Hybrid Evolutionary Static Scheduling for Heterogeneous Systems

Cristina Boeres
Instituto de Computação,
Universidade Federal Fluminense
Niterói, RJ, Brazil
boeres@ic.uff.br

Eyder Rios
Instituto de Computação,
Universidade Federal Fluminense
Niterói, RJ, Brazil
erios@ic.uff.br

Luiz Satoru Ochi
Instituto de Computação,
Universidade Federal Fluminense
Niterói, RJ, Brazil
satoru@ic.uff.br

Abstract- The complexity of the static scheduling problem on heterogeneous resources has motivated the development of low complexity heuristics such as list scheduling. However, the greedy characteristic of such heuristics can, in many cases, generate poor results. This work proposes the integration of list scheduling heuristics with search mechanisms based on both genetic algorithms and GRASP, to efficiently schedule tasks on distributed systems. The results show that the hybrid approach is robust and can converge quickly to good quality solutions.

1 Introduction

The efficient scheduling of an application is the key issue to achieve good performance from distributed computing systems. When accurate estimated execution and communication costs can be derived accurately, sophisticate static scheduling mechanisms can be applied at compile time.

The principal objective of static task scheduling is to find a schedule with the minimal length or *makespan*, for a given application and architectural model on bounded or unbounded number of processors. Given that popular distributed systems (e.g. clusters of PCs and Computational Grids [8]) tend to be made up of a variety of resources, it is imperative to consider processor heterogeneity and communication costs between processors.

Finding the optimal schedule for a parallel application on a set of processors such that the execution time is minimised is known to be an NP-complete problem [12]. Many heuristics have been proposed for homogeneous [3, 11] and heterogeneous environments [2, 15]. In general, these heuristics lead to sub-optimal schedules, although for specific classes of application and architectural models, optimal solutions can be found [4]. The main difficulty faced by these heuristics is how to tackle the variety the application and architecture characteristics, which influence the scheduling problem.

Search approaches like genetic algorithms have previously been applied to solve NP-complete problems and in many cases, successful results have been obtained. Usually, algorithms in this class achieve very efficient solutions for problems in which various distinct characteristics are specified. A number of genetic algorithm approaches have been proposed for the task scheduling problem [5, 7, 16]. Most of them consider that either the application is composed of independent tasks (jobs) or that communication cost between processors is negligible.

In this work, we propose a hybrid genetic algorithm approach, where an application is scheduled on set of heterogeneous resources in such a way that both the schedule length and the number of necessary processors are minimized. Not only performance characteristics of the processors are considered but also the costs associated with communication between distinct processors. The hybrid approach combines mechanisms of both genetic algorithms and list scheduling heuristics. In Section 2, some important definitions and assumptions are given. Then, in Section 3 we summarise the related work from the literature. The proposed hybrid genetic scheduling heuristic is described in details in Section 4. We also propose a heuristic based on GRASP [13] and list scheduling algorithms in Section 5. A series of experiments are described and analysed in Section 6, and finally, conclusions and future work are presented in Section 7.

2 Definitions

This work deals with applications which can be represented by a directed acyclic graph (DAG) $G = (T, E)$, where the set of n vertices, T, represents the tasks of the application and the set of arcs E, the precedence relation among them. The function $w(t_i)$ denotes the amount of work associated with task $t_i \in T$ and $c(t_i, t_j)$ is the weight associated with the arc $(t_i, t_j) \in E$ representing the amount of data units to be sent from t_i to t_j. The sets of immediate predecessor and successors of t_i is given by $pred(t_i)$ and $succ(t_i)$, respectively.

We wish here to tackle the problem of statically scheduling a given application G on a distributed set of heterogeneous processors efficiently. In the architectural model used here, $P = \{p_1, \ldots, p_m\}$ is the set of m fully connected processors and $h(p_j)$ the heterogeneity factor of p_j such that the execution time of task t_i on p_j is given by $w(t_i) \times h(p_j)$. When two adjacent tasks, say, t_i and t_j, are allocated to distinct processors, the cost associated with the communication of $c(t_i, t_j)$ data units is $c(t_i, t_j) \times L$, where the latency L is the average transmission time per byte incurred on the links of the system. On the other hand, if t_i and t_j are allocated to the same processor, the communication cost associated with (t_i, t_j) is negligible. Note that t_j only starts execution when all the necessary data are available on the processor allocated to it. Therefore, the *start time* of the task t_i on processor p_j depends on the time in which the data sent by its immediate predecessors are available at p_j and when p_j can execute t_i, i.e.

$st(t_i, p_j) = \max\{free(p_j), \max_{t_k \in pred(t_i)}\{st(t_k, p_k) + w(t_k) \cdot h(p_k)) + c(t_k, t_i) \cdot L\}\}$, where $free(p_j)$ is the time in which the processor p_j is free to execute t_i and p_k is the processor allocated to t_k, immediate predecessor of t_i.

The objective of the scheduling algorithm developed in this work is to define a static schedule S so that the execution time of G or *makespan* $\mathcal{M}(S)$ is minimised, where

$$\mathcal{M}(S) = \max_{t_i \in V}\{st(t_i, proc(t_i)) + w(t_i) \cdot h(proc(t_i))\}$$

and where $proc(t_i)$ is the processor assigned to task t_i in the schedule S. Before describing the strategies under study, a number of important concepts must be defined.

The *bottom level* of a task t_i, denoted by $blevel(t_i)$, is the length of the longest path from $t_i \in T$ to a sink task (a task without successors) considering the costs associated with the input graph and the target system. In heterogeneous systems, the average computation time of a task t_i is taken into account when calculating $blevel(t_i)$ [15], i.e. $\overline{w(t_i)} = (\sum_{\forall p_j \in P} w(t_i) \cdot h(p_j))/m$ and for all t_i such that $succ(t_i) \neq \emptyset$

$$blevel(t_i) = \overline{w(t_i)} + \max_{t_j \in succ(t_i)}\{c(t_i, t_j) \cdot L + blevel(t_j)\} \quad (1)$$

given that $blevel(t_i) = \overline{w(t_i)}$ for those $t_i \in T$ with $succ(t_i) = \emptyset$. On the other hand, the *top level* of a task t_i, denoted by $tlevel(t_i)$, is the length of the longest path from a source task to t_i. Given that $tlevel(t_i) = 0$ for all $t_i \in T$ with $pred(t_i) = \emptyset$, for the remaining tasks

$$tlevel(t_i) = \max_{t_j \in pred(t_i)}\{tlevel(t_j) + \overline{w(t_i)} + c(t_j, t_i) \cdot L\} \quad (2)$$

Note that these two priorities use the concept of average computation and communication costs since they are calculated prior to the effective scheduling of the tasks. Both $blevel(\cdot)$ and $tlevel(\cdot)$ are priorities which incorporate important characteristics of the application and architecture, on the attempt to provide the order in which tasks should be scheduled by a heuristic. Moreover, for each task, the summation of both concepts may represent the critical path of G when executed on the architecture P [11].

3 Related Work

In recent years, a number of differing static scheduling strategies have been proposed for heterogeneous processors [2]. Interesting enough, amongst these strategies two aspects stand out. First, the majority of heterogeneous scheduling heuristics belong to the class of *list scheduling* algorithms [2, 15]. Algorithms in this class have been shown to be of low complexity compared to other approaches such as clustering algorithms [3] or metaheuristics like genetic algorithms and GRASP [7, 13] and thus, favour a faster decision making. Furthermore, list scheduling algorithms can easily be applied when a fixed number of processors are available. As a pitfall however, for the homogeneous processor scheduling problem the results obtained by this class of heuristics tend to be far from optimal, particularly when communication costs are higher than the average computation costs of the application.

A number of *list scheduling* algorithm has been proposed in the literature. The main differences between them are principally related to the priorities adopted. Basically, at each iteration, two phases are executed, as seen in Algorithm 1. Firstly, in the *task choice* phase (line 3), the task with the highest priority is selected. Secondly, in the *processor choice* phase (line 4), another priority is used to select the processor to which the chosen task will be assigned.

```
1 $LT = V$ ;
2 **while** $LT \neq \emptyset$ **do**
3     $v$ = **task choice** $(LT)$ ;
4     $p$ = **processor choice** $(v, P)$ ;
5     $LT = LT - \{v\}$;
6 **end**
```

Algorithm 1: A List Scheduling Framework

The *Heterogeneous Earliest Finish Time* (HEFT) is considered one of the best algorithms for scheduling tasks onto heterogeneous processors [15]. Initially, HEFT calculates the *blevel* (Equation 1) of all of the tasks prior to making scheduling decisions. HEFT orders all the tasks in G in the decreasing order of their *blevel*, with ties being broken at random. During the scheduling phase, the algorithm selects the unscheduled task t_i with highest priority and looks for the earliest time that t_i can start on each processor p_j, *inserting* t_i in idle periods between two previously scheduled tasks, if possible. Finally, the processor chosen is the one where the task t_i can finish the earliest.

The *Earliest Time First* (ETF) algorithm is a very well-known list scheduler [10] which computes the earliest time of all *ready* tasks on the set of idle processors at each iteration. The task scheduled is given by the task-processor pair (t_i, p_j) that starts the earliest, i.e. task t_i is allocated to processor p_j. Note that, ETF is an algorithm designed for scheduling DAGs onto a limited set of homogeneous processors. We adapted ETF to heterogeneous processors, where the finish times of the tasks are evaluated.

Among the metaheuristiscs, more distinctively the *Genetic Algorithms* (GAs) are being shown to produce efficient solutions for combinatorial problems. The genetic algorithm simulates the natural evolution process by generating a population (usually using a randomised heuristic) that represents a set of solutions for a given problem, and gradually, by using genetic operators, like crossover and mutation, evolves to a better region in the solution space. Usually, solutions to the optimisation problem are specified as a set of chromosomes, called a population. In [16], for the static scheduling problem, each chromosome, representing a feasible schedule, is composed of m strings (m being the number of available processors), where for each processor p_i there is a string which represents the tasks scheduled to p_i. The initial population is generated randomly, respecting the precedence relation specified in G. Also, solutions can be generated by a level-scheduling algorithm [16]. The selection of chromosomes is based on the roulette wheel scheme and the crossover operator is implemented by cutting pairs of chromosomes randomly.

The *Problem Space Genetic Algorithm* (PSGA) also schedules tasks on a set of heterogeneous processors [7]. However, rather than representing a schedule, each gene of the chromosome specifies a priority associated with each task of the input DAG. Then, a trivial *list scheduling* algorithm is used to schedule the tasks onto the processors based on the priorities specified by the chromosome. The only list scheduling heuristic used was the *Earliest Finish Time* (EFT) [7].

4 The Hybrid Static Task Scheduling

Growing interested by the scheduling community has lead researchers to develop list scheduling heuristics for heterogeneous processors mainly due to the fact that these algorithms can be easily applied to a fixed number of heterogeneous processors. However, they are considered to be very greedy and may produce solutions far from the optimal one. In this manner, genetic algorithms are considered here as a technique to exploit the search space so that better solutions are achieved. Particularly, we propose a hybrid evolutionary approach for the static scheduling onto heterogeneous processors with intercommunication costs which combines concepts implemented in genetic algorithms with those from *list scheduling* [7]. The *Hybrid Task Scheduling Genetic Algorithm* (HTSGA) produces a variety of lists of priorities which are decoded by a given *list scheduling* heuristic.

Genetic Algorithms usually require long processing times to achieve efficient solutions for a variety of problems [14]. However, for the scheduling problem which considers DAGs, the task precedence reduces the number of feasible solutions and this, these algorithms require significant less processing times. In an attempt to decrease this processing time even further, we adopted the approach in which the genetic algorithm produces priority lists associated with the tasks instead of encoding the whole solution.

Decoding Heuristics: Since each chromosome code a priority list rather than a set of strings, each one being the tasks allocated to each processor, it is necessary to apply a static scheduling heuristic to produce the aimed schedule of G. We wish to evaluate the impact of using different scheduling strategies as decoding heuristics.

The HEFT heuristic is used as one of the decoding heuristic. In the original proposal, HEFT calculates the *blevel* of all of the tasks prior to making scheduling decisions. When used as a decoding heuristic, the priorities used are those defined in the given chromosome. The *Earliest Finishing Time* (EFT) is also used as a decoding heuristic, as in [7]. In this case, the heuristic schedules the *free* task t_i with highest priority specified in the chromosome on the processor p_j that minimises its finishing time. We also adapted the ETF algorithm (the version implemented in this work for heterogeneous processors) to be used as a decoding heuristic. In this case, the task choice is also performed using the priorities specified in the chromosomes.

As a matter of notation, let H be the set of decoding heuristics used in the hybrid approach. Thus, $H = \{HEFT_{GA}, EFT_{GA}, ETF_{GA}\}$ where $HEFT_{GA}$ and ETF_{GA} are HEFT and ETF adapted as decoding heuristics, respectively. Note that these heuristics were selected due to their reasonable performance, as reported in the literature. However, their performance is dependent on the class of DAG and architectural characteristics. We shall see that the application of all of the heuristic in H in different iterations leads HTSGA to be more efficient than the list schedulers by their own.

Coding of Solutions and Population Generation: In each position of the chromosome, the gene i, $1 \le i \le n$, represents the priority assigned to the task $t_i \in T$. For each population, the members are divided in $h = |H|$ groups, where the chromosomes of each group are decoded by one of the h decoding heuristics.

Let $Npop$ be the maximum number of chromosomes in the initial population, then, each gene i of a chromosome s_k, $1 \le k \le Npop$, given by $s_k(i)$ is equal to

$$\begin{cases} (k \bmod h) + 1, & \text{if } i = 0 \ \wedge \ 1 \le k \le Npop \\ blevel(t_i), & \text{if } i > 0 \ \wedge \ k = 1 \\ blevel(t_i) + random(-\frac{tlevel(t_i)}{2}, \frac{tlevel(t_i)}{2}), & \\ & \text{if } i > 0 \ \wedge \ 1 < k \le Npop \end{cases} \tag{3}$$

where $blevel(t_i)$ and $tlevel(t_i)$ are the priorities defined in Equations 1 and 2, respectively. Note that $s_k(0)$ assures that the chromosomes of each one of the h groups are decoded by the same heuristic. The data in $s_k(0)$ represents the heuristic in H used to generate the schedule associated with a chromosome s_k considering the priorities $s_k(i)$, $1 \le i \le |T|$.

Fitness Evaluation: Our primary objective in this work is to find a schedule of a DAG G with the minimal makespan. However, a second objective but no less important is the minimization of the number of processors required to execute the given schedule. In distributed systems like computational grids, resources are shared and therefore, allocating the smallest number of resources without harming the application's execution time will improve the overall performance. The fitness of a chromosome s_k, to be minimised, considers both the makespan of the associated schedule and the number of necessary processors:

$$\begin{aligned} fitness(s_k) = \ & mspan(decod(s_k)) + \\ & np(decod(s_k)) \times 10^{-(\lfloor \log_{10} m \rfloor + 1)} \end{aligned} \tag{4}$$

where $decod(s_k)$ is the schedule produced by the decoding heuristic $s_k(0)$, $np(\cdot)$, the respective number of processors to execute G in $mspan(\cdot)$ time units (recall that m is the total number of available processors). Obviously, the objective of HTSGA is to minimize the function $fitness(\cdot)$, so that a schedule with the minimal makespan is found. If there is more than one solution with the same minimal value, the one with the smallest number of processors is chosen.

Selection of chromosomes: HTSGA also implements the roulette wheel procedure where priority is given to those chromosomes with the smallest fitness value. Therefore, the probability to select a chromosome s_k in the population is

$$prob(s_k) = \frac{fitness_{max} - fitness(s_k)}{\sum_{i=1}^{Npop}(fitness_{max} - fitness(s_k)} \quad (5)$$

where $fitness_{max}$ is the maximum fitness value considering all the chromosomes in that population.

Diversification: A diversification procedure is applied when a series of generations is produced without any improvement to the best solution found so far. The diversification is an opportunity for searching a better result in a region of the solution space not yet explored. For doing so, new priorities must be associated with each task t_i, which in turn, are functions of the $blevel$ and $tlevel$. The diversification is implemented based on a given priority $prior_j$ and a given perturbation value $pert_j$, such that, a new chromosome s_k is given by:

$$s_k(i) = \begin{cases} (k \bmod h) + 1 & \text{, if } i = 0 \text{ e } 1 \leq k \leq Npop \\ prior_j & \text{, if } i > 0 \text{ and } k = 1 \\ prior_j + pert_j & \text{, if } i > 0 \text{ and } 1 \leq k \leq Npop \end{cases} \quad (6)$$

where $prior_j(t_i)$ and $pert_j(t_i)$ are proposed in Table 1.

j	$prior_j(t_i)$	$pert_j(t_i)$
1	$blevel(t_i)$	$rand(\frac{-tlevel(t_i)}{2}, \frac{+tlevel(t_i)}{2})$
2	$tlevel(t_i)$	$rand(\frac{-blevel(t_i)}{2}, \frac{+blevel(t_i)}{2})$
3	$blevel(t_i) + tlevel(t_i)$	$rand(\frac{-tlevel(t_i)}{2}, \frac{+tlevel(t_i)}{2})$
4	$blevel(t_i) + tlevel(t_i)$	$rand(\frac{-blevel(t_i)}{2}, \frac{+blevel(t_i)}{2})$

Table 1: Priorities generators for HTSGA.

Crossover Operators: The crossover operator chooses randomly a pair of individuals s_i and s_j from the population and generates, also randomly, two cut-off points c_1 and c_2 which define a segment in each chromosome. Then, the segment $[s_i(c_1), s_i(c_2)]$ in s_i is swapped with $[s_j(c_1), s_j(c_2)]$ in chromosome s_j. The operator is executed in accordance with a given probability pr_{cross}.

Mutation Operators: The mutation is performed for a chromosome under a given probability pr_{mut}. Let s_k be the chosen chromosome. A perturbation value $pert_j$ (from Table 1) is applied, and thus

$$s_k(i) = \begin{cases} s_k(i) & \text{, if } i = 0 \\ s_k(i) + pert_j(i) & \text{, if } i > 0 \end{cases} \quad (7)$$

The HTSGA Algorithm

The HTSGA chooses the best solution of the actual population and migrates it to the next generation. The algorithm stops after a given number $Ngen$ of generations. The pseudo-code of HTSGA can be seen in Algorithm 2. Note that, if $Ndiv$ iterations are performed without any improvement to the best solution, the diversification procedures is then applied.

Let $\mathcal{S}_l$ be the set of chromosomes that belong to the population in the l^{th} generation. In line 2, the initial population $\mathcal{S}_0$ is generated based on the priority pair $(prior_1(t_i), pert_1(t_i))$, as given in Equation 3. Then, the best solution in $\mathcal{S}_0$ is captured in s^*, based on Function 4.

```
1  $nu = 0; j = 1; s^* = \emptyset;$
2  $\mathcal{S}_0 = new\_population(j, s^*); s^* = best\_sched(\mathcal{S}_0);$
3  **for** $l = 1$ **to** $Ngen$ **do**
4      $\mathcal{S}_l = \{s^*\};$
5      **for** $k = 1$ **to** $Npop/2$ **do**
6          $s_i = random(\mathcal{S}_{l-1}); s_j = random(\mathcal{S}_{l-1});$
7          $crossover(pr_{cross}, s_i, s_j, f_1, f_2);$
8          $(f_1, f_2) = mutation(pr_{mut}, j, f_1, f_2);$
9          $\mathcal{S}_l = \mathcal{S}_l \cup \{f_1, f_2\};$
10     **end**
11     $s' = best\_sched(\mathcal{S}_l);$
12     **if** $fitness(s') < fitness(s^*)$ **then**
13         $s^* = s'; nu = 0;$
14     **else**
15         $nu = nu + 1;$
16         **if** $nu == Ndiv$ **then**
17             $j = (j \bmod nPrior) + 1;$
18             $\mathcal{S}_l = new\_population(j, s^*);$
19             $s^* = best\_sched(\mathcal{S}_l); nu = 0;$
20         **end**
21     **end**
22 **end**
23 **return** $decod(s^*);$
```

Algorithm 2: HTSGA - Hybrid Task Scheduling GA

In lines 3 to 22, $Ngen$ populations are generated. In each iteration l, the best solution found in the previous generation is included in the current population $\mathcal{S}_l$ (line 4). At each iteration of the loop in line 5, a pair of chromosomes s_i and s_j are selected randomly and the crossover operator is applied under probability pr_{cross}, followed by the mutation operator (Equation 7). The two new chromosomes f_1 and f_2 are then included in the current population $\mathcal{S}_l$. In line 11, the best chromosome s' of $\mathcal{S}_l$ is found and compared with the best solution found amongst all populations (lines 12 and 13). If the best chromosome is not updated after a number $Ndiv$ of generations (line 16), new priorities (line 17) are used to generate new chromosomes (line 18) promoting diversification of the current population. Finally, HTSGA returns the schedule associated with the best chromosome s^* (line 23).

5 Scheduling with GRASP

In general lines, GRASP [13] is a multi-start iterative process where each iteration consists of two phases: a construction and a local search phase. In the construction phase, a randomised greedy function is used to build up a feasible solution. During the construction phase, the choice of the next element to be added is determined by the ordering of all candidates in a list (CL) with respect to a greedy function. Because of the size of CL, a restricted candidate list (RCL) composed with the best candidates is usually defined. The choice of the candidate in RCL is performed at random. Then, the solution generated by the construction phase is refined by a local search procedure.

We also proposed a scheduling heuristic based on

GRASP, called the *Hybrid Task Scheduling GRASP* (HTSG). During the construction phase, concepts defined in GRASP are applied on a *list scheduling* algorithm (in our case, we used “randomised” versions of HEFT, ETF and EFT) in order to overcome their greedy behaviour. Let $\mathcal{H}_k(G, P, \alpha)$ be a heuristic which schedules the DAG G in the architecture P considering a random factor α used during the *task choice* phase (line 3 of Algorithm 1). At each iteration of $\mathcal{H}_k(G, M, \alpha)$, the list RCL of ready tasks with the highest priority is given, the task choice is performed by choosing randomly a task in RCL in accordance with α.

During the local search phase, we implemented the *Topological Assignment and Scheduling Kernel* [9] (TASK), which produces a new schedule based on a given schedule S. The local search performs a *critical path* analysis of G in accordance with S. Tasks are moved between processors if the *critical path*, and consequently, the makespan is minimised. A series of movements is performed so that the best allocation of tasks is found.

We must point out that we implemented in this work a *learning mechanism* in order to adapt the best value for α. Let $Niter$ be the maximum number of iterations performed by HTSG and let L_α be a list of values assigned to α. During the first $Niter/2$ iterations, a different value of $\alpha \in L_\alpha$ is used at each GRASP iteration, and the value of α associated with the best solution is kept. In the remaining $Niter/2$ iterations, this best α value is then applied.

6 Results and Analysis

The complexity of the static scheduling problem has lead researchers to tackle the problem for specific classes of DAGs (e.g. join, forks, trees, diamond, or irregular), and *granularities*[1]. Typically, the graph topologies chosen represent specific classes of applications, while varying the granularity accounts for a variety of target systems.

In this section, we compare the makespans produced by HTSGA, HTSG (both proposed here) and PSGA. The experimental analysis in this paper is based on a suite of unit execution time, unit data transfer graphs (UET-UDT) DAGs which includes both regular (various sizes of in-trees (IN), out-trees (OUT) and diamond graphs (DI)) and irregular graphs (randomly generated (RAN)) [3]. Also, we considered the Gaussian Elimination (GE) [6] as a real world application.

In the experiments with the UET-UDT DAGs, we defined a computation factor F, so that the execution weight of any task would be F. We also varied the latency L for different experiments. In this way, we were able to define more accurately fine and coarse grained instances. We carried out a series of experiments considering the following parameter pairs (F, L) for each input DAG: $(1, 1)$, $(5, 1)$ and $(10, 1)$ which characterise coarse grained instances, and, for fine grained ones the parameters pairs were $(1, 5)$ and $(1, 10)$. Since the GE’s are coarse grained, only the pairs $(1, 1)$, $(1, 5)$ and $(1, 10)$ were evaluated.

In this initial work, we assumed that a homogeneous network, which can be viewed as modeling a collection of different workstations, interconnected the heterogeneous processors or PCs interconnected via a switch. We generated a number of architecture environment parameters, in order to simulate different environments with a variety of number of available processors, and a diversity of heterogeneous factors. A total of 12 computational environments were used as the target architecture to execute each DAG, considering also the parameter pair (F, L).

The parameters of the genetic algorithms compared here, HTSGA and PSGA, were set to: $Ngen = 120$, $Npop = 20$, $pr_{cross} = 80\%$ and $pr_{mut} = 5\%$. In the case of HTSG, $Niter = 120$ and $L_\alpha = \{0, 0.1, 0.15, 0.2, 0.4, 0.80, 1.0\}$. Also, we executed the original list scheduling algorithms HEFT and ETF for the sake of comparison.

We performed a total of 2952 experiments, which only part of them are reproduced here in tables due to the lack of space. In Table 2, the columns 2 and 3 shows the number of times and percentage of the cases that HTSGA produced worse ($>$), equal ($=$) and better ($<$) schedules than PSGA and HTSG, respectively, for each class of DAG, considering all pairs (F, L) and architectures. In the column 4, the same is reported but for HTSG being compared with PSGA.

The Table 2 shows that HTSGA outperforms PSGA in all of the cases. Almost the same can be said about HTSG, which on average produces schedules which are better or equal than those produced by PSGA in 98.3%, 98.9%, 95.3%, 99.4% and 93.3% of the cases, for DI, IN, OUT, RAN and GE, respectively.

		HTSGA				**HTSG**	
		PSGA		**HTSG**		**PSGA**	
DI	$>$	0	0,0%	29	5,4%	9	1,7%
	$=$	92	21,7%	306	56,7%	155	28,7%
	$<$	448	78,3%	205	38,0%	376	69,6%
IN	$>$	0	0,0%	66	18,3%	4	1,1%
	$=$	242	67,2%	210	58,3%	228	63,3%
	$<$	118	32,8%	84	23,3%	128	35,6%
OUT	$>$	0	0,0%	37	10,3%	17	4,7%
	$=$	249	69,2%	253	70,3%	265	73,6%
	$<$	111	30,8%	70	19,4%	78	21,7%
RAN	$>$	0	0,0%	241	19,1%	7	0,6%
	$=$	811	64,4%	799	63,4%	678	53,8%
	$<$	449	35,6%	220	17,5%	575	45,6%
GE	$>$	0	0,0%	24	5,6%	29	6,7%
	$=$	332	76,9%	318	73,6%	324	75,0%
	$<$	100	23,1%	90	20,8%	79	18,3%

Table 2: Performance comparison between HTSGA, HTSG and PSGA in terms of worse ($>$), same ($=$) and better ($<$) makespans for each class of DAGs

The robustness of HSTGA is due to the fact that a series of distinct decoding heuristics were applied, together with the diversification procedure, which produced a new population and therefore, new priorities associated with tasks. The diversification procedure offered an opportunity for HTSGA to define a different order in which tasks are selected at the *task choice* phase of their decoding heuristics. For HTSG, which practically outperforms PSGA in most of

[1]Although there exists a number of both formal and mathematical definitions for the granularity of a DAG [11], loosely speaking it is considered to be the relationship or ratio between the amount of communication and computation.

the cases, the analysis on the results showed that both the local search phase and the use of different randomized *list scheduling* algorithms led to better solutions than PSGA.

	HTSGA			**HTSG**		
(F,L)	**ETF**	**HEFT**	**PSGA**	**ETF**	**HEFT**	**PSGA**
DI						
(10,1)	6,3%	1,5%	0,5%	6,2%	1,3%	0,3%
(5,1)	6,1%	4,1%	1,6%	5,9%	3,8%	1,3%
(1,1)	0,6%	21,3%	11,8%	0,8%	21,5%	12,1%
(1,5)	22,6%	32,5%	10,1%	19,6%	29,8%	6,6%
(1,10)	27,4%	32,6%	14,0%	22,2%	27,7%	7,8%
Avg.	**12,6%**	**18,4%**	**7,6%**	**10,9%**	**16,8%**	**5,6%**
IN						
(10,1)	12,0%	1,4%	0,1%	12,2%	1,6%	0,3%
(5,1)	11,6%	1,3%	0,3%	11,7%	1,4%	0,4%
(1,1)	4,4%	3,8%	1,1%	4,1%	3,4%	0,8%
(1,5)	15,1%	8,0%	4,6%	22,0%	14,9%	11,4%
(1,10)	18,2%	8,9%	6,0%	28,4%	18,9%	16,0%
Avg.	**12,3%**	**4,7%**	**2,4%**	**15,7%**	**8,0%**	**5,8%**
OUT						
(10,1)	2,1%	2,1%	0,3%	2,2%	2,3%	0,5%
(5,1)	2,7%	2,5%	0,4%	2,9%	2,7%	0,6%
(1,1)	2,1%	5,1%	0,4%	2,8%	5,8%	1,2%
(1,5)	19,4%	26,2%	1,7%	18,1%	25,0%	0,1%
(1,10)	26,3%	28,8%	2,7%	23,8%	26,4%	-0,8%
Avg.	**10,5%**	**13,0%**	**1,1%**	**10,0%**	**12,4%**	**0,3%**
RAN						
(10,1)	21,0%	0,9%	0,1%	21,1%	1,0%	0,2%
(5,1)	21,8%	1,2%	0,1%	21,9%	1,3%	0,2%
(1,1)	26,9%	17,8%	1,0%	28,4%	19,3%	3,0%
(1,5)	22,1%	17,4%	4,7%	22,2%	17,4%	4,7%
(1,10)	17,4%	9,0%	6,9%	16,6%	8,0%	5,9%
Avg.	**21,8%**	**9,2%**	**2,5%**	**22,0%**	**9,4%**	**2,8%**
GE						
(1,1)	18,5%	0,1%	0,4%	18,5%	0,1%	0,4%
(1,5)	22,8%	0,8%	0,4%	22,9%	0,8%	0,5%
(1,10)	22,5%	4,3%	0,7%	22,3%	4,0%	0,4%
Avg.	**21,3%**	**1,7%**	**0,5%**	**21,2%**	**1,6%**	**0,4%**

Table 3: Improvement on the makespans

HTSGA and HTSG were never worse than both the original HEFT and ETF, what was expected, since these two heuristics were incorporated to them, somehow. HTSGA used in one of the iterations the same priorities applied by HEFT and ETF, and in the case of HTSG, when $\alpha = 0$, HEFT$_{SG}$ and ETF$_{SG}$ had similar behaviour to their original counterpart versions.

One observation that we must point out is that in the case of fine grained DAGs, the decoding heuristic that produced most of the best results was HEFT$_{GA}$ (in HTSGA) and HEFT$_{SG}$ (in HTSG). This is due to the mechanism of inserting tasks in idle periods of time on the processors [15]. When scheduling coarse-grained instances, the respective versions of ETF on the other hand, produced most of the best results. Studies in the literature showed that ETF leads to good results when communication costs are not very high.

For fine grained in-trees, HTSG is slightly better than HTSGA due to the TASK local search procedure, since it clusters independent tasks with common immediate successors on the same processor, which is very advantageous when communication costs are high. HTSG thus, relieves the burden caused by the *list scheduling* approaches. This list-scheduling algorithm greedily allocated independent tasks to distinct processors since they give priority to the minimisation of each individual finish time rather than the minimisation of the makespan of the whole application.

We observed that in only 17 cases considering out-trees HTSG loses to PSGA. Although PSGA used only one decoding heuristic (EFT$_{GA}$) and has no diversification procedure, it produced more solutions than HTSG. The same can be observed when comparing HTSGA with HTSG for all the DAGs, but not for the random graphs. Although HTSGA deals with a bigger population of solutions, for this class of DAGs, HTSG produced solutions with makespan slightly smaller than those generated by the former, on average. Analysing more closely, we concluded that the advantage of HTSG was due to the local search TASK.

The Table 3 shows the improvements to the makespan when comparing HTSGA (columns 2 to 4) and HTSG (the last three columns) with the original list scheduling algorithms HEFT and ETF, and the genetic algorithm PSGA. Nicely, both of our proposed algorithms produce schedules with much smaller makespan mainly for fine grained instances ($(F, L) = (1, 5), (1, 10)$). These experiments shows the gains that the evolutionary approaches offers, mainly when we consider heterogeneous processors and high communication costs.

6.1 Empirical Probabilistic Analysis

We performed an empirical probabilistic analysis, as proposed in [1]. We evaluated the probability of the algorithms implemented in this work to achieve a given solution on a given time. A number of representative instances were chosen, considering a target architecture with no more than six heterogeneous processors.

Initially, each one of the algorithms PSGA, HTSGA and HTSG were executed 200 times in order to generate the target solutions. These solutions were classified as: the *easy target*, which is the arithmetic mean of the worse makespan of the schedules generated by all of the three heuristics; and the *difficult target* which obviously considers the makespan of the best schedules produced by the same set of heuristics.

Having the *easy* and *difficult* targets defined, PSGA, HTSGA and HTSG were again executed 200 times such that each execution of the algorithm stops whenever it achieves a solution with makespan smaller or equal then the given target, having the respective processing time recorded. Then, the list $K_h = \langle k_1, k_2, \ldots, k_{200}\rangle$ in non-decreasing order of processing times taken by each heuristic $h \in \{PSGA, HTSGA, HTSG\}$ was constructed and for each $k_i \in K_h$, an empirical probability $pr_h(k_i) = (i - \frac{1}{2})/200$ was calculated. Finally, we plot the three curves considering the pair $(k_i, pr_h(k_i))$, each curve corresponding to each h. All the plots can be viewed in Figure 1, for each instance, considering the ease and difficult targets.

For the diamond DAG, HTSGA and HTSG converged to the easy target in less than $10ms$, 100% of the time, while $PSGA$, only around 1%. The difficult target was achieved by both HTSGA and HTSG 100% of the times in less than $100ms$ although, in this case, HTSGA may be faster. PSGA however, reached it very few times.

The robustness of HTSGA and HTSG is one of the reasons for this performance. Although HTSG achieved the target less times than HTSGA, it converged very quickly due to its local search procedure, mainly for in-trees. HTSGA actually benefits from it diversification procedure.

The instance in which HTSG did not have a very good performance when compared with HTSGA and PSGA was out-trees. The probabilities to achieve the easy target in $10ms$ were 100%, 85.5% and 17.5% for HTSGA, PSGA and HTSG, respectively. This may be explained that the two genetic approaches produces more solutions than HTSG, leading to a better performance. A similar behaviour was observed for the difficult target, although PSGA executed more iterations to converge.

For the random graphs, both HTSG and HTSGA reached the easy target much faster than PSGA, 100% of the times. They needed no more than $6ms$ while PSGA, a total of $917ms$. Their nice performance is justified by the application of HEFT. For the difficult target, PSGA never reached it, while HTSGA and HTSG always converged.

7 Conclusion and Future Work

This paper presented two hybrid metaheuristics to solve the static task scheduling problem for heterogeneous environment. The combination of fast traditional list schedulers with time consuming genetic algorithms, provided a more efficient and practical way to schedule general DAGs on heterogeneous resources. The use of a variety of priority lists offered the opportunity for better decision making for the list scheduling heuristics. HTSGA produced much better results than the traditional list schedulers mainly when communication costs between processors were high. We also presented a novel hybrid GRASP approach that also produced efficient results, losing only to HTSGA, on average, but not to the other heuristics being compared here. As future work, HTSGA will be applied to other real applications executed on computational grids. We will also invest in HTSG, analyzing other mechanisms for local searches as well as investigate the benefits of implementing the path relink mechanism [14].

Acknowledgments

The authors are also partially funded by research grants from CNPq.

Bibliography

[1] R. M. Aiex, M.G.C. Resende, and C. Ribeiro. Probability distribution of solution time in GRASP: An experimental investigation. *Journal of Heuristics*, 8(3):343–373, 2002.

[2] O. Beaumont, A. Legrand, and Y. Robert. Static scheduling strategies for heterogeneous systems. *Computing and Informatics*, 21:413–430, 2002.

[3] C. Boeres and V.E.F. Rebello. On solving the static task scheduling problem for real machines. In M. Fiallos R. Correa, I. Dutra and F. Gomes, eds, *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications*, chapter 3, pp 53–84. Kluwer, 2002.

[4] C Boeres and V.E.F. Rebello. Towards optimal task scheduling for realistic machine models: Theory and practice. *The International Journal of High Performance Applications*, 17(2):173–190, 2003.

[5] R. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, 1999.

[6] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. Int. Thomson Computer Press, 1995.

[7] M. Dhodhi, I. Ahmad, A. Yatama, and I. Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 62:1338–1361, 2002.

[8] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. 2nd edition. Morgan Kaufmann, 2004.

[9] M-Y Hu, W. Shu, and J. Gu. Efficient local search for DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):617–627, June 2001.

[10] J-J. Hwang, Y-C. Chow, F.D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[11] Y-K Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), Dec. 1999.

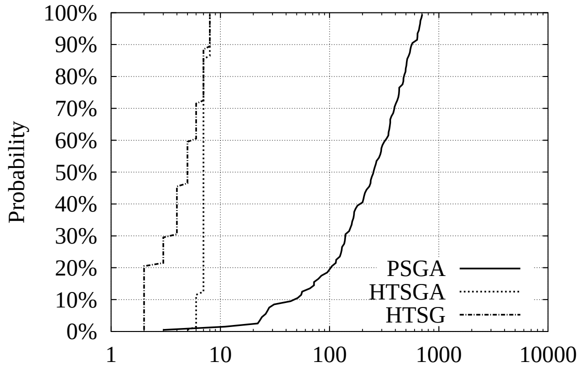
[12] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19:322–328, 1990.

[13] M.G.C. Resende and C. Ribeiro. Greedy randomized adpative search procedures. In F. Glover and G. Kochenberger, editors, *Handobook of Metaheuristics*, pp 219–249. Kluwer, 2003.

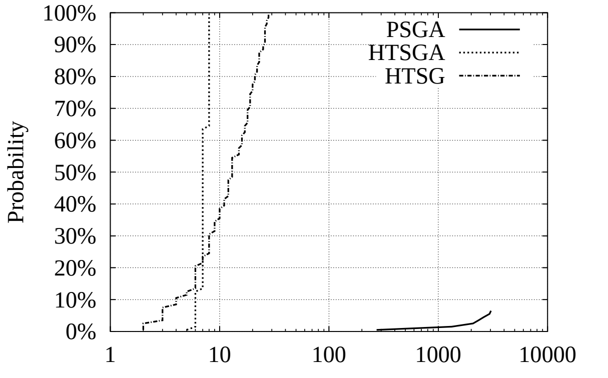
[14] M.G.C. Resende and J.P. Souza, editors. *Metaheuristics: Computer Decision-Making*. Kluwer, 2003.

[15] H. Topcuoglu, S. Hariri, and M.Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.

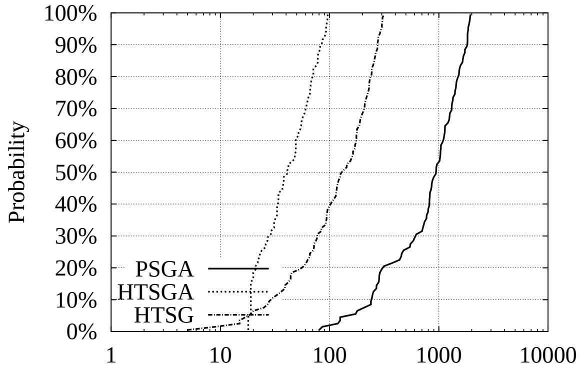
[16] L. Wang, H. J. Siegel, V.R. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel Distributed Computing*, 47(1):8–22, 1997.



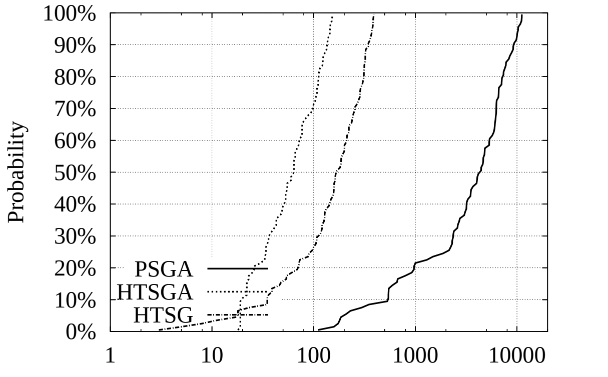
(a) DI with 256 tasks, easy target: 177



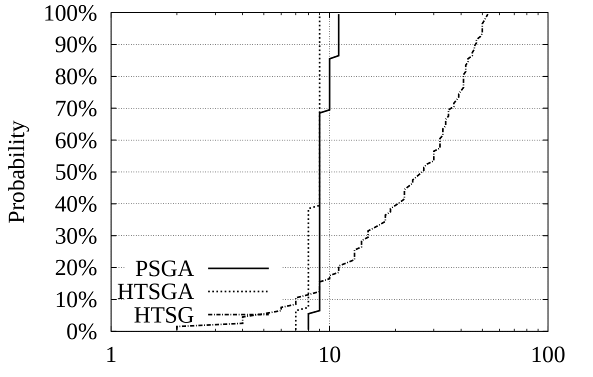
(b) DI with 256 tasks, difficult target: 159



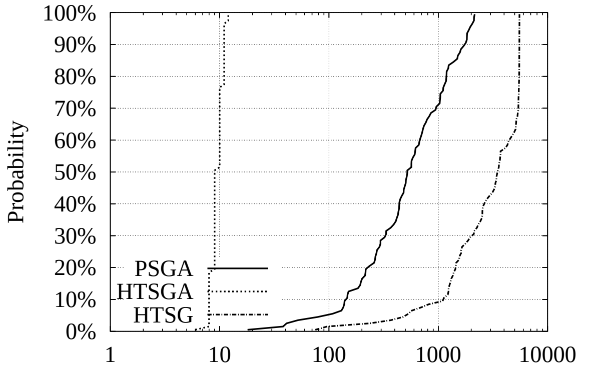
(c) IN with 255 tasks, easy target: 85



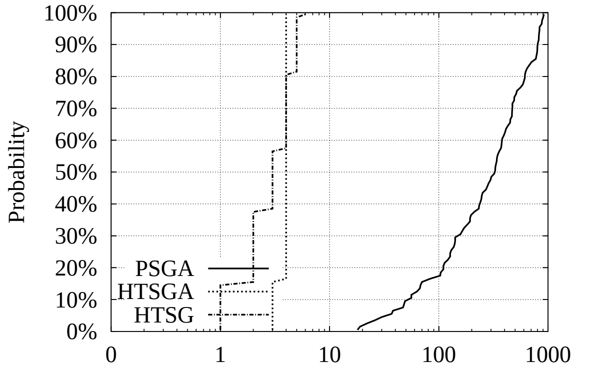
(d) IN with 255 tasks, difficult target: 79



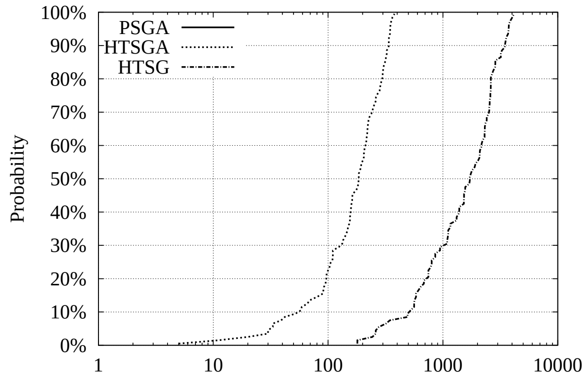
(e) OUT with 255 tasks, easy target: 79



(f) OUT with 255 tasks, difficult target: 74



(g) RAN with 140 tasks, easy target: 64



(h) RAN with 140 tasks, difficult target: 53

Figure 1: Probabilistic Analysis comparing HTSGA, HTSG and PSGA, considering easy and difficult targets