# A Case Study on Parallel Synchronous Implementations for Tabu Search Based on Neighborhood Decomposition

**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**

**Stella C.S. Porto**†
**Dept. of Telecom. Engineering**
**Universidade Federal Fluminense**
**Rua Passos da Pátria 156**
**Niterói 24210, RJ**
**Brazil**
**e-mail: stella@caa.uff.br**

PUCRioInf-MCC03/96

January 1996

**Abstract:**

We study in this paper different synchronous strategies for the parallel implementation of tabu search on a parallel machine. The task scheduling problem on heterogeneous processors under precedence constraints is used as the framework for the development, implementation, validation, and performance evaluation of different parallel strategies. Several strategies are proposed, discussed and compared: 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 schemes for message passing. The IBM SP1 parallel machine running PVM is used for the implementation and the evaluation of these strategies. Computational results are presented and the behavior of the different strategies is discussed, evaluated, and compared.

**Keywords**: Task scheduling, tabu search, parallel algorithms, distributed algorithms, master-slave, load balancing, SPMD, message passing.

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

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

**Resumo:**

Estuda-se neste trabalho diferentes estratégias síncronas para a implementação paralela da busca tabu em uma máquina paralela. O problema de escalonamento de tarefas em processadores heterogêneos sob relações de precedência é usado como estrutura básica para o desenvolvimento, implementação, validação, e avaliação de desempenho de diferentes estratégias paralelas. Diversas estratégias são propostas, discutidas e comparadas: o modelo mestre-escravo, com dois esquemas diferentes para melhor balanceamento de carga, e o modelo SPMD ("Single-Program-Multiple-Data"), com esquemas de passagem de mensagem de ficha única e múltiplas fichas. A máquina paralela IBM SP1 rodando PVM é usada para a implementação e avaliação destas estratégias. Resultados computacionais são apresentados e o comportamento das diferentes estratégias é discutido, avaliado e comparado.

**Palavras-chaves**: Escalonamento de tarefas, busca tabu, algoritmos paralelos, algoritmos distribuídos, mestre-escravo, balanceamento de carga, SPMD, troca de mensagens.

# 1 Introduction

Parallel application programs may be represented as a set of interrelated sequential tasks [6, 24]. 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 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 of the application [1]. A larger processor tightly coupled to smaller ones, responsible for executing the serial portion of the parallel application, may lead to higher performance. 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 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.

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). 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 the compilation times is entirely justified for large scientific computation programs, where the execution times are much more relevant.

The task scheduling problem in a heterogeneous multiprocessor environment with applications represented by task precedence graphs was first considered by Porto and Menascé in [25, 27]. 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 [28] 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 considerably 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 the absence of improving moves, and without falling back into local optima from which it previously emerged [28]. Tabu search has the advantage of being easy to implement for numerous problems. 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 observed for tabu search may be very large. In this sense, stands out the idea of a parallel implementation of this algorithm, considering the promising results focusing other combinatorial problems already reported in the literature [2, 4, 5, 8, 9, 10, 11, 30]. Parallel computers offer the possibility to design procedures that explore more efficiently the solution space and in less time. In Section 2 we present the formulation of the task scheduling problem under precedence constraints. In Section 3 we briefly describe the tabu search approach for the task scheduling under precedence constraints problem. Section 4 presents how the makespan computation can be reduced, based on the previous calculation of the cost of a neighbor solution. Section 5 establishes with more details the motivation for a parallel implementation of this algorithm. Section 6 is devoted to the presentation of a taxonomy of parallel strategies for tabu search. In Section 7 we describe in details the parallel strategies for the task scheduling problem. In Section 8 we present the IBM SP1 parallel machine and the PVM software package used for the implementation and computational experiments of the tabu search parallel strategies for the scheduling problem. In Section 9 we present the test framework used for the computational experiments, i.e., the description of the test problems, tabu configuration parameters and parallel srategies, followed by the numerical results obtained by the different strategies. Section 10 presents a new strategy for the neighborhood search, in which the move definition is changed in order to exploit the solution space more thoroughly. In Section 11 we discuss the computational results and some concluding remarks are presented.

# 2 Problem Formulation

A parallel application $\Pi$ 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 $\mu$, 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 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 $p_j \in P$. 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, using the precedence relations between tasks and average estimated execution times and service demand values given as characteristics of the application and system architecture [28]. 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** $(\exists t_k \in T \mid state(t_k) \neq$ 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 makespan**

---

Figure 1: Computation of the makespan of a given solution

# 3 A Tabu Search Heuristic

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 $s$. 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 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.

Tabu search [13, 14, 15, 16, 17, 18] may be described as a higher level heuristic for solving minimization 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 [7], two mechanisms are used to direct the search trajectory. The first is intended to avoid cycling through the use of tabu lists, which work as short term memories that keep track of recently examined solutions. The second mechanism makes use of one or several memories, which may be referred as long term memories, to direct the search either into the exploration of 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. The learning capability determines that tabu search supplies knowledge about the instance of the problem to be solved richer than that generated in other iterative algorithms.

In the case of the task scheduling problem considered in this paper, the cost of a solution is given by its makespan, i.e., the overall execution time of the parallel application. The neighborhood $N(s)$ of the current solution $s$ is the set of all solutions differing from it by only a single assignment. If $\bar{s} \in N(s)$, then there is only one task $t_i \in T$ for which $\mathcal{A}_s(t_i) \neq \mathcal{A}_{\bar{s}}(t_i)$. Each move is fully 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. 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 [28]. Consequently, the move may be characterized by a simple restricted representation given by $(\mathcal{A}_s(t_i), t_i, p_l)$, as far as the position task $t_i$ will occupy in the task list of processor $p_l$ is uniquely defined. The resulting algorithm **RN-STM-TS** (Restricted Neighborhood Short-Term Memory Tabu Search) is presented in Figure 2, and is described in detail in [28].

In our previous work [28] considering the application of tabu search to the scheduling problem described in Section 2, intensification and diversification strategies were not explored. The algorithm described in Figures 2 and 3 makes use of a simple tabu scheme and explores different tabu configuration patterns, each of which involving (i) a different construction for the restrict neighborhood, (ii) a different construction for the tabu lists (i.e., the moves which are made tabu at each iteration), (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 tabu tenure, i.e. the number of iterations along which a move will be considered as tabu (i.e, prohibited), and (v) a different aspiration criterion, i.e. special conditions that, when satisfied by certain tabu moves, allow them to be accepted desabling their tabu status. 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). Although the size of the problems (i.e., the number of tasks in the parallel application) involved in these computational experiments using the tabu search sequential algorithm has been limited due to hardware and computer time availability, the algorithm showed 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.

Some of the parameters involved in the definition of the tabu configuration patterns did not show themselves to have great influence on the results attained by the algorithm. We have retained the following strategy as that which has lead to the best results in most of the test problems: (i) restrict neighborhood built through a dynamic enumeration technique determining only one single position for the moving task in the task list of each target processor, (ii) tabu list organized as a 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) an aspiration criterion establishing that a certain tabu move drops its tabu classification if it takes the current solution $s$ to a neighbor solution which improves the best solution $s^*$ found so far. Those are also the strategies and parameter values which will be used in the parallel implementations described in the present work.

# 4 Accelerating the Makespan Computation

As seen in Section 2, the computation of the makespan of a parallel application is performed in $O(n^2)$ time complexity, where $n$ is the number of tasks. The labeling technique incorporated in algorithm **makespan** is, in fact, a deterministic simulation of the execution of the parallel application, where the execution time of each task $t_k$ in processor $p_j$ is assumed to be constant and equal to its average estimated value $\mu(t_k, p_j)$ $\forall t_k \in T$, $\forall p_j \in P$. This deterministic simulation is always started with all tasks in the *non-executable* state and variable *clock* set to zero. At each iteration, one must calculate the makespan of each neighbor solution $\bar{s}$ obtained from $s$ by the move $(\mathcal{A}_s(t_i), t_i, \mathcal{A}_{\bar{s}}(t_i))$. The deterministic simulation must be recalculated for the new solution $\bar{s}$, from the time instant given by $clock = 0$.

However, due to the partial execution order between tasks defined by the task precedence graph, some of them will not have their execution start time changed when going from $s$ to $\bar{s}$. There is a time instant $clock = T_0$ up to which the parallel application is executed identically, either under the effect of the scheduling given by $s$, or by that defined by $\bar{s}$. Therefore, it is desirable to use the computation of $c(s)$ to accelerate the computation of

**algorithm RN-STM-TS**
**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 RN-STM-TS**

Figure 2: Tabu search algorithm **RN-STM-TS** for the task scheduling problem

$c(\bar{s})$. One must determine the time instant $clock = T_0$ of the deterministic simulation from which it is possible to occur changes in the task execution order of $\bar{s}$ relatively to $s$, repeating in $\bar{s}$ the scheduling of tasks which start their execution before $T_0$.

Let $t_i \in T$ be the task involved in the move from $s$ to $\bar{s}$. If $t_i$ does not have any predecessors, then it is *executable* since the beginning of the simulation ($clock = 0$) and $T_0 = 0$. If this is not the case, let $pred(t_i)$ be the set of predecessors of tasks $t_i$ and $end(t_i)$ its termination time. Moreover, let $t_f \in pred(t_i)$ be such that $end(t_f) = max_{t_j \in pred(t_i)}\{end(t_j)\}$. Thus, $t_f$ is the last predecessor of $t_i$ to be terminated and $end(t_f)$ is the time instant when $t_f$ becomes *executed*. The results of the simulation already performed during the computation of $c(s)$ are not affected by any move of task $t_i$ from the source processor $\mathcal{A}_s(t_i)$ at any time instant before $end(t_f)$, regardless of the target processor where it is placed, as far as it cannot switch to *executable*. However, after the time instant $end(t_f)$, the task $t_i$ is already *executable*. If it is allocated to a different processor, i.e. $\mathcal{A}_{\bar{s}}(t_i) \neq \mathcal{A}_s(t_i)$, it may alter the execution order

```
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**

of those tasks which are not yet *executing*. Consequently, $T_0 = max_{t_j \in pred(t_i)}\{end(t_j)\}$ is the first time instant from which the schedule associated to $\bar{s}$ may differ from that associated to $s$, so that it must be re-evaluated from this time instant.

Thus, the optimization of the computation of $c(\bar{s})$ for a certain solution $\bar{s} \in N(s)$ may profit from the previously computed values obtained during the calculation of $c(s)$. In the context of the tabu search implementation, the value of $T_0$ is evaluated according to the task $t_i$ which defines the above mentioned move in the moment when it takes $s$ to $\bar{s}$. This value $T_0$ is given to algorithm **makespan** to evaluate $c(\bar{s})$, which is initialized with this time instant. The starting times of the tasks which start their execution before $T_0$ are not recalculated, being identical to those in $s$.

Table 1 shows the reduction in the computation times of the tabu search algorithm, achieved with the optimization of the computation of $c(\cdot)$ for applications with MVA topology of different sizes (different values of $n$) [28]. The processing times are presented in seconds and were measured while executing the tabu search algorithm in a SPARC 4 workstation. The percentual reduction is given by $(\tau_{original} - \tau_{optimized})/\tau_{original}$, where $\tau_{original}$ is the computation time of the tabu search algorithm with the complete procedure for makespan calculation each time a certain scheduling must be evaluated, and $\tau_{optimized}$ is the computation time of the tabu search algorithm for the optimized makespan calculation as described above.

| $n$ | $\tau_{original}$ (s) | $\tau_{optimized}$ (s) | Reduction (%) |
|---|---|---|---|
| 16 | 2.1 | 1.7 | 20.1 |
| 36 | 32.4 | 23.6 | 27.2 |
| 64 | 260.1 | 183.4 | 29.5 |
| 100 | 4428.2 | 2979.9 | 32.7 |
| 144 | 4285.8 | 2760.4 | 35.6 |
| 196 | 17739.9 | 11116.3 | 37.3 |
| 256 | 21488.8 | 12904.0 | 40.0 |
| 324 | 54169.3 | 32683.0 | 39.7 |

Table 1: Percentual reductions of the computational times due to the optimized calculation of $c(\cdot)$

The optimized computation achieved reductions in the range of 20 up to 40% on the computation times for the algorithm **RN-STM-TS**, which grows progressively larger with the increase of problem sizes. This result is significant, and turns possible the solution of problems with greater sizes. However, one should have in mind that this optimized algorithm still has $O(n^2)$ time complexity, which is not changed with this optimization technique.

# 5 Parallelization of Tabu Search

Obtaining good solutions with a guided iterative local search method is often charged by high computational times, due to a high number of iterations or to intensive computation iterations (and therefore of long duration each). If the search is already optimized on a single processor computer, one could make use of a multiprocessor system in order to accelerate it. To increase the debit of the number of iterations by time unit, we must (i) accelerate the calculations within each iteration, or (ii) execute several moves simultaneously. A third parallelization technique, which actually does not belong to any of the two approaches described above, consists of executing several independent searches.

The first trend supposes the parallelization of the evaluation of the objective function or move values, or even the choice of the best move. The second leads to a problem decomposition or duplication: if several moves may be performed simultaneously, then they are independent and the problem is susceptible of being decomposed [34]. Still considering the first approach, we may envision a second type of parallelization where move evaluation is a costly sequential procedure, but it is possible to execute concurrently the election of the best neighbor. This is the approach used in this work.

According to Crainic *et al.* [7], a great number of tabu search procedures may be derived using different strategies to implement the small-, medium- and long-term memories. This tendency is emphasized when parallel implementations are contemplated. Parallel architectures allow the 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 for the parallelization of tabu search. A tabu search scheme to the Quadratic Assignment Problem (QAP) was presented in [33]. The author suggests that the most efficient way of executing tabu search concurrently is based on distributing across many processors the most computationally intensive phases of the algorithm. For the QAP, this phase consists of evaluating the neighborhood of the current solution. The neighborhood is divided into parts of about the same size, which are evaluated on different processors. Each processor computes the values of the moves assigned to it and communicates to the others the best move it found, receiving the best moves found by the other processors. Each processor chooses and performs the best move proposed to (or by) it and updates the cost function and move value accordingly. Computational results on a ring of transputers are presented. Efficiency values of up to 85% are reported for 10 processors. Another parallelization strategy proposed in the same work consists of performing many independent searches from different initial solutions.

Chakrapani and Skorin-Kapov [4] also developed a heuristic scheme to perform tabu search for the QAP. Dynamically changing tabu list sizes, aspiration criteria and long term memory are some of the main features of this algorithm. The proposed algorithm is divided into four phases and requires a massively parallel machine: initialization phase, preliminary search phase, intensification phase and diversification. The machine used is the Connection Machine CM-2, which is a fine grain SIMD dynamically reconfigurable computing system. In addition to computations, the CM-2 system can perform interprocessor communications in parallel. In this implementation, the CM-2 processors are organized as a two dimensional grid. All processors in row $p$ store a parallel array corresponding to row $p$ of the distance matrix. Similarly, all processors in column $q$ store the corresponding column $q$ of the flow matrix. The required information to evaluate a pairwise exchange is computed in parts by four different processors. Since the benefits of a massively parallel implementation increase with the dimension of the problem, only large QAP problems were tested. The average time per iteration ratio is compared with three other tabu search algorithms from the literature.

The same authors formulated as a QAP the problem of mapping tasks to a set $P$ of processors in a multiprocessor system in order to minimize communication time, implementing a tabu search algorithm on the CM-2 [5]. This heuristic was based on greedily selecting a pair of tasks and swapping the processors to which they are mapped to. Two levels of parallelism are employed in the parallel implementation. First, the candidate tasks to be swapped are identified in parallel. Second, more than one pair of tasks are swapped in a single iteration. The scope of pairwise exchanges is restricted in order to reduce the neighborhood size: two processors can exchange their tasks only if their addresses differ by exactly one bit (in the CM-2, these two processors are adjacent in the hypercube, i.e. they are only one link apart). Move values are evaluated by an approximation and multiple moves may be performed at the same iteration in a parallel implementation. The authors also designed a diversification strategy which performs many moves exploiting parallelism to a greater extent.

Rassai [30] also considered the parallelization of tabu search on the CM-2 for the solution of the QAP. Three algorithms are proposed, implemented and evaluated: a simple classical scheme (sequential and parallel versions), an enhanced parallel strategy including aspiration criteria and diversification features, and a second enhanced parallel strategy using two sequencers of the CM-2. The parallel version of the simple sequential tabu search is based on the parallelization of the neighborhood search. Each processor evaluates a single move, and the best move results from the comparison of all move values. The sequential algorithm is executed on a VAX front-end. The parallel version uses this same front-end for the sequential instructions, while the parallel instruction are sent to the CM-2 nodes. The enhanced parallel tabu search algorithm includes an aspiration function and a diversification phase. The final parallel approach is, in fact, a concurrent execution of both the classical and the enhanced parallel algorithms previously described. After obtaining the new frequency based diversified initial solution, both programs (executing on two different CM sequencers) resume their execution. Both programs communicate with each other during execution.

August and Mautor [2] also studied the parallelization of tabu search to the solution of the QAP on the CM-2 with 16K processors and 2 sequencers. The massively parallel version is directly inspired on the sequential classic version. The decomposition is done in such a way that each of the $N^2$ processors owns a pair location/unit. The quadratic costs are obtained by each processor at each move evaluation through an elementary calculation. The results are slightly less good than those obtained by Taillard [33] and Chakrapani and Skorin-Kapov [4].

Garcia and Toulose [11] proposed a tabu search approach for the vehicle routing problem with time windows. A synchronous parallel algorithm is presented and an implementation on two distributed architectures is evaluated. An asynchronous version was also elaborated in order to solve the problems encountered with the synchronous version, which is based on the partition of the neighborhood among several processors. The processors are organized in a master-slave scheme, although the communication is performed using a tree based interconnection structure. The master is responsible for choosing the best move at the end of each iteration. The acceleration due to parallelization makes possible the exploration of a wider neighborhood. In the asynchronous version interprocessor communication occurs exclusively in two situations: (i) when a processor finds a new optimum and wants to turn this public to other processors, and (ii) if some processor has not improved its solution after a certain number of iterations, then it will look for help from its peers, demanding their current optimal solutions. The best solution found is never worse than that found by the sequential version, as far as all processors execute the sequential version independently, exchanging optimal results from time to time. The difference relies in the way each processor does its own sequential search. Features such as initial solution and neighborhood size are varied to differentiate the searches among processors.

Fiechter [10] proposed an efficient tabu search algorithm for large traveling salesman problems on MIMD parallel computers. The general idea is that the local search can be sliced into several independent searches performed in parallel, without much loss of quality, while tabu search assures a high global solution quality. Two general classes are considered according to the method used for parallelizing moves. First, the search for the next move to be performed can be parallelized, requiring the partition of the set of feasible moves. The overall best move is then determined and applied to the current solution. This technique requires extensive communication, since synchronization is required at each step. It is therefore only worth applying to problems in which the search for the best move is relatively complex and time consuming. The second type of parallelization consists in peforming several moves concurrently, what can be done by partitioning the problem itself into several independent subproblems. The global solution is then obtained by combining the subsolutions. This method needs no communication or synchronization, except for its initialization and for grouping the subsolutions at the end. One can therefore expect a real gain in efficiency by the parallel algorithm even when moves are simple. The difficulty with this kind of parallelism is that it strongly limits the move possibilities and thus generally induces a loss of quality. High-level tabu search seems particularly well suited to overcome this difficulty, as this type of parallelism can be used in the intensification phase, ensuring the global quality of the final solution by the diversification procedure. The intensification strategy consists in dividing the current tour in several open subtours (*slices*), each of them having a vertex in common with the two adjacent slices. The path between these two fixed vertices is then optimized independently on each slice. Two passes of this scheme with shifted slices are applied to allow other combinations of edges, particularly around the boundaries between slices. The entire procedure is clearly suited for parallel computation, as the optimization is done completely independently on each slice. The algorithm has been implemented in OCCAM on a network of transputers. The parallel algorithm has been tested for 500, 3000, and 10000 vertices, and the speedups have been observed to be close to optimal.

Crainic *et al.* [8, 9] presented an appraisal of synchronous and asynchronous parallel strategies for tabu search, applied to the multicommodity location-allocation problem with balancing requirements. They identify the most promising parallelization approaches and evaluate the impact of some parameters on performance and solution quality, namely the length of the synchronization steps, the number of processors and the manipulation of the exchanged information between processors.

In [35], Taillard presents a special method based on tabu search for the job shop scheduling problem, considering both serial and parallel implementations. The goal is to find a schedule for the operations in the machines, considering the precedence restrictions, which minimizes the overall makespan, i.e. the finish time of the last operation to be completed in the schedule. The sequential case shows that tabu search is more efficient than other methods previously proposed in the literature, such as simulated annealing and the shifting bottleneck procedure. The addition of long-term memory has shown itself easy to implement and very efficient in enhancing solution quality for larger problems. For small problems, tabu search is slower than the best known branch-and-bound methods. However, when problem size increases, the efficacy of tabu search is greater than any other exact or heuristic method already published. The parallelization is done over the computation of the longest path, since this is the most intensive computational part of the algorithm. This parallel method was implemented on a transputer based machine. However, the lower performance results are due to the interconnection topology, which does not adapt itself to the completely connected communication topology between processes, producing large message transmission delays relatively to the processing time. For the general case of sequencing problems, like "flow shop", this processor interconnection configuration is ideal, since the information exchange occurs exclusively between neighbors. In this case, the speedup results are better. The communication delays are still significant, but as the processing times increase, this implementation becomes more and more interesting. A second approach with fine grain parallelism was proposed through the parallelization of the shortest path computations. In this case, common memory structures are nece ssary, for the simultaneous access to shared variables. This parallelization scheme was implemented on a Cray-2 with two processors. However, due to the inefficiency of this machine in executing parallel applications which require frequent synchronization, it was observed that the time necessary to synchronization is larger than that required by the computations, leading to low performance results. A third approach, based on a generic method of parallelizing probabilistic algorithms, was considered and theorically evaluated through several consecutive executions of the sequential tabu search algorithm. It was shown that parallelization based on the execution of multiple independent search trajectories is capable of producing almost linear speedup. The tabu search approach finds new and better solutions for every problem in two sets of benchmark problems.

The problem of mapping a set of tasks composing a parallel algorithm into a set of identical processors is considered in [22]. Generally, the mapping problem has the goal of minimizing the overall execution time of the parallel application, whose tasks communicate and present temporal dependencies between each other. One must assure that the time gain in the parallelization is not lost in excessive communication between tasks allocated to different processors. An adaptation of tabu search, providing a sequential algorithm, is described, which is then parallelized using PVM. The asynchronous parallelization strategies are based in the work of Crainic *et al.* [7]. The processors perform partial independent searches, exchanging information on certain synchronism points, which alternate diversification and intensification phases. This kind of strategy allows the processors to obtain knowledge of the global state of the search. This algorithm was implemented and evaluated on a heterogenous SPARC workstation network e on a parallel 32 node IBM/SP1 machine, achieving good solution quality and speedup results.

As mentioned by Fiechter [10], synchronized parallelization schemes generally require extensive communication. Therefore, they are only worth applying to problems in which the computations performed at each iteration are complex and time consuming. This is exactly the case of our scheduling problem, in which the search for the best move at each iteration is a time consuming task. The parallelization of the tabu search for this scheduling problem is justified because it displays both combinatorial nature and computationally intensive requirements. Due to the presence of the precedence constraints, the calculation of the cost of each solution implies in running a full computational procedure for task scheduling and makespan evaluation, similar to a deterministic simulation of the execution of the parallel application. Although the scheme proposed in Section 4 allows some reductions in computational times, the time complexity associated with the computation of the makespan of each neighbor solution remains $O(n^2)$. Thus, from this point of view, the sequential tabu search algorithm for this problem is already a good candidate for parallelization. Moreover, the size of the problems amenable to be solved in reasonable computational time by sequential tabu 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.

# 6 A Taxonomy of Parallel Strategies for Tabu Search

The design of parallel implementations for tabu search may use some basic ideas derived from the work of Crainic *et al.* [7], whose main interest is to take into account the differences in control and communication strategies which are so important when designing parallel algorithms. The proposed taxonomy, which is here briefly described, has a twofold basis: first, on how the search space is partitioned; and second, on the control and communication strategies used by parallel tabu search procedures.

Since exploration of the solution space and knowledge acquisition is one of the building blocks of tabu search, the strategies used for its parallelization must constitute an important taxonomy criterion. Furthermore, any parallelization implies some decomposition either of the domain, or of the basic steps and tasks of the algorithm, or of both. Consequently, not all information is necessarily available at all times. Therefore, how the knowledge is gathered during the parallel exploration of the domain, and the way it is exchanged and combined among processes are issues as important as how the domain is divided among, or how the tasks are allocated to, the various processes.

The taxonomy is built according to three dimensions meant to capture all these factors. The first two dimensions represent the parallelization schemes relatively to the control of the search trajectory and the communication approach, while the third accounts for the strategies used to partition the domain and to specify the parameters for each search. They are: *(i)* search control cardinality (1-control and $p$-control); *(ii)* control type (rigid synchronization, knowledge synchronization, collegial, and knowledge collegial) and *(iii)* search differentiation (SPSS, SPDS, MPSS, and MPDS).

## 6.1 Search Control Cardinality

Control of the parallel search may either stay with one processor, usually called the master or main processor, or be distributed among several processors. Two categories may thus be defined:

• **1-control:** This category trivially corresponds to the sequential case. In the parallel context, it represents the approach where one processor essentially executes the algorithm, but delegates some of its work to other processors. The master is responsible for the algorithm execution itself: it collects and reconciliates the information, distributes the tasks to be executed by the other processors, and determines when the search has to stop. The tasks that are delegated may consist of time-consuming numerical computations, corresponding to the so-called low level parallelism in a branch-and-bound context. It may also consist of the parallel exploration of the neighborhood or the construction and evaluation of the candidate list.

• **$p$-control:** This case corresponds to that one where the search is shared among $p > 1$ processors. The classical collegial arrangement, where each processor is in charge of its own search, as well as of establishing communications with other processors, belongs to this category. The global search terminates once each individual search stops. Coordination of information exchanges and attempts to ensure that the adequate information is available when required are among the main issues in this context. They also play an important role in defining the type of control that is exerted.

## 6.2 Control Type

The second dimension of the taxonomy is based on the type and flexibility of the control. It takes into account the communication organization, synchronization and hierarchy, as well as the way information is processed and shared among processors. The control type dimension is made up of four stages or degrees, that combine with the two levels of control cardinality defining the parallelization strategy relatively to process and information handling:

• **Rigid synchronization:** This strategy ideally complements the 1-control approach. It represents the classical master-slave case, where the master coordinates the sequential execution of the search, using other processors which only perform computation intensive tasks. Communication is exclusively performed between the master and each of the slave processors. The information is kept and handled exclusively by the master. This kind of synchronization with a $p$-control cardinality is the straightforward parallelization strategy where independent searches are performed simultaneously: each search may start from a different initial solution, or may use a different set of parameters, or both. Again, there is no communication among processors during the search, and each search terminates when its own stopping criteria is met. The best solution is selected once all processors have stopped. It is interesting to notice that with the same control type, but with different cardinality, we obtain the two extremes of the parallel work: rigid collaboration work under a central device control, and a pool of processors which work in a totally independent manner.

• **Knowledge synchronization:** This scheme is also characterized by a synchronous operating mode, but an increased level of communication permits to build and exchange knowledge. When using the 1-control framework, the master continues to be the keeper of the information, to synchronize the processes, and to dispatch work to the slaves, but it delegates a larger part of the work. The slave processors still do not communicate among themselves. Their tasks, however, are more complex than in the rigid synchronization case, and may need local memory structures. An example would be a slave processor executing a limited sequence of tabu search steps on a given subset of the neighborhood (intensification on promising candidates). The master synchronizes as well, in order to compute the results from the slaves, and then dispatches new tasks to them. When using the $p$-control strategy, the knowledge synchronization mode corresponds to several independent search trajectories through the domain, similarly to the rigid synchronization case, with the difference that each control processor stops at a predetermined iteration, the same for all processors. At that moment, an intense communication phase begins among all processors. It may be considered as a hybrid approach between rigid synchronization and independent collegial.

Summarizing, in a synchronous mode, the 1-control strategy implies vertical master-slave communication channels exclusively. On the other hand, only horizontal processor to processor communications exist in a $p$-control strategy, where no processor plays the role of the master and they collaborate in a collegial communication scheme. The difference between rigid and knowledge synchronization is not always clear in the 1-control context, since it is mostly based on how much work the master assigns to each slave. This difference is much more significant for $p$-control strategies, since it corresponds to the absence or to the presence of interprocess communication and knowledge exchanges.

The third and fourth degrees of the control strategies make use of asynchronous communication modes. The differences are relative to the quantity, quality, and treatment of the exchanged information. Both strategies make sense in the $p$-control context only. When considering asynchronous versions of the master-slave strategies, one rapidly arrives at some variant of the $p$-control approach:

• **Collegial:** In that case, each processor searches all or part of the domain, eventually using a different implementation strategy of tabu search. When a processor finds an improving solution (locally or globally, according to the chosen strategy), it broadcasts this solution (together eventually with its context and history) to all or to some (the neighboring ones) of the other processors. It may also store this solution in a central memory, and only broadcasts (if at all) that a better solution has been found. In all cases, however, communications are simple, in the sense that each message sent corresponds to each message received.

• **Knowledge collegial:** Contents of communication are analysed to infer additional information concerning the global search trajectory and the global characteristics of good solutions. Global memories (containing the frequency of change of some global variables) that reflect the dynamics of the asynchronous parallel exploration of the domain may thus be built, and the information may be returned to the individual processors. Therefore, the message received by a processor is generally richer than, and not identical to, the one initially sent by another processor.

## 6.3 Search Differentiation Strategy

This dimension is quite similar to the one considered by Vo$\beta$ [36]. It deals with the number of different starting solutions, and with the number of different solution strategies. The naming refers directly to the decision to start the exploration of the domain from the same or from different points, and the use of either a unique or different search strategies for each search thread.

• **SPSS** (Single Point Single Strategy): this is the most simple case, generally allowing only low level parallelism. It is exclusively associated with synchronous 1-control parallelization schemes.

• **SPDS** (Single Point Different Strategies): this case refers to the situation where each processor runs a different tabu search strategy, but they all start with the same initial solution.

• **MPSS** (Multiple Points Single Strategy): this strategy stands for the case where each processor starts from a different solution of the domain, but use the same tabu search settings and rules to explore the domain.

• **MPDS** (Multiple Points Different Strategies): this final situation represents the most general class and has all others as special cases.

# 7 Parallel Strategies for the Task Scheduling Problem

In this section, we describe several tabu search parallelization strategies for the **RN-STM-TS** algorithm presented in Section 3 for the task scheduling problem under precedence constraints. As it was mentioned before, this scheduling problem involves the computation of a cost function through an algorithm for calculating the makespan of a parallel program. This makespan routine must be performed each time a new solution (obtained from the current solution by the application of a move) has to be evaluated. The main goal of the parallelization is to allow the solution of longer problems, as well as to reduce the computation times of the tabu search algorithm. In this sense, parallelization does not provide a new search strategy with respect to the sequential version [28], since the parallel implementations lead to identical search paths.

The parallelization strategies 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 programming 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. Five 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 call them in the following by *problem-tasks*. The tasks which effectively compose the parallel implementation of the tabu search algorithm, and which are distributed to the different processors, are called differently depending on their role in each strategy, namely by slave-task, master-task, parent-task, child-task or simply tasks.

## 7.1 Master-Slave Model

The classic master-slave model may be classified as a 1-control, rigid synchronization SPSS strategy. For each current solution $s$, its neighborhood is partitioned into subsets of neighbor solutions, which are evaluated by different slave-tasks. The most balanced way of dividing the neighborhood is to assign an equal number of neighbors (or moves) to each slave-task. The latter will then strictly consider neighbor solutions obtained through moves involving some pre-determined problem-tasks. The search for the best neighbor performed by each slave over a partition (or subset) of the neighborhood is then called a best neighbor partial search. Moreover, as it 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 best neighbor partial search, instead of possibly being idle while slave-tasks are executing their search procedures. The number and size of partitions still offer different approaches to this strategy model:

• **Single Partitioning (MS-SP):** The neighborhood is partitioned only once in approximately equal size partitions (subsets), 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 with the initial solution, as far as the slaves do not need to keep track of the best solution. 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 no non-tabu moves have 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.

• **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 in 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 to this slave a new partition. When all partitions have been searched, the master has the value of the best neighbor. It then proceeds as in the former case. In this way, more work is given to the less loaded slave-tasks. 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.

• **Multiple Variable Size Partitioning:** In the former case, load balancing was performed only in accordance to the work demand of slave-tasks. Here, we attempt to reach a more accurate load balancing. Initialization is just the same as with multiple (fixed size) partitioning. At each iteration, when the master receives results from a slave and the domain still has not been fully searched, it will send to this slave another partition so that it can continue on searching. However, the size of the partition which is sent depends on the time taken by this slave to finish the previous best neighbor partial search. If a slave takes more time to finish its work than other slaves, then it will receive a smaller partition to be searched. This approach is based on the fact that if a slave has taken more time to finish the work given before, with great probability it has less available computational power. The master task works in the same way as before, but it must also keep track of the order and time in which slave-tasks are finishing their work, so that it can determine new partition sizes accordingly.

## 7.2 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 strategy can be considered according to two different approaches, namely single token and multiple token, due to their resemblence to the equally named access level of local network protocols:

• **Single Token (SPMD-ST):** Initialization is done similarly to 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 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 to 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 a message from its predecessor, 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, as subsequently do the other tasks, 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 each 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 possibly 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 up to $M$ messages each time.

# 8 Software and Hardware Environments

In this section, we present the basic framework used for the implementation of the parallel strategies, in terms of software and hardware characteristics. 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. 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). More details about the harware and software environments are presented in the subsequent sections.

## 8.1 PVM 3: Technical and Performance Considerations

PVM stands for *Parallel Virtual Machine*. It is a software package developped by Oak Ridge National Laboratory, that allows a heterogeneous network of parallel and serial computers running Unix to appear as a single concurrent computational resource. It is composed of two main parts: a daemon process (`pvmd`) and a user library (`libpvm`) that contains routines for initiating processes on other machines, for communication between processes and for changing the configuration of machines. PVM is used in several institutions and is distributed freely, reinforcing portability as one of its main advantages.

Under PVM, a user defined collection of serial, parallel, and vector computers appears as one large distributed-memory computer. In other words, PVM allows application tasks to exploit the architecture best suited for their solution. PVM handles all data conversion that may be required if two computers use different integer floating point representations. It also allows the virtual machine to be interconnected by a variety of different networks. A task is defined as a unit of computations in PVM, analogous to a Unix process. Each computer in the network is called a *host*. Applications written in C or in Fortran 77 can be parallelized by using message passing constructs, common to most distributed-memory computers. PVM supports heterogeneity at the application, machine, and network level. Some interesting features which clearly define its functionality and main advantages are commented below. Some examples of machines which support PVM are: Alliant, Butterfly, CM-2 and CM-5, Convex, Cray, iPSC/860, KSR1, Paragon, IBM RS 6000, SUN systems, and IBM SP1.

The task integer identifier (`tid`) is the primary and most efficient method of identifying processes in PVM. Since `tid`'s must be uniquely defined across the entire virtual machine, they are supplied by the local `pvmd` and are not user chosen. There are several routines that return `tid` values so that the user application can identify other processes in the system. In this context, groups of processes may be defined by the user and processes will be named according to a reference to their respective groups.

PVM supplies routines that enable a user process to become a PVM task and to become a normal process again, add and delete hosts, start and terminate tasks, send signals between tasks, find information about configuration and active tasks. PVM has routines for packing and sending messages between tasks. The model assumes that any task can send messages to any other task and that there is no limit to the size or number of such messages. Although the hosts have physical memory limitations, the communication model does not restrict itself to the limitations of a particular machine. Message buffers are allocated dynamically, so that the maximum size of messages that can be sent or received is limited only by the amount of available memory on a given host. There are several point-to-point communication functions available: asynchronous blocking send, asynchronous blocking receive, and non-blocking receive functions. PVM also supports multicast to a set of tasks and broadcast to user defined groups of tasks. On the other hand, due to its main characteristic of portability, it does not profit from the facilities of the architecture to perform communications. In this sense, considering the transmission through the interconnection network, broadcast and multicast primitives are performed as a sequence of point-to-point communications. PVM supports wild cards in its receive functions and provides routines for capturing information about received messages. The PVM model guarantees that message order is preserved.

Application programs view PVM as a general and flexible parallel computing resource that supports a message-passing model of computation. This resource may be accessed at three different levels: (i) the *transparent* mode, in which tasks are automatically executed on the most appropriate hosts (generally the least loaded computer), (ii) the *architecture-dependent* mode, in which the user may indicate specific architectures on which particular tasks are to execute, and (iii) the *low-level* mode, in which a particular host may be specified. Such layering permits flexibility, while retaining the ability to exploit particular strengths of individual machines on the network. Generally, the scheduler simply does a round-robin assignment, i.e., it distributes tasks uniformly through the hosts that compose the parallel virtual machine.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the active processes may have arbitrary relationships between each other. In addition, any process may communicate and/or synchronize with any other one. This allows the most general form of MIMD parallel computation, but in practice most concurrent applications are more regularly structured. Two typical structures are the SPMD model, in which all processes are identical, and the master-slave model, in which slave processes perform work for one or more master processes. These are exactly the two models on which the parallel strategies developed in this paper are based (see Section 7). There are no limitations to the programming paradigm used with PVM. Any specific control or dependency structure may be implemented under PVM by appropriate use of its constructs. However, the user should be aware of the performance considerations which apply to any message-passing parallel architecture, such as: task granularity, interprocessor communication cost, multiuser and multitasking effects, system heterogeneity and load balancing.

Multiuser and multitask environments specially affect program performance. One of the most important considerations is large message latencies through the network. It can be caused by the distance between machines, in the case of long distance networks, or due to contention in local networks. If the application is designed so it sends messages exclusively to its neighbor tasks, then one may assume that there is no contention. This would be the case of a distributed memory multiprocessor, where message transmissions can be performed in parallel. However, Ethernet networks may be considered as a single bus through which one single message will be sent at a time. In this sense, transmissions through Ethernet networks, such as token-ring, FDDI and HiPPI have properties that may cause variable latency. Besides, processing performance and the network effective bandwidth change dynamically, since many users share common resources. Consequently, the application may achieve good performance during one execution and l ower performance during another.

## 8.2 The IBM SP1

The IBM SP1 [3] is the first parallel machine with distributed memory commercialized by IBM. Its first installations took place in 1993. The configurations go from 8 to 64 processors. The configuration used as the platform for the implementation of the algorithms described in this work goes up to 32 nodes, although only 16 were available for running PVM-based parallel applications. The basic processor is the RS 6000 (Risc System 6000). Each processor can dispose of 256 Mbytes of memory and 2 Gbytes of disk space. The peak performance is said to be of 8 Gflops with a maximal configuration of 64 processors.

The processing nodes are connected by a multiple stage interconnection network. The basic component of this network is an $8 \times 8$ communication circuit, which permits to store blocked packets and contains fault transmission detection devices. These circuits are connected through a multi-stage network with various paths between each pair of nodes, which improves the performance and fault tolerance due to alternative paths for interprocessor communication, although it still is a blocking interconnection network. The maximum node bandwidth is 40Mbytes/s and the inducted latency is of 500 ns. The network bandwidth is 640 Mbytes/s. The size of the network can be increased in order to contain up to many thousands of nodes. In this case, the bandwidth increases linearly with the number of nodes and the latency is maintained below $1\mu$s.

The programming model is based on message passing. A communication library is offered to the user, with point-to-point and group communication procedures. The latter can be used over dynamically defined process groups. Each node disposes of a complete Unix system (AIX 6000). Consequently, any program developed for a RS 6000 workstation can be executed on a single SP1 node. All nodes offer the same user interface environment (such as password, file system, printing, and storage). Scheduling and load balancing schemes can be used during the batch execution of sequential and parallel applications. In this case, the user does not know the identification of the processes which compose his parallel application.

The IBM AIX parallel environment allows the development and design of programs for the SP1 as for the RS 6000 workstation network, by offering many tools such as a parallel debugger, a parallel profiler, and a trace visualization program linked with monitoring tools. Other programming environments, such as Linda and PVM, are also available. The FORGE tool, commercialized by Applied Parallel Research, allows the conversion of Fortran programs into parallel code, as well as programming with the High Performance Fortran.

# 9 Computational Experiments

In this section, we describe first the framework for the computational experiments presented further. Three aspects are discussed, namely the test problems, the configuration pattern of the basic tabu search algorithm, and the description of the implemented parallel strategies. Following, computational results are reported.

## 9.1 Test Problems

The characteristics which fully describe the scheduling problem are the same as those in Porto and Ribeiro [28]. 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 e.g. deviations in task execution times due to interrupts and contention for shared memory, they make possible the static assignment of tasks to processors [29]. 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 [23] measures the heterogeneity of the architecture, given by the ratio between the execution time of any instruction in the homogeneous processor and its execution time in the fastest processor.

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 [31, 37]. Figure 4 depicts an example of a task graph associated with the MVA algorithm for $n = 25$ tasks. There are $n_h$ tasks in the horizontal central axis of the graph. For this same topology pattern, we define 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 taken equal to one, while that of the inner tasks equals two [38]. 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. $m = n_h/2$. The characteristics of the nine test applications are given in Table 2.

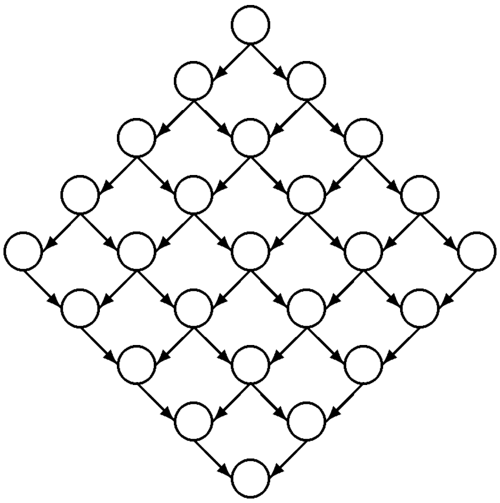

Figure 4: Task graph for an application of the MVA algorithm with $n = 25$ tasks

| Application | Number $n_h$ of tasks in the horizontal axis | Number $n$ of tasks | Serial fraction |
|---|---|---|---|
| P-01 | 4 | 16 | 0.417 |
| P-02 | 6 | 36 | 0.25 |
| P-03 | 8 | 64 | 0.178 |
| P-04 | 10 | 100 | 0.139 |
| P-05 | 12 | 144 | 0.114 |
| P-06 | 14 | 196 | 0.096 |
| P-07 | 16 | 256 | 0.083 |
| P-08 | 18 | 324 | 0.073 |
| P-09 | 20 | 400 | 0.066 |

Table 2: Characteristics of the test problems used in the experiments

## 9.2 Tabu Configuration Pattern

The **RN-STM-TS** 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 aspiration criteria and on the restrictiveness of the tabu list.

Several experiments were made in [28] to obtain the best tabu configuration pattern, which would provide the best performance for the **RN-STM-TS** 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) a restricted neighborhood built through a dynamic enumeration technique determining only one single position for the moving task in the task list of each target processor, (ii) tabu list organized a 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) an aspiration criterion establishing that a certain tabu move drops its tabu classification if it takes the current solution $s$ to a neighbor solution which improves the best solution $s^*$ found so far.

## 9.3 Parallel Strategies

Except for the Master-Slave with Multiple Variable Size Partitioning strategy, all others strategies presented in Section 7 were implemented and analysed. 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 a number of processors ranging from 4 to 16 ($4 \leq q \leq 16$).

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.

## 9.4 Numerical Results

Let $\mathcal{S}(q)$ be the *speedup* achieved by the parallel algorithm using $q$ identical processors, relatively to the processing time of the sequential algorithm for the same problem when executed on a single processor:

$$\mathcal{S}(q) = \frac{\tau_{seq}}{\tau_{par}(q)},$$

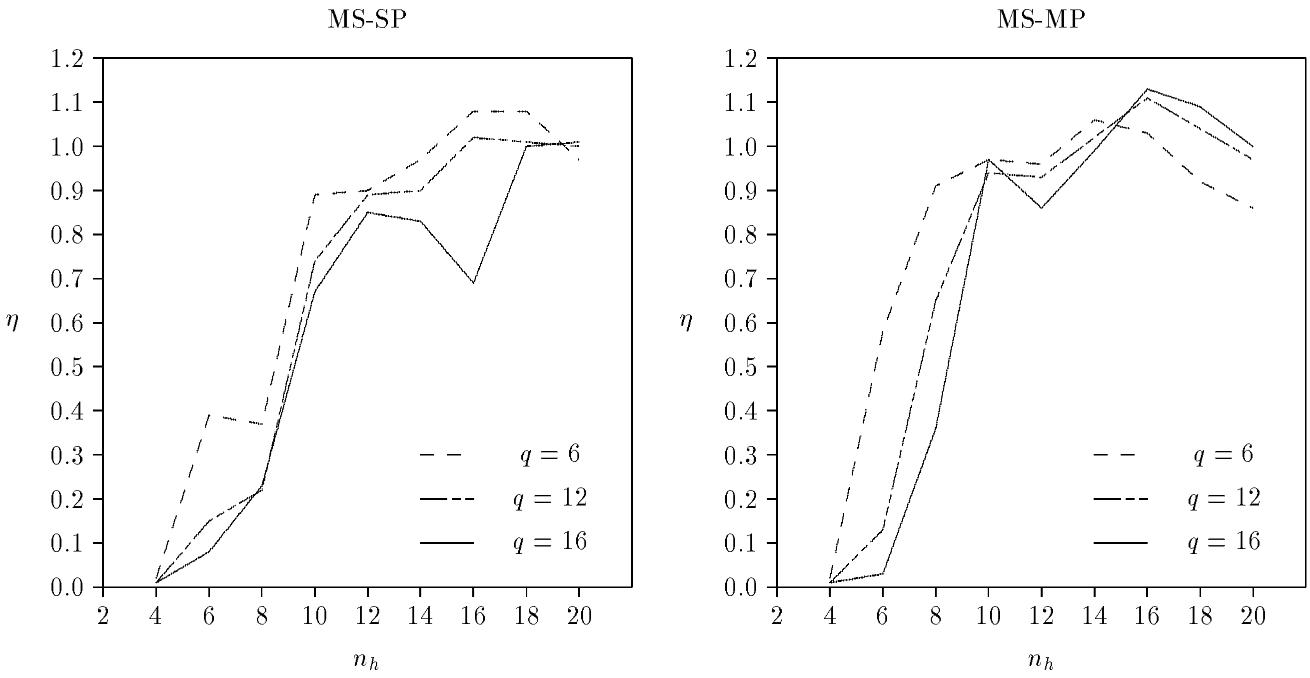


Figure 5: Efficiency ($\eta$) vs. problem size ($n_h$): MS-SP and MS-MP strategies

where $\tau_{seq}$ and $\tau_{par}(q)$ are, respectively, the sequential and parallel elapsed times observed for the sequential algorithm and some parallel strategy for the tabu search algorithm. The

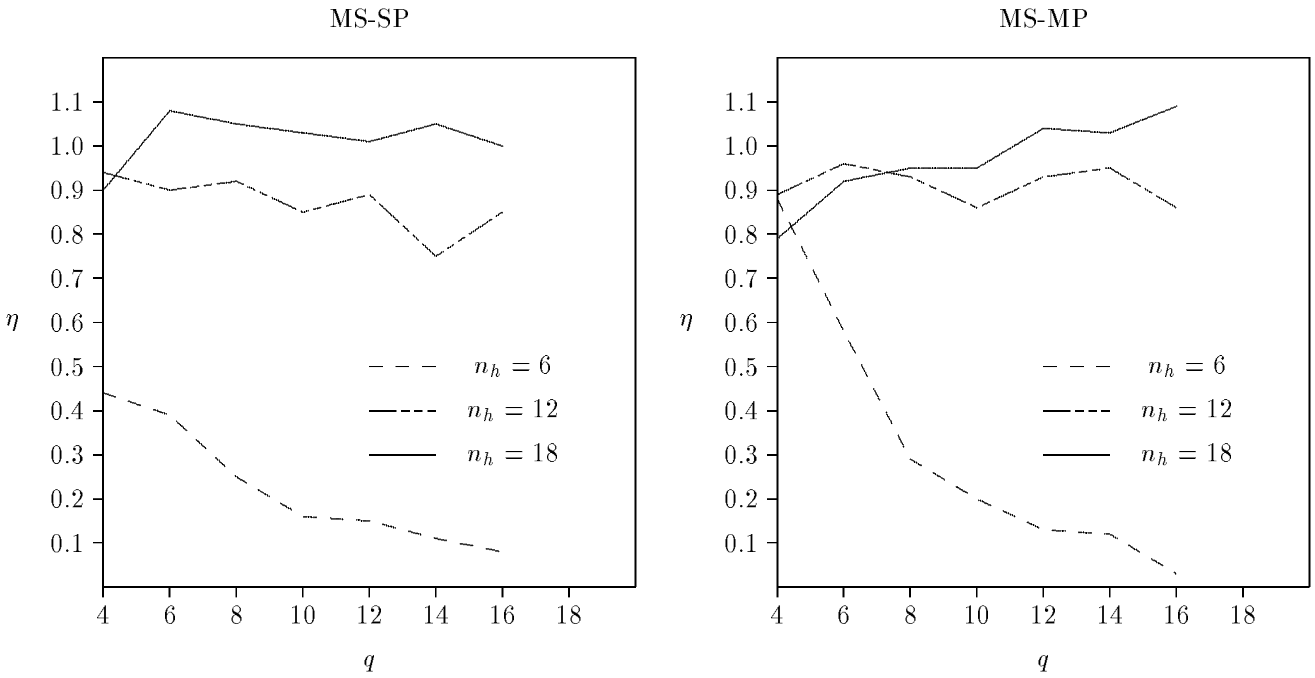


Figure 6: Efficiency ($\eta$) vs. number of processors ($q$): MS-SP and MS-MP strategies

processor efficiency $\eta(q)$ measures the contribution of each processor to the parallel solution, when $q$ processors are employed:

$$\eta(q) = \frac{\mathcal{S}(q)}{q} .$$

In this work, the time $\tau_{seq}$ corresponds to the execution of the sequential tabu search algorithm **RN-STM-TS**, presented in Section 3.

All four parallel strategies start from the same initial solution, as far as they all use the same heuristic to produce it. Also, all of them find the same best solution. Therefore, the quality of the solutions obtained by the four strategies is the same, and they may be compared exclusively on the basis of 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 10. As all four strategies led to rather similar results, only some figures with selected results are shown.

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 larger problem sizes.

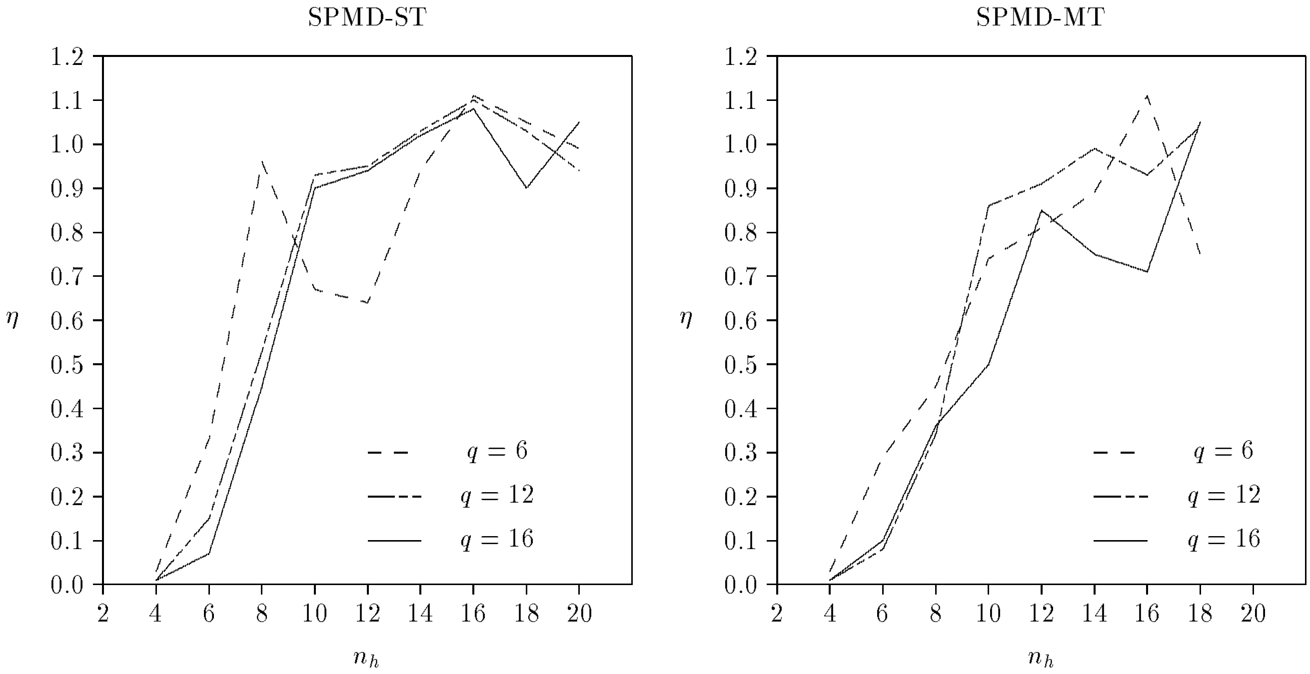


Figure 7: Efficiency ($\eta$) vs. problem size ($n_h$): SPMD-ST and SPMD-MT strategies

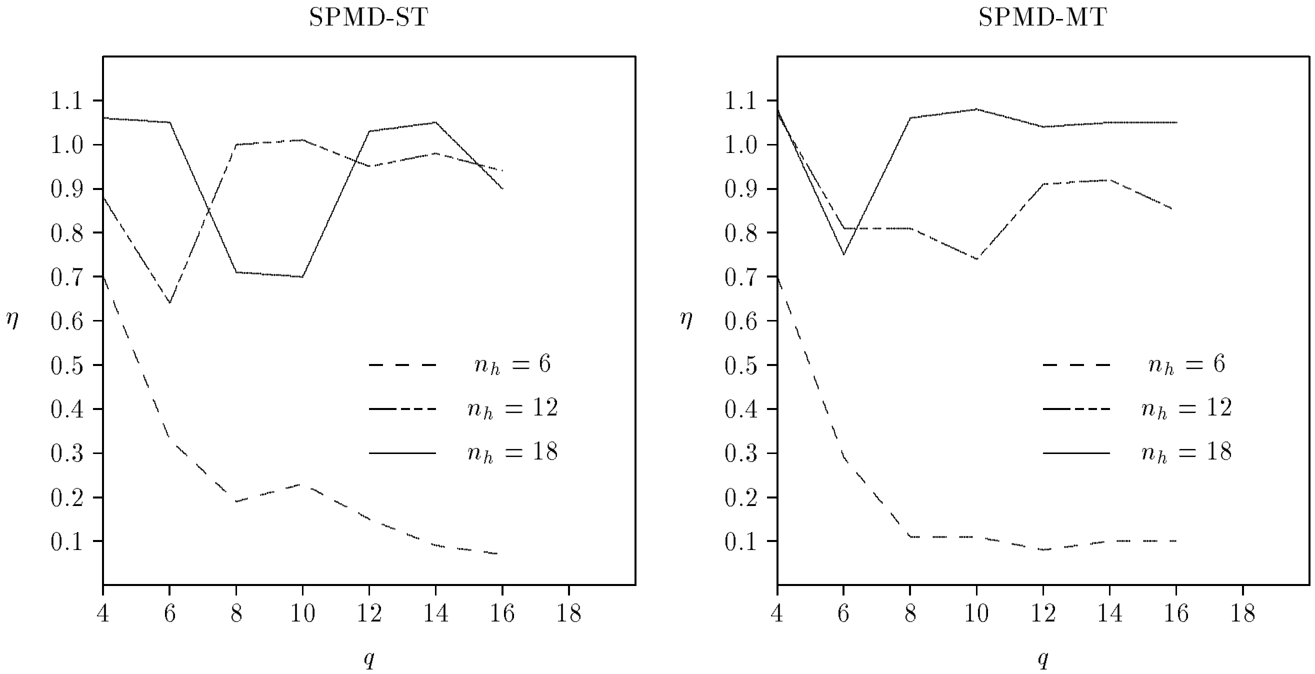


Figure 8: Efficiency ($\eta$) vs. number of processors ($q$): SPMD-ST and SPMD-MT strategies

On the other hand, we can observe very low efficiency values for small size problems. In these cases, the neighborhood partitions distributed among parallel tasks are not sufficiently large to overcome the 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. The discrepancies in the efficiency values plotted in Figures 5 to 10 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.

The comparison between MS-SP and MS-MP strategies, based on the master-slave model, is illustrated in Figures 5 and 6. Both strategies present similar results. We can notice that the efficiency: (i) increases with the problem size, for a fixed number of processors, and (ii) decreases with the number of processors for a given problem size, due to the increase in the communication/processing ratio. The result of the load balancing scheme implemented in the MS-SP strategy is significant if one considers that, in this scheme, the master-task does not execute any best neighbor partial search. The neighborhood to be searched is partitioned exclusively between slave-tasks. The master remains as a partition distributer (manager) and is in charge of the comparison and selection of the best global move at each iteration. Thus, load balancing is so effective that it compensates the decrease in the processing capacity dedicated to the search of the best neighbor solution. In this case, the comparisons between successive best partial moves are immediate, since the master is completely dedicated to receiving the results sent by the slaves. The still increasing efficiency values attained by the MS-MP strategy observed in Figure 6 for larger problems can also be explained by the efficiency of the load balancing, which does not exist in the MS-SP strategy.

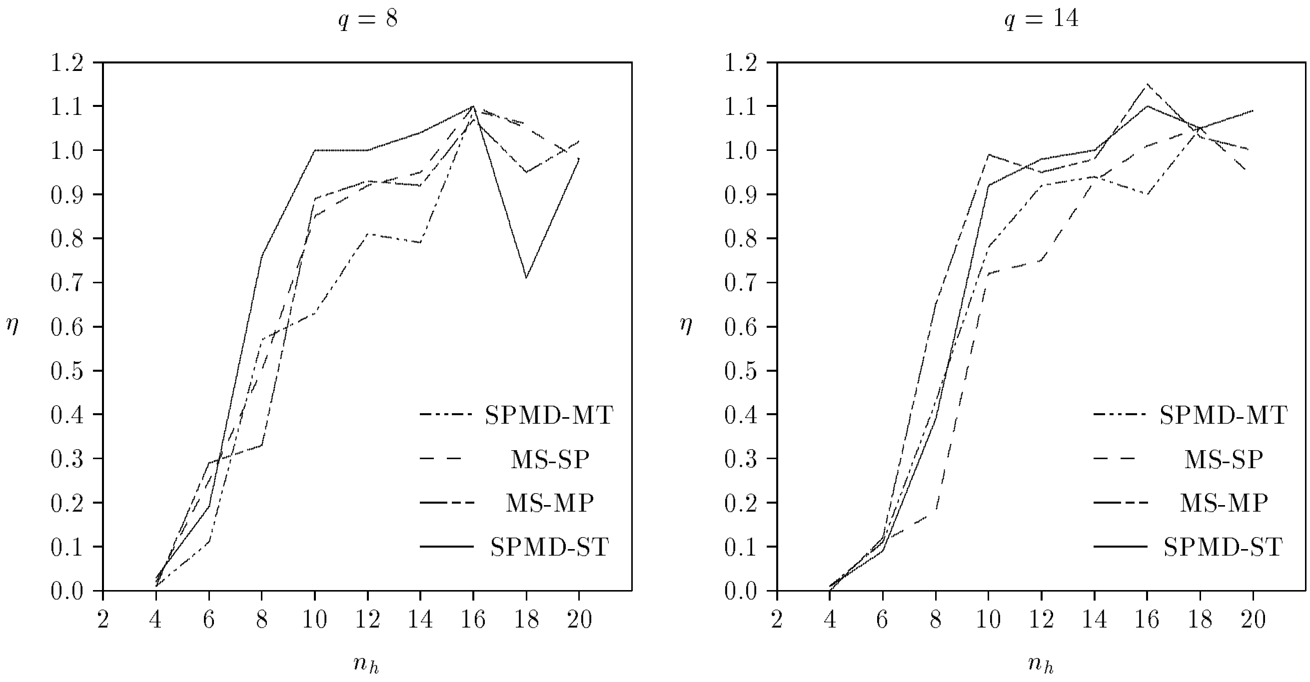


Figure 9: Efficiency ($\eta$) vs. problem size ($n_h$) for $q = 8$ and $q = 14$

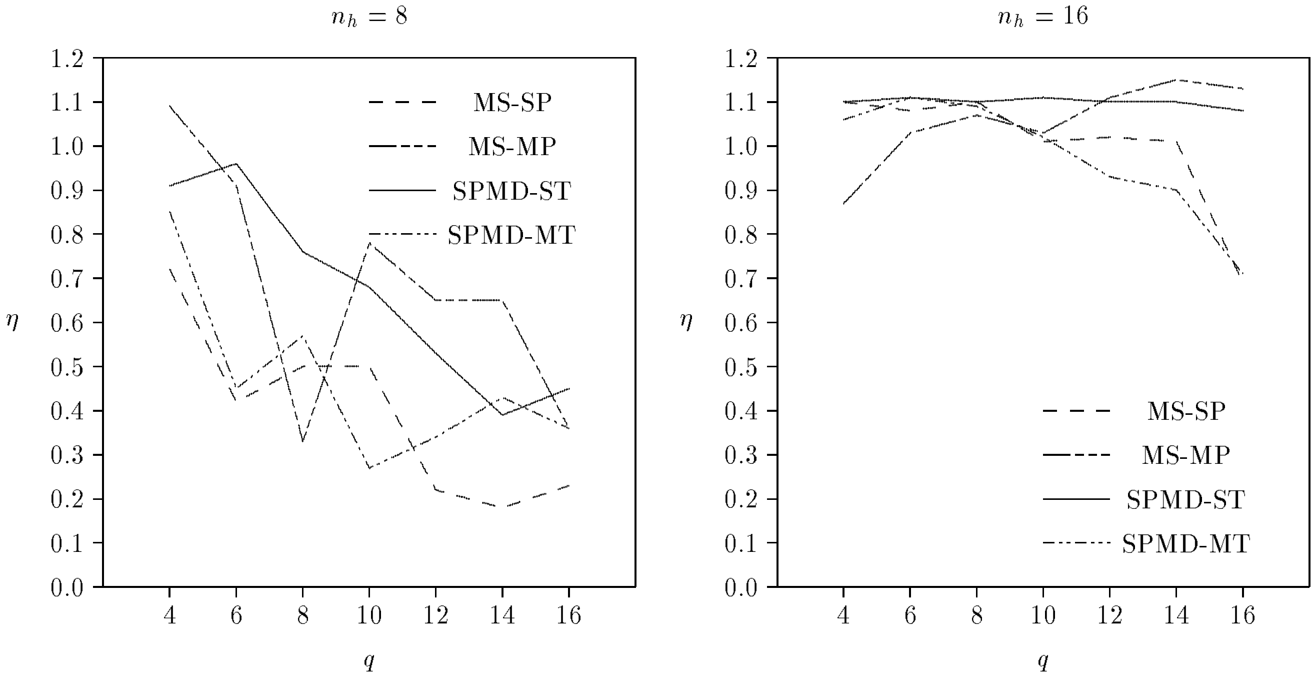


Figure 10: Efficiency ($\eta$) vs. problem size ($n_h$) for $n_h = 8$ and $n_h = 16$

Figures 7 and 8 show performance results for the strategies using the SPMD model. In the case of SPMD-ST, the comparison cycle depends on the parent-task initiative, following the natural order of the logical ring. Each child-task sends the message with its best partial move after having received a message from its predecessor in the ring. However, due to the small size of these messages and to the speed of the communication media, the two message exchange cycles do not produce high overhead. The rigid synchronism determined by the SPMD-ST strategy is compensated by the performance increase due to the overlap between communication and computation. Similar results were obtained for the SPMD-MT strategy. Although SPMD-MT presents a communication scheme with a less rigid synchronism than SPMD-ST, this fact is attenuated by the larger number of exchanged messages.

The interesting aspects of the logical ring communication scheme, either under the single token approach or under the multiple token approach, are the communication descentralization and the distribution of the comparison procedure of the best partial moves. Although the logical ring communication scheme may appear to be more promising in terms of the overall performance, specially in the multiple token case, where several task pairs may communicate simultaneously, it must be noted that this potential advantage is totally dependent on the underlying parallel machine architecture. The standard PVM does not take any profit from the architectural characteristics to perform message difusion. The communication scheme in the IBM SP1 under PVM is as if the processors were workstations connected through a local Ethernet network. The access procedure to the communication media under this organization determines that only one message will be in transit at each moment. In this case, there is no benefit from the use of a logical ring communication scheme.

Finally, Figures 9 and 10 present overall performance results, with a slight superiority of the MS-MP and SPMD-ST strategies.

# 10 Enhancing Solution Quality with Parallel Tabu Search

As described in Section 3, a move in which task $t_i \in T$ is taken out from the task list of processor $\mathcal{A}_s(t_i)$ and transferred to that of $p_l \in P$ at position $pos$ maybe fully characterized by $(\mathcal{A}_s(t_i), t_i, p_l, pos)$. However, due to the high computational cost of evaluating each move, the neighborhood was restricted by considering a unique task list dependent position for task $t_i$ in the task list of the target processor $p_l$.

The numerical results presented in the previous section have shown that parallelization attains almost linear efficiency and reduces significantly the execution time of the tabu search algorithm. This gives rise to the idea of enhancing the search of the solution space, in order to look for still better solutions with smaller makespans and larger cost reductions with respect to the initial solution, where the cost reduction defined in [28] is given by

$$\frac{c(s_0) - c(s^*)}{c(s_0)},$$

$s^*$ is the best found solution and $s_0$ is the initial solution.

The modified search is based on examining not only a restricted neighborhood, but instead the complete neighborhood. All moves obtainable by moving each task to all possible positions in the task list of each target processor, will be evaluated. Under this approach, each move of task $t_i \in T$ from a given solution $s$ will be again characterized by the tuple $(\mathcal{A}_s(t_i), t_i, p_l, pos)$ and not simply by $(\mathcal{A}_s(t_i), t_i, p_l)$, as in the case of the restricted neighborhood. The **UN-STM-TS** (Unrestricted Neighborhood Short Term Memory Tabu Search) algorithm investigates, for each task-processor $(t_i, p_l)$ pair, all the possible positions of task $t_i$ in the task list of $p_l$.

The algorithm **UN-STM-TS** was implemented using the MS-SP parallelization strategy and tested using 8 processors of an IBM-SP1. Some slight quality changes were obtained with the algorithm **UN-STM-TS** and are shown in Table 3. In some cases ($n_h = 8, 12, 18$) the solutions obtained with **UN-STM-TS** are better, although this behavior is not systematic and in some cases the new solutions are even of inferior quality. Although this type of behavior is absolutely coherent with heuristic search techniques, it possibly reveals that the sequential tabu search algorithm **RN-STM-TS** is well ajusted and that it would be difficult to obtain systematically better solutions. It would also be interesting to have an exact algorithm [21] for solving the problem of task scheduling on heterogenous processors under precedence constraints, which would eventually allow the determination of the effective quality of the solutions obtained with the algorithm **RN-STM-TS**.

| Problem size | Cost reduction (%) | |
|---|---|---|
| $n_h$ | **RN-STM-TS** | **UN-STM-TS** |
| 8 | 26.5 | 28.6 |
| 10 | 25.4 | 25.4 |
| 12 | 15.1 | 17.7 |
| 14 | 17.1 | 15.0 |
| 16 | 12.1 | 11.8 |
| 18 | 7.4 | 10.7 |

Table 3: Cost reductions attained by algorithms **RN-STM-TS** and **UN-STM-TS** according to problem size ($n_h$)

# 11 Conclusions

We have presented four different message-passing parallelization strategies for the implementation of a tabu search algorithm developed for the solution of a problem of task scheduling into heterogeneous processors under precedence constraints. All strategies are synchronous and based on the decomposition of the neighborhood of the current solution at each iteration. 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 the achievement of almost linear efficiency in the majority of the test problems.

The task scheduling problem considered in this study is characterized by very large and costly to explore neighborhood structures. However, the speedups achieved through simple parallelization techniques made possible the use of a less restricted neighborhood search, considering all possible positions for a task in the task list of the target processor. Better solutions were found in some cases using this new move characterization. As good performance results for the parallelization of tabu search for other problems with similar structure have also been reported in recent literature, this suggests an interesting research path to overcome the inherent intensive computational demand of this kind of algorithm, giving way to the exploration of other aspects, such as more intricate tabu search features and asynchronous parallelization schemes.

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

## References


[1] 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.

[2] N. August and T. Mautor, ”Méthode tabou massivement parallèle pour le problème d’affectation quadratique”, Rapport de Recherche no 2182, Institut National de Recherche en Informatique et en Automatique, December 1993.

[3] G. Authié, A. Ferreira, J.L. Roch, G. Villard, J.Roman, C.Roucairol e B. Virot (editores), *Algorithmes Parallèles: Analyse et Conception*, Hermès, Paris, 1994.

[4] J. Chakrapani and J. Skorin-Kapov, “Massively Parallel Tabu Search for the Quadratic Assignment Problem”, Annals of Operations Research 41(1993), 327-341.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] B. Garcia and M. Toulouse, “A Parallel Tabu Search for the Vehicle Routing Problem with Time Windows”, *Computers and Operations Research* 21 (1994), 1025–1033.

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

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

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

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

[16] 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.

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

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

[19] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., CA, 1994.

[20] T.G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall International, 1992.

[21] N. Maculan, S.C.S. Porto, C.C. Ribeiro and C.C. de Souza, *A New Formulation for Scheduling Unrelated Processors under Precedence Constraints*, Research report PUCRioInf-MCC37/95, Catholic University of Rio de Janeiro, Department of Computer Science, Rio de Janeiro, 1995.

[22] T.Mautor and L. Stein, "Recherche Tabou Parallèle Appliquée au Probléme de Placement de Tâches", Rapport de Recherche RR-94/33, Laboratoire PRISM, *Université de Versailles*, 1994.

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

[24] 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.

[25] 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.

[26] S.C.S. Porto and D.A. Menascé, "Processor Assignment in Heterogeneous Message Passing Parallel Architectures", *Proceedings of the Hawaii International Conference on System Science*, 186–191, Kauai, 1993.

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

[28] 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.

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

[30] M. RASSAI, *Parallélisation d'une méthode approchée pour la résolution du problème d'affectation quadratique*, Mémoire d'ingénieur IIE, Conservatoire National des Arts et Métiers, Institut d'Informatique d'Entreprise, Evry, 1993.

[31] 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.

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

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

[34] E. TAILLARD, *Recherche Iterative Parallèle*, PhD Thesis no. 1153, Département de Mathématiques, École Fédérale Polytechnique de Lausanne, 1993.

[35] E. TAILLARD, "Parallel Taboo Search Techniques for the Job Shop Scheduling Problem", *ORSA Journal on Computing*, 6 (1994), 108–117.

[36] S. VO$\beta$, "Tabu Search: Applications and Prospects", Technical report, Technische Hochshule Darmstadt, 1992.

[37] 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.

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