS-MP, corresponding to an average granularity of two partitions per slave.

5.4 Numerical Results

The performance of a parallel strategy may be evaluated by its speedup $\mathcal{S}$ and its efficiency η defined as:

$$\mathcal{S} = \frac{\tau_{seq}}{\tau_{par}} \quad \text{and} \quad \eta = \frac{\mathcal{S}}{q},$$

where τ_{seq} and τ_{par} are, respectively, the sequential and parallel elapsed times observed for the sequential algorithm and this parallel strategy for the tabu search algorithm. For the sequential algorithm we were faced with the choice of using either the sequential algorithm previously developed in [16] or the parallel version presented in this paper with one single processor. As the speedup is more accurately measured using the best known sequential algorithm, we have used the first option to calculate speedup and efficiency values in our experiments.

All four parallel strategies start from the same initial solution, since they use the same heuristic to produce this initial solution. They also generate an identical final solution. Therefore, the quality of the solutions obtained by the four strategies is the same, and the strategies may be compared considering exclusively their attained speedup with respect to the sequential algorithm.

Samples of the results obtained on the IBM SP1 machine are presented in Figures 5 to 7. All four strategies led to rather similar results, and we show only selected results in our figures.

Linear speedup is attained when the efficiency is equal to one, i.e. the speedup is equal to the number of processors q. The closer the efficiency is to one, the more the parallelization scheme benefits from the system parallelism. Differences in code and data distribution for sequential and parallel versions of the algorithm may generate distinct memory access latencies. Moreover, other applications in the system impact differently the performance of the sequential and parallel tabu search algorithms. The sequential version is more affected by other applications, as far as it is executed on a single processor. We notice that, in the particular IBM SP1 system used at the LMC/IMAG (Grenoble), slave tasks were run in most cases on less loaded nodes. Also, as the parallel algorithm presents communication-computation overlap, the overhead due to the communication introduced by the parallelization is minimized as a factor of performance degradation. As a consequence, efficiency values greater than one were obtained in some experiments with problems of larger sizes.

On the other hand, we can observe very low efficiency values for small problems. In these cases, the neighborhood partitions distributed among parallel tasks are not sufficiently large to overcome the embedded overhead due to synchronization between cooperative tasks in the parallel algorithm. This effect is even greater for large values of q, which also contribute to decrease the sizes of the neighborhood partitions. Because the efficiencies have been calculated using the elapsed times, the discrepancies in Figures 5 to 7 seem to be due to the fact that the different cases were run throughout a long period of time, during which the machine was subject to different workloads, consequently producing a different impact on the elapsed time of each test problem and strategy.

Figure 5 illustrates the behavior of the efficiency according to the problem size for different number of processors (q), in the case of MS-SP and MS-MP strategies. Both strategies present very similar results. However, the load balancing aspect of the MS-MP strategy is significative if one considers that, in this scheme, the master-task does not execute any partial best neighbor search. The solution neighborhood to be searched is divided solely among slave-tasks. The master rests as a partition distributer and is in charge of the comparison and election of the best move at each iteration. Thus, the importance of load balancing is so effective that it compensates by far the decrease in the computational power dedicated to the best neighbor search. In this case, the comparisons between successive partial best moves is immediate, because the master is completely dedicated to the reception of the results sent by the slaves.

We present in Figure 6 the behavior of the efficiency according to the number of processors for different problem sizes, in the case of the MS-SP strategy. We notice from these figures that the efficiency systematically (i) increases with the problem size for the same number of processors and (ii) decreases with the number of processors for the same problem size, due to the decrease in the communication/computation ratio.

Figure 7 shows performance results for the SPMD-ST strategy. The comparisons cycle is dependent on the parent-task initiative and on the natural order of the ring. Each child-task only sends the message with its own best partial move when it has already received a message from its predecessor in the ring. However, due to the small size of the messages and to the fast communication media, the two cycles of message exchange do not produce high overhead. The performance gain due to the overlap between communication and computation is sufficient to compensate the rigid synchronism established by this strategy. Similar results were obtained with the SPMD-MT strategy The greater number of messages in the case of SPMD-MT compensates higher communication synchronization in SPMD-ST.

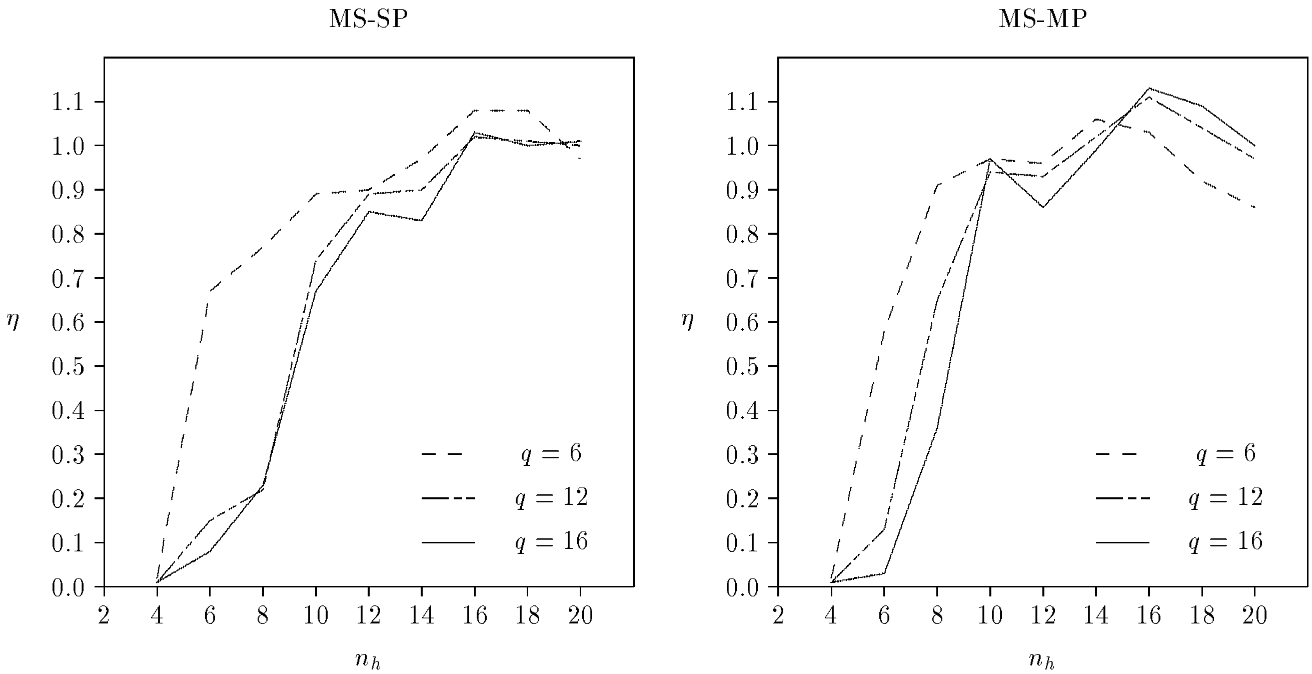

Figure 5: Efficiency (η) vs. problem size (n_h): MS-SP and MS-MP strategies

The interesting aspects of the logical ring communication scheme, following either a single token or a multiple token approach, are the decentralization of the communication and the parallelization of the procedure for the comparison of the best partial moves. Although the logical ring communication scheme could apparently be more promising in terms of the overall performance, specially in the multiple token case, where many pairs of tasks may communicate simultaneously, it should be noticed that this advantage is totally dependent on the architecture of the parallel machine. The standard PVM does not profit from the architectural characteristics to perform message diffusion. Thus, communication on the IBM SP1 under PVM is done as if processors were workstations linked by a local Ethernet network. The access method to the communication media in this topology assures that only one communication will be in execution at each time. In this case, it may be possible that no profit results from the use of the logical ring communication scheme.

6 Conclusions

We have presented four different message-passing parallelization strategies for the implementation of a tabu search algorithm developed for the solution of a task scheduling problem under precedence constraints on heterogeneous processors. All strategies are synchronous and their differences rely exclusively on distinct information communication patterns between parallel tasks during execution. The parallel programs were implemented on a IBM SP1 machine under PVM for varying problem sizes and number of processors. The computational results confirm the great adaptability of this kind of algorithm to parallelization, showing that communication is not a burden to achieving almost ideal efficiency in the majority of the test problems. The task scheduling problem considered in this study is characterized by very large neighborhood structures that are costly to explore. As we have observed, recent studies in the literature have similarly reported performance results for the parallelization of tabu search for other

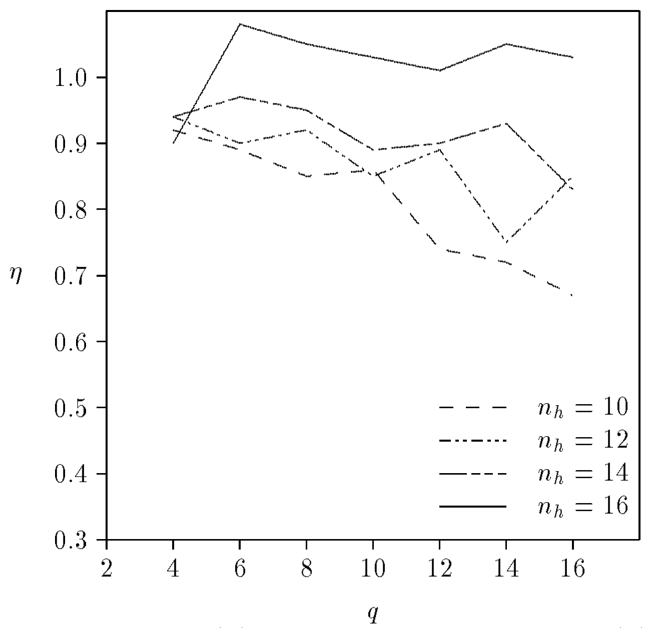

Figure 6: Efficiency (η) vs. number of processors (q): MS-SP strategy

problems with related structures. We suggest that future research to obtain still better outcomes for problems that pose such inherently intensive computational demands will benefit by considering more intricate tabu search features and asynchronous parallelization schemes.

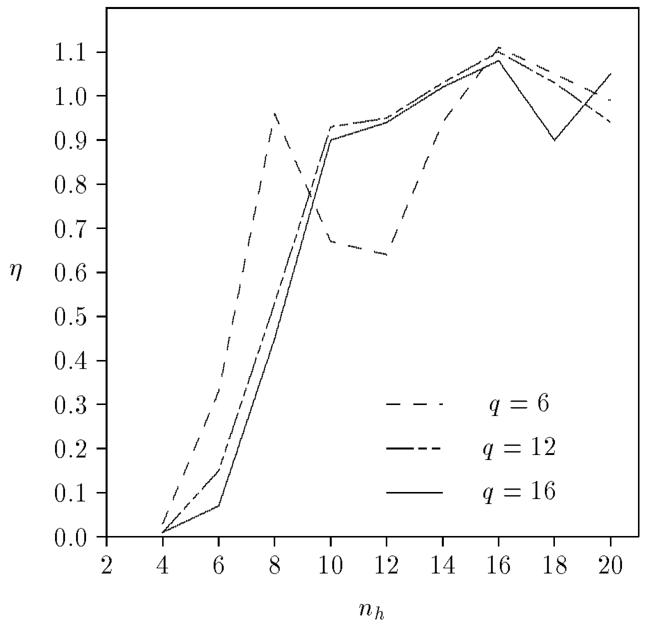

Figure 7: Efficiency (η) vs. problem size (n_h): SPMD-ST strategy

Acknowledgements. We acknowledge Denis Trystram for his technical remarks in the initial phase of this work, and for making available the facilities and the computational resources of the *Laboratoire de Modélisation et Calcul* at the *Université Joseph Fourrier* (Grenoble).

References

[1] J. Chakrapani and J. Skorin-Kapov, "Massively Parallel Tabu Search for the Quadratic Assignment Problem", Technical Report HAR-91-06, Harriman School, Stony Brook, 1992.

[2] J. Chakrapani and J. Skorin-Kapov, "Mapping Tasks to Processors to Minimize Communication Time in a Multiprocessor System", Working paper, 1993

[3] T.G. Crainic, M. Toulouse and M. Gendreau, "Towards a Taxonomy of Parallel Tabu Search Algorithms", Research Report CRT-933, Centre de Recherche sur les Transports, Université de Montréal, 1993.

[4] T.G. Crainic, M. Toulouse and M. Gendreau, "A Study of Synchonous Parallelization Strategies for Tabu Search", Research Report CRT-934, Centre de Recherche sur les Transports, Université de Montréal, 1993.

[5] T.G. Crainic, M. Toulouse and M. Gendreau, "An Appraisal of Asynchronous Parallelization Approaches for Tabu Search Algorithm", Research Report CRT-935, Centre de Recherche sur les Transports, Université de Montréal, 1993.

[6] C.-N. Fiechter "A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems", *Discrete Applied Mathematics* 51 (1994), 243–267.

[7] B. Garcia and M. Toulouse, "A Parallel Tabu Search for the Vehicle Routing Problem with Time Windows", to appear in *Computers and Operational Research*, 1994.

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

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

[10] F. Glover and Manuel Laguna, "Tabu Search", Chapter 3 in *Modern Heuristic Techniques for Combinatorial Problems* (C.R. Reeves, ed.), 70–150, Blackwell Scientific Publications, Oxford, 1992.

[11] F. Glover, E. Taillard, and D. de Werra, "A User's Guide to Tabu Search", *Annals of Operations Research* 41 (1993), 3–28.

[12] 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.

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

[14] 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.

[15] S.C.S. Porto, *Algoritmos Heurísticos para o Escalonamento de Tarefas em Multiprocessadores Com Arquitetura Heterogênea: Construção Sistemática e Avaliação de Desempenho* M.Sc dissertation, Catholic University of Rio de Janeiro, Department of Computer Science, Rio de Janeiro, 1991.

[16] S.C.S. Porto and C.C. Ribeiro, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints", *International Journal of High Speed Computing* 7 (1995), 45–71.

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

[18] E. TAILLARD, "Robust Tabu Search for the Quadratic Assignment Problem", *Parallel Computing* 7 (1991), 443–455.

[19] 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.

[20] J. ZAHORJAN, *personal communication*, 1992.