Hybrid Task Scheduling: Integrating Static and Dynamic Heuristics*

Cristina Boeres† Alexandre Lima
Vinod E. F. Rebello†
Instituto de Computação, Universidade Federal Fluminense (UFF)
Niterói, RJ, Brazil
{boeres,alima,vinod}@ic.uff.br

Abstract

Researchers are constantly looking for ways to improve the execution time of parallel applications on distributed systems. Although compile-time static scheduling heuristics employ complex mechanisms, the quality of their schedules are handicapped by estimated run-time costs. On the other hand, while dynamic schedulers use actual run-time costs, they have to be of low complexity in order to reduce the scheduling overhead. This paper investigates the viability of integrating these two approaches into a hybrid scheduling framework. The relationship between static schedulers, dynamic heuristics and scheduling events *are examined. The results show that a hybrid scheduler can indeed improve the schedules produced by good traditional static list scheduling algorithms.*

1. Introduction

Efficient task scheduling is essential to achieve good performance from parallel and distributed computing systems. Given its importance, extensive research has been carried out on the NP-complete problem of statically scheduling applications onto networks of *homogeneous* processors. Numerous high quality scheduling algorithms have been proposed and various optimality results found for specific classes of applications and system parameters [6]. Crucial to the applicability of these heuristics is the availability of precise information relating to characteristics of both the application and target architecture (i.e. computation and communication costs).

Growing interest is currently being paid to *Computational Grids* [8], due in part to the fact that they make computing power available to users who themselves have insufficient resources locally to resolve or execute their applications. The key to harnessing the Grid's potential effectively depends on the ability of scheduling strategies to consider that the resources available in these systems are typically both *heterogeneous* and *shared*.

The problem of scheduling an application onto a set of *heterogeneous* resources considering inter-processor communication is also NP-complete [7]. In this problem, processing rates of individual processors and communications delays between pairs of processors may differ. In recent years, a number of differing static scheduling strategies have been proposed for heterogeneous systems [1]. Interestingly, the majority of these heterogeneous heuristics belong to the class of *list scheduling* algorithms [1, 15]. Algorithms in this class have been shown to be of low complexity compared to other approaches such as clustering algorithms or metaheuristics and thus, favour faster decision making. Furthermore, list scheduling algorithms can easily be applied when only a limited number of processors are available. It is known, however, that 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 [2].

In shared environments, predicting the processing power and communication bandwidth available to a given application is difficult. This makes designing efficient and effective static scheduling algorithms extremely challenging. The estimations assumed by the static scheduler may no longer be the same during execution and so cause the application to perform poorly [10, 14]. To overcome this handicap, *dynamic* schedulers use information available at run-time to make their scheduling decisions. These decisions, however, need to be made quickly in order to avoid stalling the application's execution and, thus minimise the impact (i.e. the scheduling overhead) on the execution time.

By using run-time information as it becomes available, dynamic schedulers are forced to make a series of instant scheduling decisions. Unfortunately, decisions made with-

* This work is funded by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), under Process No. 552205/2002-8.
† The authors are also partially funded by research grants from CNPq.

Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)
0-7695-2046-4/03 $17.00 © 2003 IEEE

out considering future requirements can adversely affect the final execution time. On the other hand, since the impact of scheduling an application at compile time is low when compared with dynamic scheduling, more sophisticated mechanisms tend to be proposed for static problem [3], particularly for homogeneous environments.

With the objective of achieving better run-time performance, *hybrid schedulers* (i.e. schedulers which employ both a static and dynamic phase) provide the dynamic heuristic with information derived from a statically generated schedule. Although based on estimated costs, the idea behind this methodology is to provide the dynamic scheduler's decision making process with some indication of the relative importance of each task in terms of the application's overall performance.

Maheswaran and Siegel [14] showed that a hybrid strategy can be used to promote better performance when executing an application in heterogeneous systems. They presented three variants of a list scheduling-based algorithm called the *hybrid remapper* for the dynamic phase and stated that the static phase could be implemented by any static scheduling algorithm. The results presented in [14] showed that their hybrid remapper managed to improve performance in dynamic environments by as much as 15% for large randomically generated graphs when compared with the static scheduling algorithm baseline [14] (and estimated run-time values) by itself. However, the results also indicate that the performance achieved using the hybrid remapper is around *100% worse* than what could be obtained if accurate run-time information was available to the static algorithm.

This work proposes to extend the work of Maheswaran and Siegel by proposing a number of dynamic scheduling variants and investigating how the quality of the initial static schedule impacts the overall performance. This paper focuses on a worse-case scenario for a hybrid scheduler, i.e. we assume that the static scheduler has perfect knowledge of the run-time costs. In this effectively static environment, unlike the work of Maheswaran and Siegel, it will be much harder for the hybrid scheduler to improve on a statically generated schedule. We wish to verify if in fact hybrid scheduling can improve the solutions produced by good traditional static scheduling heuristics. Furthermore, considering the different static heuristics implemented here, we wish to evaluate which benefit most from dynamic scheduling.

The following section presents the definitions adopted in this work. Section 3 discusses the frequency at which a dynamic scheduler should be executed and introduces the concept of *scheduling events*. The suite of hybrid schedulers under investigation is described in Section 4. An overview of three static schedulers and details of the dynamic heuristics and the proposed scheduling event mechanisms are also given. Section 5 presents our experimental environment, the scheduling instances analysed, and a summary of the results of our initial evaluation. Finally, conclusions and on-going work are outlined in Section 6.

2. Definitions

This work considers the class of parallel applications in which each program can be represented by an directed acyclic graph (DAG). In these applications, tasks are indivisible processing units characterised by the input data they receive from, and the output data sent to, other tasks. In a *DAG* $G = (V, E, \varepsilon, \omega)$: the set of vertices, V, represent the tasks of the application; E, the precedence relation among them; $\varepsilon(v_i)$ is the amount of work associated to task $v_i \in V$; and $\omega(v_i, v_j)$ is the weight associated to the edge $(v_i, v_j) \in E$, representing the amount of data units transmitted from task v_i to v_j. The set of immediate predecessor and successor tasks of v_i is given by $pred(v_i)$ and $succ(v_i)$, respectively.

In the architectural model, $P = \{p_0, \ldots, p_{m-1}\}$ is the set of m processors and $h(p_i)$ the heterogeneity factor of processor p_i such that the execution time of task v on p_i is given by $\varepsilon(v) \times h(p_i)$. We also consider that when two adjacent tasks, say, u and v, are allocated to distinct processors p_u and p_v, respectively, the cost associated with the communication of (u, v) is $L \times \omega(u, v)$, where the latency L is the transmission time per byte incurred on each link.

Before describing the mechanisms under investigation, a number of important concepts are defined as follows. These concepts are priorities based on characteristics of the DAGs and the target architecture.

The level of a task v_i in G, denoted by $level(v_i)$, is defined as $\max_{u \in pred(v_i)}\{level(u)\} + 1$, where the level of any source task is zero. Note that, in this case, only knowledge about the topology of the graph is considered. On the other hand, the *bottom level* of a task v_i, denoted by $blevel(v_i)$, takes into account the costs associated with the input graph and the target system. In heterogeneous systems, the average computation time of a task v_i is used when calculating $blevel(v_i)$ [15], i.e.

$$\overline{\varepsilon_P(v_i)} = \Big(\sum_{\forall p_j \in P} \varepsilon(v_i) \times h(p_j)\Big)/m$$

and given that $blevel(v_i) = \overline{\varepsilon_P(v_i)}$ for all $v_i \in V$ with $succ(v_i) = \emptyset$, $blevel(v_i)$ is equal to

$$\overline{\varepsilon_P(v_i)} + \max_{w \in succ(v_i)}\{\omega(v_i, w) \times L + blevel(w)\} \qquad (1)$$

The *scheduled bottom level* of a task v_i, denoted by $blevel_s(v_i)$, is the bottom level of v_i based on the schedule generated by the static scheduler prior to the execution of the application, i.e. $blevel_s(v_i)$ is

$$\varepsilon(v_i) \times h(p_j) + \max_{w \in succ(v_i)}\{comm(v_i, w) + blevel_s(w)\}$$

where the communication cost $comm(v_i, w) = \omega(v_i, w) \times L$ if v_i and w are allocated to distinct processors, otherwise, zero.

The class of distributed schedulers generally execute a dynamic scheduler on each processor [13]. In order to obtain global information about the system load, the schedulers are required to exchange data which can be costly. The dynamic scheduler considered in this work belongs to the class of central schedulers, i.e. schedulers which are executed on one of the processors. The reason for this choice is related to requirements of *system-aware applications* and the configuration of the EasyGrid Framework [4]. Embedded into system-aware applications by the framework is a globally distributed, locally centralised dynamic scheduler, i.e. a central scheduler at each local site cooperates with those at other sites to execute the application across the grid.

When the dynamic scheduler assigns a task to a processor, the task is placed in the job queue of that processor. In order to be ready for execution, all data for the task must be available on the respective processor. We assume that interprocessor communication can only be scheduled once the receiving task has been allocated (by the dynamic scheduler) to a processor.

3. The Dynamic Scheduling Framework

Calling the dynamic scheduler too frequently can lead to poor performance particularly when the application is composed of a large number of tasks and different aspects of the target system have to be analysed. On the other hand, if it is too infrequent, the dynamic scheduler will not be able to adjust the application's execution to changes in the system.

It is common, however, to assume that the execution of the dynamic scheduler is instantaneous [14], since scheduling decisions are typically performed over a small subset of the application's tasks. Nevertheless, since the dynamic scheduler is executed whenever a *scheduling event* occurs, it is important to analyse the effects of different scheduling event mechanisms on application performance. The frequency at which they should occur is related to the level of instability of the system (e.g. changing processor loads and resource failures) and also the information about the application necessary for the scheduler to provide a good mapping. At each scheduling event, a given subset of tasks are evaluated by the dynamic scheduler and allocated to processors in accordance with a given policy. The more frequent the scheduling events, the smaller this subset will be, and thus fewer scheduling options will be available.

As a technique to implement variations in the intervals between scheduling events, the input DAG can be partitioned into *blocks* which contain tasks belonging to N adjacent levels of the DAG, for a given $N \geq 1$. At each scheduling event, all tasks within a block are re-assigned to processors. The execution of tasks in a block can be used to trigger the scheduling event of a successive block. A more formal definition will be given in Section 4.

In [14], the input DAG G is partitioned into blocks containing only tasks of the same level (i.e $N = 1$). In this paper, we propose to partition G into blocks with $N > 1$ adjacent levels. The principal aim being to allow the scheduler to evaluate, during a scheduling event, the allocation of task v_i and its predecessors before tasks independent of v_i.

4. The Hybrid Schedulers

This present work aims to investigate the benefits of including information produced by static heuristics into the decisions making of the dynamic scheduler. The impact of schedules generated by different static heuristics, new scheduling event mechanisms, and an alternative dynamic scheduling heuristic to those proposed in [14], are analysed.

4.1. The Static Schedulers

In this paper, we consider three static schedulers which are based on the list scheduling approach [12]. *Heterogeneous Earliest Finish Time* (HEFT) is considered one of the best algorithms for scheduling tasks onto heterogeneous processors [15]. Initially, HEFT calculates the *blevel* of all of the tasks prior to making scheduling decisions. HEFT orders all the tasks in G in decreasing order of their *blevel*, with ties being broken at random. During the scheduling phase, the algorithm selects the unscheduled task v_i with highest priority and looks for the earliest time that v_i can start on each processor p_j, *inserting* v_i in idle periods between two previously scheduled tasks if possible. Finally, the processor chosen is the one where the task v_i can finish the earliest.

The *Dynamic Critical Path* (DCP) algorithm [11] employs a dynamic priority (i.e. the priority is recalculated after every iteration of the scheduling process) based on the critical path of the input DAG. The start time of the tasks are fixed only when all tasks of G have been scheduled. The choice of processor to which task v_i is assigned is based on predicting the potential start time of the most important immediate successor of v_i on that processor. Only the processors that have been assigned tasks which communicate with v_i, plus an used one, are analysed. DCP was designed for the problem of scheduling tasks on an unbounded number of homogeneous processors and has the best overall performance of all list scheduling algorithms [12].

The *Earliest Time First* (ETF) algorithm is a very well-known list scheduling algorithm [9] which computes the earliest start time for each *ready* task on the set of idle processors at each iteration. The task scheduled is given by the task-processor pair (v_i, p_j) that starts the earliest, i.e. task v_i

is allocated to processor p_j. Note that ETF considers a limited number of homogeneous processors.

4.2. The Dynamic Schedulers

Dynamic scheduler can be distinguished by the mechanisms employed to trigger the scheduling events and the priorities used to schedule tasks during each event. The dynamic schedulers implemented here utilize (to various degrees) information from a schedule generated *a priori* by a static scheduler. This information is applied at each scheduling event, when a subset of tasks in V will be *reallocated* to the available processors. Each scheduling event consists of two phases: the *task choice* phase, when a task is selected in accordance with a given priority from the list of tasks to be re-allocated during the current scheduling event; and the *processor choice* phase, when another priority is used to select the processor to which the chosen task will be assigned. These two phases are repeated until all of the tasks in the subset have been re-allocated to processors.

Next, we describe the four pairs (*task choice* and *processor choice*) of priorities considered in this evaluation. Note that the first three pairs are based on schedulers implemented in [14]. Also described are a number of variations on the definition of *blocks* of tasks to be reallocated during a scheduling event. These variations will specify different frequencies at which the scheduling events occur and are applied to each one of the four dynamic priority pairs.

During the *task choice* phase, the *Minimal Partial Completion Time Static Priority* (PS) selects the next task v_i with the highest $blevel_s(v_i)$ (scheduled bottom level). Then, during the *processor choice* phase, PS re-allocates v_i to a processor p_j that minimises the finish time of v_i. Let $idle(p_j)$ be the time that p_j becomes idle. Therefore, $\forall p \in P$, task v_i is allocated to the processor p_j that minimises

$$ft(v_i, p) = \max\{idle(p), \max_{u \in pred(v_i)}\{ft(u, p_k) + comm(u, v_i)\}\} + \varepsilon(v_i) \times h(p) \quad (2)$$

where $ft(u, p_k)$ is the finish time of task u already re-allocated to processor p_k either during a previous scheduling event or the current one. Note therefore that, this time maybe be the *real* finishing time of task u, if it has already been executed on p_k, or the *estimated* time if it has been re-allocated to p_k but waiting for execution or currently being executed.

The *Minimal Completion Time Static Priority* (CS) also selects task v_i with the highest $blevel_s(v_i)$. However, during the *processor choice* phase, task v_i is mapped to the processor p_j that minimises the critical path through v_i, denoted by $cp(v_i, p_j)$. Formally, v_i is allocated to p_j, if

$$cp(v_i, p_j) = ft(v_i, p_j) + \max_{w \in succ(v_i)}\{comm(v_i, w)\} + blevel_s(w)$$

is the minimal for all processors P ($ft(v_i, p_j)$ is defined as in Equation 2).

The third dynamic scheduling policy implemented is the so-called *Minimum Completion Time Dynamic Priority* (CD) Algorithm, where during the *task choice* phase, the task v_i with the highest critical path $cp(v_i, p_j)$ is selected (p_j is the processor to which v_i was allocated by the static scheduler). Note that the algorithm is called dynamic because it considers costs based on the processor allocation of the predecessors of v_i made by CD itself during previous scheduling events. The *processor choice* phase implements the same priority as the CS Algorithm.

As an alternative, we propose the *Minimal Completion Time Bottom Level Graph Priority* (CB) where during each *task choice* phase the task v_i with the highest $blevel(v_i)$ is selected (recall that it is not the scheduled bottom level, but the one defined in Equation 1). The *processor choice* phase is the same as in CS Algorithm.

The subset of tasks to be re-allocated

One of the crucial issues to be tackled when scheduling a DAG is the order in which tasks are chosen to be assigned to a processor. The main conclusions found in the literature have lead to the use of priorities for choosing tasks in static heuristics that are re-calculated at each iteration of the scheduling process (the so-called dynamic priorities) [12]. These priorities are related to the costs which impact the execution of the application on the target system.

In dynamic scheduling, the priority used to define the allocation order of the tasks is also a key issue. However, depending on the mechanisms used to implement the scheduling events, more critical tasks (considering the characteristics of the application and the target system) may be scheduled during later scheduling events, thus effectively giving higher priority to less important tasks.

In this work, we wish to extend the definition of blocks of tasks to be scheduled proposed in [14] so that each block includes tasks from more than one adjacent level. However, the number of levels considered must be limited so that the re-allocation of tasks is based more on accurate run-time information than statically estimated values. The following variants are defined:

V1.1 Each block contains all the tasks of a given level as proposed in [14]. The scheduling event for block k occurs when the first task of block $k-1$ has completed its execution;

V1.N Each block contains the tasks from N adjacent levels, and again the scheduling event for block k occurs when the first task of block $k-1$ has completed its execution. In this paper, we consider $N = 2, 3$;

V2.N Block zero has the tasks from level 0 and the remaining blocks each contain tasks from N distinct adjacent levels. The scheduling event for block k occurs when the first task of the last level of block $k-1$ has finished its execution. Again, we consider $N = 2, 3$.

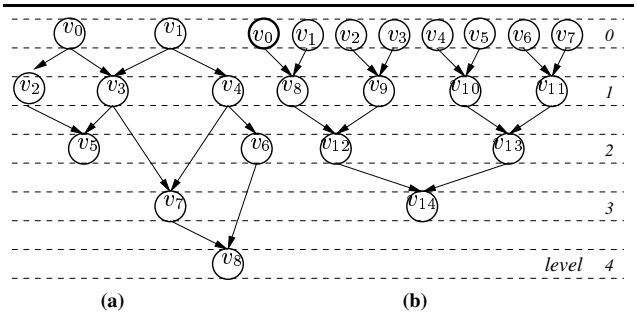

Figure 1. (a) A DAG with five levels. (b) An in-tree DAG with four levels.

Let us assume $B(k)$ denotes block k and use the example DAG in Figure 1(a) to illustrate the differences between these variants. In the case of the variant **V1.1**, five blocks are defined as $B(0) = \{v_0, v_1\}$, $B(1) = \{v_2, v_3, v_4\}$, and so on. In this case, the scheduling event which reallocates the tasks in block $B(1)$ occurs when the first task in $B(0)$, say, v_0, completes its execution. In the case of variant **V1.2**, the blocks are $B(0) = \{v_0, v_1, v_2, v_3, v_4\}$, $B(1) = \{v_5, v_6, v_7\}$ and $B(2) = \{v_8\}$. Tasks in $B(1)$ are reallocated when the first task in $B(0)$, (for example, v_0) finishes its execution. On the other hand, in the case of **V2.2**, where the blocks are $B(0) = \{v_0, v_1\}$, $B(1) = \{v_2, v_3, v_4, v_5, v_6\}$ and $B(2) = \{v_7, v_8\}$, tasks in $B(2)$, for example, are only reallocated when the first task from level two completes its execution (either v_5 or v_6).

As proposed in [14], the tasks in block $B(0)$ are not reallocated by the given dynamic scheduler, but executed on the processors allocated by the static scheduler. The thinking behind this mechanism is that, for the variants **V1.1** and **V2.N**, $B(0)$ contains only source tasks and therefore, almost no information about these tasks will change before the execution of the application.

For consistency, we also apply this definition to the variants **V1.2** and **V1.3** even though block $B(0)$ contains non-source tasks. However, in order to provide an opportunity for the dynamic scheduler to improve the task allocation produced by the static scheduler, we also propose alternatives for each of the variants **V1.N** which implements the re-allocation of block $B(0)$. Thus, before executing any task of the application, the dynamic scheduler immediately comes to action to reallocate the tasks in $B(0)$. We shall denote these new variants as **V1.Nd**. In this paper, we consider $N = 1, 2, 3$.

5. Experiments and Performance Analysis

The main objective of this work is to investigate the performance improvements provided by hybrid scheduling algorithms with respect to two characteristics which play an important role in re-allocation decisions: scheduling information produced by three well known static heuristics, and; the set of tasks analysed during each scheduling event (which in turn influences the frequency of scheduling events).

Our objective is to tackle the problem of efficiently executing an application on a set of heterogeneous resources. However, in order to have a better understanding of the interactions between the various mechanisms implemented, in this paper we consider the following simplifications: all the processors are homogeneous (where $h(p_i) = 1,\ \forall p_i \in P$) and the tasks computation costs considered by the static schedulers match their actual execution times.

Rather than taking into account all of the complexities of a real environment, such as software overheads, network topologies, processors loads etc., our study is based on a simpler environment simulated by SimGrid [5]. The objective of this package is to create an environment which simulates the execution of distributed applications in grids and to provide a means to evaluate both static and dynamic scheduling algorithms.

The complexity of the scheduling problem has lead researchers to attempt to tackle the problem for specific classes of DAGs, be it in accordance with the topology (e.g. join, forks, trees, diamond, or irregular) and/or with the *granularity*[1]. Typically, the graph topologies chosen represent specific classes of applications, while varying the granularity can also account for a variety of target systems. Adopting such an approach not only allows theoretical results to be obtained (e.g. optimality bounds) for specific problem instances but also identifies the parameter space for which experimental analysis of proposed mechanisms should focus. The 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 trees and diamond graphs) and irregular graphs (randomly generated and graphs taken from the literature) [3]. Also considered are the Peer Set Graphs (PSG), irregular DAGs with arbitrary costs, taken from the benchmark suite proposed by Kwok and Ahmad in [12].

1 Although there exists a number of both formal and mathematical definitions for the granularity of a DAG [12], loosely speaking it is considered to be the relationship or ratio between the amount of computation and communication.

	PS			CS			CD			CB		
	V1.1	V1.2	V1.3	V1.1	V1.2	V1.3	V1.1	V1.2	V1.3	V1.1	V1.2	V1.3
HEFT	-0.4	2.0	1.9	1.3	1.8	2.5	0.7	1.5	3.5	1.0	2.1	1.6
DCP	-4.1	-1.9	-1.4	-0.2	-0.1	0.7	-2.9	-4.0	-3.1	-0.2	0.3	1.2
ETF	-4.5	-2.5	-1.0	-1.9	-0.8	0.0	-5.3	-3.9	-4.1	-3.3	-2.0	-1.5
Average	-3.0	-0.8	-0.2	-0.3	0.3	1.1	-2.5	-2.1	-1.2	-0.8	0.2	0.4

	PS			CS			CD			CB		
	V1.1d	V1.2d	V1.3d	V1.1d	V1.2d	V1.3d	V1.1d	V1.2d	V1.3d	V1.1d	V1.2d	V1.3d
HEFT	-0.4	1.8	2.2	1.0	2.7	2.6	0.8	3.0	3.0	0.7	3.1	3.0
DCP	-3.7	-1.6	-0.6	-0.1	0.9	0.6	-2.8	-2.7	-2.4	-0.2	1.7	2.7
ETF	-4.5	-2.9	-1.0	-2.3	-0.2	0.0	-5.4	-2.2	-3.7	-3.7	-1.0	-0.5
Average	-2.9	-0.9	0.2	-0.5	1.1	1.1	-2.5	-0.6	-1.0	-1.1	1.2	1.7

	PS		CS		CD		CB	
	V2.2	V2.3	V2.2	V2.3	V2.2	V2.3	V2.2	V2.3
HEFT	-0.3	1.3	1.7	2.3	2.2	2.6	1.7	2.6
DCP	-3.8	-2.1	0.7	0.3	-1.9	-3.3	-0.2	0.0
ETF	-3.6	-2.4	-1.3	0.6	-3.3	-2.8	-2.8	-1.9
Average	-2.6	-1.0	0.4	1.1	-1.0	-1.1	-0.5	0.2

Table 1: Average percentage improvements of the Hybrid Schedulers over all (fine and coarse grained) graphs

In the experiments with the UET-UDT DAGs, we defined a multiplicative computation factor M, so that the execution cost of any task $v_i \in G$ is M. We also vary the latency for different experiments so that the communication costs associated with each edge is equal to L. In this way, we are able to define more accurately fine and coarse grained applications. We carried out a series of experiments with in-trees, out-trees and diamond DAGs, considering the following characteristic (M, L) for each input DAG: $(1, 1)$, $(2, 1)$, $(4, 1)$ and $(8, 1)$ which characterise coarse grained DAGs, and, for fine grained ones the parameters pairs were $(1, 2)$, $(1, 4)$ and $(1, 8)$. For the PSG graphs, the parameters were $(1, 1)$. In all our experiments, the number of processors available was set to the number of tasks in the graph, for the sake of DCP and to allow the schedulers to decide how many processors are necessary to achieve their best makespan. The rest of this section presents highlights of some of the results from over 3000 experiments, each experiment being defined by the following triple: a DAG; a parameter pair (M, L); and a hybrid scheduler. Table 1 shows the average percentage improvement (or degradation when negative) over all graphs and granularities for each hybrid scheduler (i.e. static and dynamic heuristic combination) analysed.

Of the three topologies, the in-tree was the graph which benefited the most from the dynamic scheduling phase. Interestingly for DCP and ETF, a large number of the variants managed to improve the schedule produced by these static heuristics dramatically for fine graphs (by 27.7%). In addition, the variants **V1.1d** and **V1.Nd** for CS and **V1.1d** for CD combined with either DCP, ETF or HEFT also managed to reduce by 50% the number of processors required. As expected, there was no improvement for coarse grained graphs, since the static scheduler tends to produce near optimal solutions. The static schedulers effectively allocate linear clusters to processors (i.e. no independent tasks are allocated together on the same processor). However, for fine grained in-trees, the optimal solution is only achieved when independent tasks are also clustered together. For example, given the DAG shown in Figure 1(b) and $(M = 1, L = 2)$, the optimal solution is achieved when each one the following subsets of tasks are allocated to a distinct processor: $\{v_0, v_1, v_8, v_{12}, v_{13}, v_{14}\}$, $\{v_2, v_3, v_9\}$, $\{v_4, v_5, v_{10}\}$ and $\{v_6, v_7, v_{11}\}$. Dynamic schedulers with the *processor choice* phase based on the scheduled blevel ($blevel_s()$) (i.e. the schedulers based on CS, CD and CB) take the opportunity to cluster the two immediate predecessors of a task v_i together with v_i on the same processor. In other words, during the static scheduling phase, one critical path of the in-tree was allocated to a processor. The dynamic scheduler then re-allocated to this same processor the second most critical path in the DAG.

For out-trees however, the results are quite the opposite. While HEFT benefits by using CD+V1.2d (by 17.9% for fine grained and 7.7% overall), all of the dynamic schedulers worsen the schedules produced statically by ETF and DCP. This effect is particularly noticeable for fine grained graphs, where the dynamic mechanisms seem to undo the

good scheduling decisions to cluster tasks made previously by their static counterparts. During the static scheduling phase, the two successor tasks are only clustered together with their immediate predecessor if it minimises both of their finishing times. Otherwise, the successors are allocated to distinct processors. In this case, the dynamic schedulers, being based on the scheduled blevel, cluster the successor previously scheduled on a distinct processor by the static scheduler together with its immediate predecessor in order to minimise its finishing time, and thus inadvertently causing the other successor to be delayed. On average HEFT is 15% worse than DCP, but using CD+V1.2d, this falls to only 4%.

For the diamond DAGs, the PS based dynamic schedulers turn out to be the best ones, in general. For example, the PS+V1.N and PS+V1.Nd with HEFT give a 6% overall improvement for fine grained graphs. PS+V1.3 and PS+V1.3d improve the ETF by 2.5% and in the case of DCP, PS+V1.2 and PS+V2.3 provided 1.9% improvement, all for fine grained graphs. DCP together with PS+V1.2 or with PS+V2.3, produced the best results. The benefits provided by the PS dynamic scheduler are due to the *task choice* priority. The tasks of the diamond DAG are remapped in accordance with their $blevel_s()$, which turns out to be similar to a block partition where task computation and communication between adjacent blocks are overlapped. Note that the optimal schedule for diamond DAGs is achieved when the graph is partitioned into blocks of tasks whose size is a function of the communication costs [2]. The CS and CB dynamic schedulers have a similar behaviour. In the case of CD, tasks are remapped in the order of their critical path cost (Equation 2), which causes processors to remain idle while critical tasks wait for messages from their predecessors and thus delay the execution of lower priority tasks.

With regard to the PSG graphs, the variant CB+V1.3d provided a 4.2% improvement over ETF and 5.1% over DCP for fine grained graphs, and 2.1% and 3.8%, respectively, over all graphs. HEFT benefited from PS+V2.3d by 7.9% and 1.6%, for fine grained and overall, respectively. In many of the PSG graphs (which are irregular DAGs), a task v_i at level $level(v_i)$ can have one of its predecessors, say u, at level $level(u) < level(v_i) - 1$ (for example, tasks v_4 and v_7 in Figure 1(a)). When scheduling dynamically by blocks, particularly when blocks have tasks from only one level (e.g. V1.1), it is possible for task u to finish its execution before the re-allocation of task v_i. In the dynamic scheduling framework adopted here, the message (u, v_i) cannot be sent until task v_i has been assigned a processor. Since this re-allocation will only be held in a future scheduling event, the sending of the message (u, v_i) will most probably be delayed. This delay will be more pronounced for coarse grained DAGs, since messages (u, v_i) sent to tasks in level $level(v_i)$, could be delayed by as much as the execution time of the first task in level $level(u) + 1$.

As seen in Table 1, compared to the original PS and CS algorithms of [14], all of the variants V1.N, V1.Nd and V2.N, for $N = 2, 3$, produced better results on average for each of the three static schedulers. CD+V1.2, CD+V1.3 and CD+V2.3 for DCP were the only exceptions for the CD variants, i.e. they were the only cases where the new variants produced worse results on average than the original CD (CD+V1.1).

We observed that the variants that reallocated block $B(0)$ provided better results when considering blocks of tasks of more than one level (the variants V1.Nd, $N = 2, 3$). Also, there does not seem to be one variant that was good for all three static algorithms.

In summary, hybrid scheduling can improve the schedules produced by well-known traditional static list scheduling algorithms, such as HEFT, ETF and DCP. The variant CD+V1.2d provided a 9.8% (3.0%) improvement on average over HEFT and CD+V1.3 provided 9.1% (3.5%) on average for all the fine grained graphs (over all graphs). In the case of ETF, CS+V2.3 improved the static scheduler by 4.1% on average for fine grained, but only 0.6% overall. The schedules produced by DCP, which were in general better than those of the other static schedulers (see Table 2), were improved on average 5.4% by CB+V1.3d for fine grained graphs and 2.7% overall. Given the quality of the schedules produced by the static list scheduling algorithms, there is not much room for improvement for coarse grained graphs in general. Table 2 presents a pair-wise comparison of the three static schedulers and the four best hybrid ones (for fine grained and over all graphs) based on the average percentage improvement. From this table, we see that that DCP and DCP with CB+V1.3d produce better schedules than the others. While there may not appear to be a conclusive difference between some of these schedulers, in terms of the average degradation from the best schedules achieved in all experiments, the HEFT hybrids obtain a degradation of 9.0% (CD+V1.3) and 9.4% (CD+V1.2d) compared to 13.3% for HEFT. For ETF, the degradation is 6.1% against 5.1% for CS+V2.3. Finally, DCP is off by 5.9% while the hybrid DCP with CB+V1.2d has an average degradation of only 1.6%.

6. Conclusions and on-going work

This work proposes the use of hybrid schedulers for the problem of scheduling tasks onto a limited number of processors. A hybrid scheduler is effectively a two stage strategy where initially, a static heuristic generates a schedule based on estimated costs for the input DAG and then, a dynamic heuristic uses this scheduling information to generate a new schedule at run-time. This paper analysed the impact of different static heuristics, scheduling event mechanisms

			HEFT CD+V1.3	HEFT CD+V1.2d	DCP Cb+V1.3d	ETF CS+V2.3	HEFT	DCP	ETF
		>		8	16	15	5	14	11
HEFT	CD+V1.3	=		14	13	12	18	14	16
		<		10	3	5	9	4	5
		>	10		16	15	7	14	15
HEFT	CD+V1.2d	=	14		13	14	13	12	12
		<	8		3	3	12	6	5
		>	3	3		2	2	7	5
DCP	Cb+V1.3d	=	13	13		17	14	16	17
		<	16	16		13	16	9	10
		>	5	3	13		3	13	9
ETF	CS+V2.3	=	12	14	17		12	13	17
		<	15	15	2		17	6	6
		>	9	12	16	17		15	13
	HEFT	=	18	13	14	12		16	17
		<	5	7	2	3		1	2
		>	4	6	9	6	1		3
	DCP	=	14	12	16	13	16		22
		<	14	14	7	13	15		7

Table 2: A pair-wise comparison in terms of the number of worse, equal and better schedules

and dynamic heuristics on the performance of hybrid schedulers. The results show that even for a completely static environment, hybrid schedulers can improve on the schedules generated by traditional static heuristics. On going work is currently extending this investigation to both heterogeneous and dynamic environments.

References

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

[2] C. Boeres, G. Chochia, and P. Thanisch. On the scope of applicability of the ETF algorithm. In A. Ferreira and J. Rolim, editors, *The 2nd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'95)*, LNCS 980, pages 159–164, Lyon, France, September 1995. Springer.

[3] C. Boeres, A. P. Nascimento, and V. E. F. Rebello. Cluster-based task scheduling for LogP model. *International Journal of Foundations of Computer Science*, 10(4):405–424, 1999.

[4] C. Boeres and V. Rebello. Easygrid: Towards a framework for the automatic grid enabling of MPI applications. In *Proceedings of the First International Workshop on Middleware for Grid Computing*, Rio de Janeiro, Brazil, Jun 2003.

[5] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 430–437, May 2001.

[6] J. Colin and P. Chrétienne. C.P.M. scheduling with small communication delays and task duplication. *Operations Research*, 39(4):680–684, July 1991.

[7] D. Fernández-Baca. Allocating modules to processors in a distributed system. *IEEE Trans. Software Engrg.*, SE-15(11):1427–143, 1989.

[8] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

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

[10] D. Kondo, H. Casanova, E. Wing, and F. Berman. Models and scheduling mechanisms for global computing applications. In *Proceedings of the Int. Parallel and Distributed Processing Symposium (IPDPS02)*, Fort Lauderdale, USA, Apr 2002.

[11] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating tasks graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[12] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, Dec. 1999.

[13] M. Maheswaran and C. Chen. Distributed dynamic scheduling of composite tasks on grid computing systems. In *Proceedings of the 11th Heterogeneous Computing Workshop*, Nice, France, Apr 2002. IEEE Computer Society Press.

[14] M. Maheswaran and H. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the 7th Heterogeneous Computing Workshop (HCW98)*, pages 57–69, Orlando, Florida, March 1998. IEEE Computer Society Press.

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