Parallelizing the Microcanonical Optimization Metaheuristic: A Case Study for the Task Scheduling Problem

Stella C.S. Porto[1]*, André M. Barroso[1], José R. A. Torreão[1]

[1] Instituto de Computação
Universidade Federal Fluminense
Rua Passo da Pátria 156
24210-240 Niterói, RJ, Brazil
{stella,melon,jrat}@caa.uff.br

Abstract—
The present work deals with the parallelization of the microcanonical optimization metaheuristic (μO), and implements a parallel algorithm for the task scheduling problem on heterogeneous processors under precedence constraints without communication delays. The μO algorithm consists of two iterative procedures – the initialization and the sampling phases, which are alternately applied. Our parallel implementations are based on a scheme where p processes execute alternate parallel versions of the initialization and sampling phases, coupled at a synchronization point. They have been implemented on a network of workstations using the MPI communication library, and an evaluation of the quality of the solutions generated has been performed for different sets of the algorithm parameters. The solution quality has been measured by the makespan reduction achieved in comparison with the best greedy algorithm, and with tabu search, for the same problem instances [7, 10]. The conditions under which the new algorithm is able to show a superior performance are then highlighted by our preliminary results.

Keywords— **Metaheuristic parallelization, microcanonical optimization, task scheduling, network of workstations**

I. Introduction

For $\mathcal{NP}$-hard combinatorial problems, exact search algorithms degenerate into complete enumeration, with exponential increase in CPU time, when problem size increases. Therefore, in practice, heuristic search algorithms are necessary for finding sub-optimal solutions to these problems [5]. Obtaining good solutions with a guided iterative local search method, such as simulated annealing, is often hampered by long computational times, due to the great number and/or the computationally intensive character of the required iterations. Therefore, efficient parallel implementations of the search algorithm can significantly increase the size of the problems that can be tackled in plausible processing times [1].

The microcanonical optimization heuristic (referred, heretofore, as μO) was proposed by Torreão and Roe [14] for image processing applications, and later refined and employed in the solution of the traveling salesman problem, yielding significant results [4]. The basic algorithm consists of two iterative procedures – initialization and sampling –, which are alternately applied. The initialization procedure implements an iterative improvement search, in order to approach a local minimum solution, while the sampling procedure tries to free itself from that local minimum, but at the same time keeping close to it, in terms of cost. The μO heuristic shows good adaptability towards parallelization, due to its alternating two-phase structure and to the randomness of its move selection procedure, this latter aspect also being responsible for its controlled diversification characteristic.

Task scheduling is a challenging problem and is known to be $\mathcal{NP}$-hard. It deals with the choice of the partial order under which a certain number of tasks should be performed, and with the assignment of such tasks to processors in a parallel/distributed environment. The intractability of this problem has led to the proposal of a large number of heuristics. Porto and Menascé [7], for instance, proposed greedy algorithms for processor assignment of parallel applications modeled by task precedence graphs in heterogeneous multiprocessor architectures without communication delays. Porto and Ribeiro [10, 9] subsequently applied the tabu search metaheuristic to the same problem using sequential and parallel implementations, which generated identical best final solutions.

After proposing and describing general parallelization strategies for μO metaheuristic, we report on the results of a preliminary study of these strategies applied to the task scheduling problem. We have decided on two implementations, among some considered alternatives, where p processes execute alternate versions of the initialization and sampling phases, coupled at a synchronization point. We have shown that the parallel μO heuristics, starting from a

*The author has been partially supported by: Project SIAM/DCC/UFMG (grant MCT/FINEP/PRONEX 76.97.1016.00), Project FINEP – Recope/SAGE and CNPq Research Scholarship

random initial solution, achieves the same quality results as the earlier implementation [10] of tabu search starting from the deterministic solution provided by the DES+MFT algorithm.

The next section describes the microcanonical optimization heuristic and discusses its parallelization potential. Section III presents the scheduling model used in the context of this work. In Section IV we describe the parallel algorithm based on the μO heuristic for the scheduling problem. Section V reports our preliminary findings concerning the quality of the solutions obtained with our parallel algorithm, for different parameter sets of the μO heuristic. Section VI closes this paper with final remarks and directions for future work.

II. The Microcanonical Optimization (μO) Approach

To describe the μO heuristic, we first consider a general combinatorial optimization problem, P_o, formulated as

minimize $c(s)$
subject to $s \in S$,

where S is a discrete set of feasible solutions, and $c(s)$ is said to be the cost of a solution s.

A local search approach for solving problem P_o starts from an initial solution $s^0 \in S$ and, at each iteration, generates a set of new solutions in the neighborhood, $N(s)$, of the current solution, s, through the application of slight perturbations, called *moves*. A move is an atomic change which transforms the current solution s into one of its neighbors, say $\bar{s}$ ($s \oplus move = \bar{s}$). The $movevalue = c(\bar{s}) - c(s)$ is the difference between the value of the cost function after the move, $c(\bar{s})$, and its value before the move, $c(s)$. At a given iteration, the search procedure evaluates the $movevalues$ corresponding to the set of new solutions, and selects one of them as the new current solution, according to a rule which tries to guarantee that, after a plausible number of such iterations, the function $c(s)$ will be minimized.

There have been proposed a great number of local search optimization heuristics along these lines, differing basically on the nature of the move selection rule employed. μO is one such general-purpose heuristic (metaheuristic) which, similarly to the well-known simulated annealing algorithm, is based on principles of statistical mechanics.

μO consists of two iterative procedures – the initialization and the sampling phases [4, 13, 14]– which are alternately applied. The initialization phase searches randomly through the solution space for a lower-cost solution. It may be seen as a "hill-descending" procedure, since, at each iteration, a move is randomly proposed which is accepted only if it imposes a cost decrease on the current solution, i.e., if $movevalue < 0$. The goal here is to quickly approach a local-minimum solution. Optionally, an aggressive implementation of this phase can be chosen, meaning that the algorithm, at each iteration, will pick the best candidate in a subset of possible moves. During the initialization, a list of the moves rejected (with a size given by $size_{list}$) for leading to higher-cost solutions ($movevalue \geq 0$) is compiled, to be used in the subsequent sampling phase. The initialization ends when a certain number of consecutive moves, $maxmove_{init}$, have been rejected, meaning that the algorithm is close to a local minimum.

In the sampling phase, μO aims at freeing itself from the local minimum reached in the initialization, at the same time trying not to stray too much, in terms of cost, from that solution. Considering a 3D space as a metaphor of the search space, one may envision the μO heuristic as trying to get "around the hill", instead of "hill climbing", in order to break free from the local minimum. This is achieved by implementing the so-called Creutz algorithm of statistical physics [3], where an extra degree of freedom – called the *demon* – generates controlled disturbances (moves) on the current solution. At each sampling iteration, the randomly proposed move will only be accepted if the demon can supply or receive the cost variation ($movevalue$) implied by that move. The demon thus restricts the maximum cost variation which is allowed if a move is to be effected. It is defined by two parameters: its capacity, d_{max}, and its initial load (cost), d_i. The sampling phase generates a sequence of solutions of fixed cost, except for small fluctuations which are modeled by the demon. Calling $c(s_i)$ the cost of the solution s_i obtained in the initialization, and d and $c(s)$, respectively, the costs of the demon and of the solution s at a given instant in the sampling, we will have $c(s) + d = c(s_i) + d_i = \text{constant}$. Thus, the sampling phase generates solutions in the cost interval $[c(s_i) - d_{max} + d_i, c(s_i) + d_i]$, with $d_i, d_{max} << c(s_i)$.

Therefore, d_i and d_{max} are the main parameters of the sampling phase. They are determined thus: the list of rejected moves compiled in the initialization is sorted in growing order of the cost jumps, and two of its lower entries are chosen as the values of demon capacity and initial load. The idea is that such values will be representative of the hills found in the landscape of the solution space, in the region being searched, thus being adequate for defining the magnitude of the perturbations required for the evolution of the current solution in the sampling phase. This phase stops when a given number of iterations, $maxiter_{samp}$ has been reached, after which a new initialization procedure is run. The algorithm thus proceeds, alternating the two phases, until a stopping condition (such as a certain number of iterations without global improvement, $maxiter_{alg}$) is obtained.

A. Parallelizing the μO Metaheuristic

Efficient parallel implementations of search algorithms can significantly increase the size of the problems that can

be solved. Work on parallel heuristic search algorithms is relatively recent [5] with some papers on parallelizing metaheuristics such as tabu search [9, 12].

In the case of the μO heuristic, one may envision different possibilities for parallel implementations based on two distinct parallelization approaches, namely: *Neighborhood-Partitioning* (NP) and *Multi-Threading* (MT). Neighborhood-Partitioning represents the class of strategies where p processes start the search from an unique single solution, a *neighborhood partition* (a subset of the neighbor solutions) is given to each process at the beginning of each iteration and each process performs the local search strictly over this previously determined partition, during this iteration. Variations of such approach, for example, are based on (i) the partitioning scheme which defines the subset of neighbor solutions on which each process will work and (ii) the heuristic parameter settings employed by each process. Taking into account this latter feature, similarly as for tabu search in [2], one may define two different trends: *Single-Parameter-Setting* (SPS) and *Multiple-Parameter-Setting* (MPS). In a single parameter setting, all processes use the same parameter values, while, in the multiple parameter setting, processes have different values for the heuristic parameters.

The Multi-Threading strategy, on the contrary, is initiated from different starting solutions, one for each of the p parallel processes, at the most. Each process performs the iterative search over the entire neighborhood of its own starting solution point. Again, in this case, it is possible to establish variations based on the heuristic parameter settings determined for each process, namely: the SPS and MPS approaches previously mentioned.

The parallel μO algorithm will be composed of parallel initialization and parallel sampling phases. Between each pair of alternating phases, one may implement what we call *interphase stages*, during which certain interprocess communication patterns may take place before the next phase begins. The intensity of the communication depends directly on the parallel strategies being implemented for each phase. The interphase stages, in some cases, play the role of synchronization points, but on the other hand, may also not impose any waiting barrier for the parallel processes. In this sense, when building a parallelization strategy for the μO heuristic, one must specify: (i) a parallel strategy for the initilization phase, (ii) a parallel strategy for the sampling phase, and (iii) the interprocess communication pattern at the interphase stages. Figures 1 and 2 describe the μO initialization and sampling phases, respectively. This description is generic enough to encompass any parallel μO algorithm.

```
procedure Initialization_Phase (s, NP(s))
begin
    Let NP(s) be the neighborhood partition to be worked on during the
initialization phase;
    Empty list-of-rejected-moves;
    Let maxmoves_init be the maximum number of consecutive rejected
moves;
    Let s be the starting solution of the initilization phase;
    num_moves ← 0;
    while (num_moves < maxmoves_init) do
    begin
        Choose a feasible move randomly, such that s ⊕ move = s' and
s' ∈ NP(s);
        movevalue ← c(s') − c(s)
        if (movevalue ≥ 0) then
        begin
            Put move in the list-of-rejected-moves;
            num_moves ← num_moves + 1;
        end if
        else
        begin
            num_moves ← 0
            s ← s'
        end else
    end while
end
```

Fig. 1. Procedure of the initialization phase

III. The Task Scheduling Problem

Parallel applications with regular and well-known behavior, where task execution time estimates are fairly reliable, are suited for static task scheduling, which is the case of a great majority of scientific applications. For these applications, the static scheduling algorithm is executed once, before the execution of the parallel program, which is then run several times according to the previously obtained schedule. Consequently, even if the scheduling algorithm is a costly procedure, this cost will be amortized throughout the numerous executions of the parallel application, since the obtained schedule is repeatedly applied.

Processor heterogeneity, here represented by processors with different processing speeds, has already demonstrated the potentiality in reducing the performance degradation resulting from the execution of the inherent serial fractions of the parallel application on a homogeneous processor set [6]. Task scheduling on this heterogeneous environment is even more complex than on a homogeneous one [7], since the assignment of a certain task to different processors may significantly affect the execution times.

For the sake of simplicity, the task scheduling (or processor assignment) problem considered in this work does not explicitly represent intertask communication costs, Thus, in our scheduling model, a parallel application Π with a set of n tasks $T = \{t_1, \cdots, t_n\}$ and a heterogeneous multiprocessor system composed by a set of m interconnected processors $P = \{p_1, \cdots, p_m\}$ can be represented by a task precedence graph $G(\Pi)$ and an $n \times m$ matrix μ, where

```
procedure Sampling_Phase (s, NP(s))
begin
    Let NP(s) be the neighborhood partition;
    Select d_max and d_i from the list-of-rejected-moves;
    Let maxiter_samp be the maximum number of iterations;
    Let s be the starting solution of the sampling phase;
    num_iter ← 0;
    d ← d_i
    while (num_iter < maxiter_samp) do
    begin
        Choose a feasible move randomly,
        such that s ⊕ move = s' and s' ∈ NP(s);
        movevalue ← c(s') − c(s)
        if (movevalue ≤ 0) then
        begin
            if (d − movevalue ≤ d_max) then
            begin
                s ← s';
                d ← d − movevalue;
            end if
        end if
        else { movevalue > 0 }
        begin
            if (d − movevalue ≥ 0) then
            begin
                s ← s';
                d ← d − movevalue;
            end if
        end else
        num_iter ← num_iter + 1;
    end while
end
```

Fig. 2. Procedure of the sampling phase

$\mu_{kj} = \mu(t_k, p_j)$ is the estimated execution time of a task $t_k \in T$ at processor $p_j \in P$. Each processor can run one task at a time, all tasks can be executed by any processor, and processors are said to be uniform in the sense that $\frac{\mu_{kj}}{\mu_{ki}} = \frac{\mu_{lj}}{\mu_{li}}, \forall t_k, t_l \in T, \forall p_i, p_j \in P$. In a framework with a single heterogeneous processor, the heterogeneity may be expressed by a unique parameter called *processor power ratio*, PPR, which is the ratio between the processing speed of the fastest processor, and that of the remaining ones (those in the subset of homogeneous processors).

Given a solution s for the scheduling problem, a processor assignment (task scheduling) function is designed as the mapping $\mathcal{A}_s : T \rightarrow P$. A task t_k 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 assignment of the set of tasks onto that of the processors, in terms of the makespan of the parallel application, i.e. the completion time of the last task being executed, which is the cost of the solution s, $c(s)$. 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. 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 may be characterized by a simple representation given by $(\mathcal{A}_s(t_i), t_i, p_l)$, as long as the position that task t_i will occupy in the task list of procesor p_l is uniquelly defined.

The computation of the makespan of a parallel application [10] presents $O(n^2)$ time complexity, which determines a high computational cost for the entire neighborhood evaluation.

IV. The Parallel μO Algorithm

Following the parallelization trends discussed before, we have developed two distinct parallel implementations for the task scheduling problem with the μO heuristic, both based on a master-slave scheme where p processes execute alternate parallel versions of the initialization and sampling phases, coupled at an interphase stage which works as a synchronization point coordinated by the master process. The name given to both algorithms is based on the ordered acronyms of the parallel strategies used during the initialization and sampling phases respectively.

MT/NP parallel version In the first implementation, the parallelization strategy of the initialization phase is based on an MT-SPS approach, while the strategy used for the sampling phase is based on an NP-SPS approach. The first interphase stage, between initialization and sampling, determines a synchronization point, where the master gathers results from all the processes and determines the starting solution of the following sampling phase. The second interphase stage, between the sampling and initialization phases, has no communication between processes.Thus, the processes continue from the sampling to the following initialization phase without any delay.

During initialization, each process executes the random "hill-descending" search procedure previously described, over the entire solution space. At the synchronization point, one of the processes, called the master process, receives the results of the initialization phase from all processes and selects the best solution. If the master concludes that the search should proceed (stopping conditions have not been attained), it broadcasts the selected solution to all processes. Thus, every process will start the sampling phase from the same solution. However, differently from the initialization phase, the solution space is now divided into p disjoint and equally sized regions, and each process is made responsible for the search over one particular region during the sampling phase. A region of the solution space is defined by a given subset of the tasks (subject to the scheduling) which are allowed to move (from one pro-

cessor to another) during the heuristic search. In order to assign similar workloads to each process, the subset of tasks is randomly selected by the master in the beginning of each sampling phase. There is no synchronization between the sampling phase and the following initialization phase, meaning that each process starts a new initialization phase from the solution reached during its previous sampling phase.

MT/MT parallel version This second implementation distinguishes itself from the previous one due exclusively to the parallelization approach employed during the sampling phase. In this case, the processes still start from the same best solution found during the initialization phase, but the neighborhood is not partitioned among them. Each process is free to perform any feasible trial move in the sampling phase, without the restrictions imposed in the MT/NP parallel version. Initialization phase and interphase stages remain unchanged.

Figures 4 and 3 describe the master and slave algorithms which implement the MT/NP parallel strategy described above. The master and slave algorithms for the MT/MT parallel strategy are not separately presented because: (i) the majority of our computational tests were done considering the MT/NP version, and (ii) these algorithms are very similar to the ones described in Figures 4 and 3, differing only by the interphase stage, where for MT/MT strategy, the master does not divide the neighborhood into partitions before entering the sampling phase as explained above.

```
μO slave(p)-algorithm { slave process version }
begin
    Let p be the slave process identification;
    Obtain s_0 ∈ S randomly;
    s_p ← s_0;
    continue ← TRUE;
    while (continue = TRUE) do
    begin
        Inicialization_Phase (s);
        Send to master (s_p,list-of-rejected-moves(p));
        Receive from master message: (continue);
        if(continue = TRUE) then
        begin
            Receive from master (N_p(s),s_p)
            Sampling_Phase (s_p,N_p(s));
        end if
end
```

Fig. 3. Slave version of the parallel μO algorithm

V. Preliminary Numerical Results

Different parameters are needed to fully specify the parallel μO algorithm, and they were studied side-by-side, in order to determine the conditions under which the algorithm provides the best results. As the programs were executed on a non-dedicated network of workstations (NoW) with great variations of processor workload, response time was not considered an appropriate metric for evaluating algorithm performance, and thus are not reported here The parameters under study are those already mentioned in Section II, namely: (i) $maxiter_{alg}$ is the maximum number of iterations of the parallel μO algorithm without improvement on the best found solution, s^*; (ii) $maxmove_{init}$ is the maximum number of consecutive rejected moves during the initialization phase; (iii) $maxiter_{samp}$ is the maximum number of iterations during the sampling phase; (iv) $size_{list}$ is the size of the list of rejected moves during initialization phase; and (v) pos_{demo} is the position on the list of rejected moves which provides the value of the demon capacity, d_{max}, during the sampling phase (we have assumed, in our implemen-

```
μO master-algorithm { master process version }
begin
    Let m be the master process identification;
    Obtain s_0 ∈ S randomly;
    Set maxiter_alg to the max. num. of iter. of the algorithm,
    without improvement on the best global solution, s*;
    num_iter_alg ← 0;
    s* ← s_0;
    s ← s_0;
    while (num_iter_alg < maxiter_alg) do
    begin
        Inicialization_Phase (s,N(s));
        {N(s) is the entire neighborhood}
        for each slave process p do
        {communication between slaves and the master process}
        begin
            Receive from p (s_p,list-of-rejected-moves(p));
            if (c(s_p) < c(s)) then
                s ← s_p;
        end for
        if (c(s) < c(s*)) then
        begin
            s* ← s;
            num_iter_alg ← 0;
        end if
        else
            num_iter_alg ← num_iter_alg + 1;
        if (num_iter_alg ≥ maxiter_alg) then
            Send to all slaves message: (continue = FALSE);
        else
        begin
            Send to all processes message: (continue = TRUE);
            Determine NP_p(s) for each process p (and master m);
            for each slave process p do
                Send to slave process p: (NP_p(s),s)
            Sampling_Phase (s,NP_m(s));
        end else
    end while
end
```

Fig. 4. Master version of the parallel μO algorithm

tations, that $d_i = d_{max}$).

The evaluation of solution quality is based on the relative cost reduction, computed as: $\mathcal{R} = \frac{c(s^0)-c(s^*)}{c(s^0)}$ where s^0 is the solution obtained by the greedy algorithm DES+MFT (*Deterministic Execution Simulation with Minimum Finish Time*) [7] and s^* is the best solution found by the μO algorithm. The DES+MFT algorithm was selected because it produces deterministic final solutions, has previously shown the best results published in recent literature, and has also been used in [10] to evaluate the quality of the results yielded by the tabu search algorithm (sequential and parallel) for the same scheduling problem. The DES+MFT algorithm iteratively schedules tasks following their partial ordering (described by the task precedence graph), according to the simulated execution of the parallel application (DES), which is based on the estimated task execution times. Scheduling decisions, at each iteration, are made according to the minimum finishing time (MFT) strictly for the tasks which are considered to be schedulable, i.e. tho se whose predecessors have already been executed during this simulation. The reader is referred to [7] for a detailed description of the DES+MFT algorithm.

A. Experimentation Framework

We have focused our preliminary numerical experimentation on two problem instances. Both use the task graph structure of the Mean Value Analysis (MVA) [11] solution package for product form queuing networks, which is a diamond-shape graph and presents *wavefront* precedence relations. The choice of this unique type of task precedence graph is based on the results reported in [8], which demonstrated and explained that for diamond-shaped graphs, tabu search was able to achieve significantly superior values of relative cost reduction ($\mathcal{R}$) than with other graphs with fewer precedence constraints (fewer task dependencies in execution order). Problem instances may be distinguished by regular or irregular estimated task execution times, thus: (i) MVA_{reg} is the MVA task precedence graph with $n = 100$ tasks, task execution times equal 1, except for the tasks over the central vertical axis [10], which assume task execution times equal 2, number of processors to be assigned $m = 5$, and $PPR = 5$; and (ii) MVA_{rand} is the MVA task precedence graph with $n = 100$ tasks, task execution time are values between $[1.9, 2.1]$ for the tasks on the central vertical axis of the graph, and between $[0.9, 1.1]$ for the rest of the tasks, number of processors to be assigned $m = 4$, and $PPR = 4$. The values for m and PPR were selected based on the work presented in [8] which has shown best results for the parallel tabu search algorithm. The irregularity of task execution time is considered here as a mean to study a less symmetric parallel application, which presumably produces a more complex solution space. The parallel μO algorithm was implemented using C+MPI on 4 processors of a network of workstations. We have also tested results of a sequential version, by executing the parallel algorithm on a single processor. In the following, we report our main conclusions based on a series of experiments, and present a sample of the obtained numerical results.

B. Preliminary Solution Quality Evaluation

We have restricted ourselves to showing only the best results which are significant and lead to conclusions about the behavior of the parallel μO algorithm. The results presented in Tables I through IV were obtained using the MT/NP strategy (parallel and sequential executions), while Table V compares the results obtained with both MT/NP and MT/MT parallel algorithms.

Table I shows the solution quality results, measured by the relative cost reduction, $\mathcal{R}$, for different values of $maxiter_{samp}$, $maxmove_{init}$ and pos_{demo} parameters. The best result (reduction of 25.4%) obtained using tabu search [10] (which was primarily evaluated considering exclusively regular MVA task precedence graphs) is also achieved using the parallel μO algorithm. It it worth noticing that in the case of tabu search, the algorithm has its initial solution given by the greedy algorithm DES+MFT, which is also used as comparison for both tabu search and μO. However, μO starts from different random initial solutions, which yield different final results. These tables also demonstrate that the values of relative cost reduction decrease with increasing values of $maxiter_{samp}$ and $maxmove_{init}$. In general, the best value for pos_{demo} equals 10, but in some cases greater values lead to higher relative cost reductions.

TABLE I
MT/NP ALGORITHM FOR MVA_{reg}.

$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxiter_{samp}$	50	250	500
$\mathcal{R}$ (%)	21.83	23.24	23.94
$maxiter_{alg} = 50$; $maxiter_{samp} = 250$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxmove_{init}$	50	100	500
$\mathcal{R}$ (%)	20.42	23.24	25.35
$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $maxiter_{samp} = 250$; $size_{list} = 250$.			
pos_{demo}	3	10	50
$\mathcal{R}$ (%)	22.53	23.24	22.53

Similar conclusions also arise by observing the results obtained using the MVA_{rand} problem instance, as shown in

Table II. However, in this case, relative cost reductions are higher and differences are more subtle for distinct parameter settings.

TABLE II
MT/NP ALGORITHM FOR MVA_{rand}.

$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $size_{list} = 50$; $pos_{demo} = 10$.			
$maxiter_{samp}$	50	250	500
$\mathcal{R}$ (%)	37.39	37.07	37.17
$maxiter_{alg} = 50$; $maxiter_{samp} = 250$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxmove_{init}$	50	100	500
$\mathcal{R}$ (%)	36.81	37.07	37.95
$maxiter_{alg} = 50$; $maxiter_{samp} = 250$; $maxmove_{init} = 100$; $size_{list} = 250$.			
pos_{demo}	10	50	100
$\mathcal{R}$ (%)	37.07	37.13	37.04

In Table III, we observe the results obtained for the MVA_{reg} problem instance when running the algorithm sequentially (with a single processor). The results are inferior for all parameter values. These results are not comparable to those obtained with tabu search, showing the improvement achieved with our parallel implementation. It is worth noting that, for the sequential version, differences in the relative cost reduction obtained with the various parameter sets are even more significant.

TABLE III
SEQUENTIAL MT/NP ALGORITHM FOR MVA_{reg}.

$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxiter_{samp}$	50	250	500
$\mathcal{R}$ (%)	-7.7	5.6	12.0
$maxiter_{samp} = 250$; $maxmove_{init} = 100$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxiter_{alg}$	25	50	100
$\mathcal{R}$ (%)	19.0	5.6	19.0
$maxiter_{alg} = 50$; $maxiter_{samp} = 250$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxmove_{init}$	50	100	500
$\mathcal{R}$ (%)	4.9	5.6	21.8

We have also tested this sequential execution when the initial solution is given by the DES+MFT greedy algorithm, as in the tabu search implementation of [10]. Table IV shows the results for different values of $maxiter_{samp}$. As expected, the relative cost reduction is significantly better than what was obtained in the sequential execution with random initial solutions, but are still inferior to those obtained in the parallel executions.

TABLE IV
SEQUENTIAL MT/NP ALGORITHM WITH DES+MFT AND MVA_{reg}.

$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $size_{list} = 250$; $pos_{demo} = 10$.				
$maxiter_{samp}$	50	250	400	1000
$\mathcal{R}$ (%)	18.3	14.8	21.8	19.01

Finally, in Table V we compare both parallel strategies running with one master and four slaves. The MT/NP parallel version achieves better results, which leads us to the conclusion that the imposed randomness on the pos_{demo} values, during the sampling phase, does not determine better solution quality.

TABLE V
MT/NP AND MT/MT ALGORITHMS FOR MVA_{reg}.

$maxiter_{alg} = 50$; $maxmove_{init} = 100$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxiter_{samp}$	50	250	500
$\mathcal{R}$ (%)MT/NP	21.83	23.24	23.94
$\mathcal{R}$ (%)MT/MT	21.83	22.53	21.83
$maxiter_{alg} = 50$; $maxiter_{samp} = 250$; $size_{list} = 250$; $pos_{demo} = 10$.			
$maxmove_{init}$	50	100	500
$\mathcal{R}$ (%) MT/NP	20.42	23.24	25.35
$\mathcal{R}$ (%) MT/MT	19.7	22.53	24.6

VI. FINAL REMARKS

This paper has discussed different parallelization trends for the μO optimization metaheuristic. We have described two different parallel strategies for the application of μO to the task scheduling problem on heterogenous processors. Both strategies were implemented and tested on a network of workstations, using the C+MPI platform. The implementation parameters of the heuristic were analysed for their effect on solution quality, highlighting when possible the conditions under which the parallel algorithm is able to show a superior performance. Solution quality was measured by the makespan reduction relatively to the best greedy algorithm

reported so far (DES+MFT) [7] for this problem. We have shown that the parallel μO heuristic, starting from a random initial solution, achieves the same quality result as an earlier implementation [10] of tabu search starting from the deterministic solution provided by the DES+MFT algorithm. Our ongoing work on this subject includes the performance evaluation of the parallel μO algorithm according to different problem parameters, such as (i) the number of tasks n of the parallel application, (ii) the processing power ratio PPR, (iii) the number of processors m during the scheduling process and (iv) the serial fraction F_s of the parallel application. Moreover, parallel implementations where the master process starts from the solution provided by the DES+MFT greedy algorithm are also to be considered. Having these parallel procedures, and others, execute on a real parallel computer may also enhance our performance evaluation with numerical values relative to execution time and speedup.

REFERENCES

[1] R. CORRÊA, A. FERREIRA and S.C.S. PORTO, Selected Algorithmic Techniques for Parallel Optimization, chapter in *Handbook of Combinatorics*, 1998, 407–456.

[2] T.G. CRAINIC, M. TOULOUSE e M. GENDREAU, "Towards a Taxonomy of Parallel Tabu Search Algorithms", Research Report CRT-933, Centre de Recherche sur les Transports, Université de Montréal, 1993.

[3] M. CREUTZ, "Microcanonical Monte Carlo Simulation", *Physical Review Letters* 50 (1983), p.1411.

[4] A. LINHARES and J.R.A. TORREÃO, "Microcanonical Optimization Applied to the Traveling Salesman Problem", *Intl. Journal of Modern Physics C* 9(1) (1998), 133-146.

[5] T. MAVRIDOU, P.M. PARDALOS, L. PITSOULIS and M.G.C. RESENDE, "Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP", Proceedings of the Workshop on *Parallel Algorithms for Irregularly Structured Problems*, Lyon, France, September 4-6, 1995.

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

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

[8] S.C.S. PORTO, J.P.W. KITAJIMA, and C.C. RIBEIRO, "Perfomance Evaluation of a Parallel Tabu Search Task Scheduling Algorithm", accept for the Special Issue on High-Performance Computing for Operational Research of the *Parallel Computing* (1999).

[9] S.C.S. PORTO and C.C. RIBEIRO, " Parallel Tabu Search Message-Passing Synchronous Strategies for Task Scheduling under Precedence Constraints", *Journal of Heuristics* 1 (1996), 207–233.

[10] 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.

[11] M. REISER e S.S. LAVENBERG, "Mean Value Analysis of Closed Multichain Queueing Networks", *Journal of the Association for Computing Machinery* 27 (1980), 313–322.

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

[13] J.R.A. TORREÃO, J.C.B. LEITE, O.G. LOQUES, and A.M. BARROSO, *An Experiment with a New Heuristic for Task Scheduling in Real-Time Distributed Systems*, Technical Report RT_01-97, Applied Computing & Automation, Universidade Federal Fluminense, Niterói, Brasil, January 1997.

[14] J.R.A. TORREÃO and E. ROE, "Microcanonical Optimization Applied to Visual Processing", *Physics Letters A* 205 (1995), 377–382.