From Heterogeneous Task Scheduling to Heterogeneous Mixed Data and Task Parallel Scheduling

Frédéric Suter, Henri Casanova, Frédéric Desprez, Vincent Boudet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

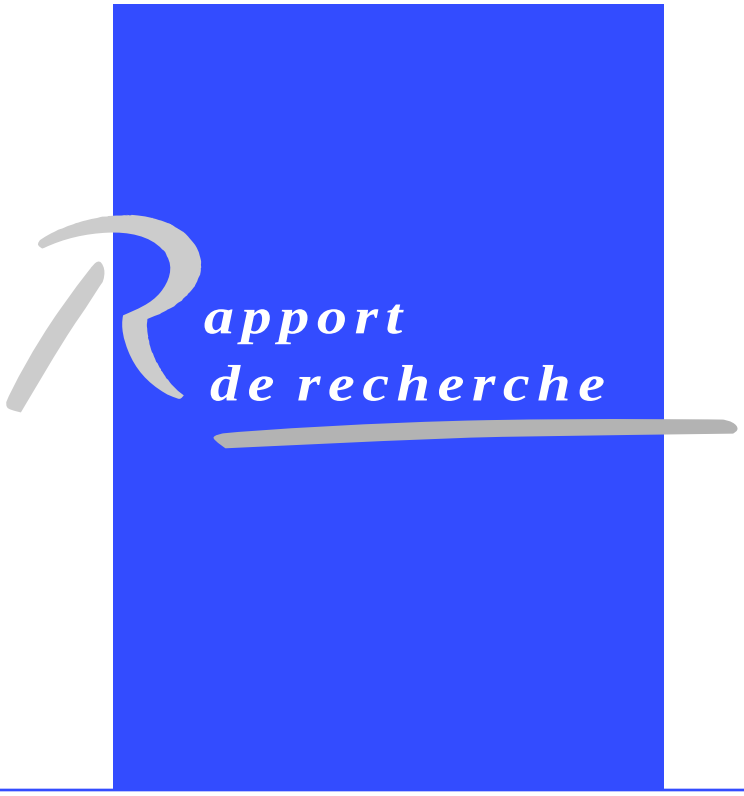
From Heterogeneous Task Scheduling to Heterogeneous Mixed Data and Task Parallel Scheduling

Frédéric Suter, Henri Casanova, Frédéric Desprez, Vincent Boudet

No 4995

November 2003

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4995--FR+ENG

From Heterogeneous Task Scheduling to Heterogeneous Mixed Data and Task Parallel Scheduling

Frédéric Suter*, Henri Casanova, Frédéric Desprez, Vincent Boudet

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n ° 4995 — November 2003 — 17 pages

Abstract: Mixed-parallelism, the combination of data- and task-parallelism, is a powerful way of increasing the scalability of entire classes of parallel applications. Exploiting both types of parallelism simultaneously makes it possible to deploy these applications on platforms comprising multiple compute clusters, which have become increasingly popular in the last decade. However, high performance application executions are only possible if effective scheduling strategies are available. While multi-cluster platforms are predominantly heterogeneous, previous work on mixed-parallel application scheduling targets only homogeneous platforms. In this paper we develop a method for extending existing scheduling algorithms for task-parallel applications on heterogeneous platforms to the mixed-parallel case. After detailing the foundations of our method and our assumptions, we present a case study in which we generate a mixed-parallel version of the popular HEFT scheduling algorithm, which we evaluate with an extensive set of simulation experiments.

Key-words: Mixed Parallelism, heterogeneous task scheduling.

(Résumé : tsvp)

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

* This work has been supported by the INRIA international post-doctoral funding.

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
T´el´ephone : 04 76 61 52 00 - International : +33 4 76 61 52 00
T´el´ecopie : 04 76 61 52 52 - International : +33 4 76 61 52 52

De l'ordonnancement hétérogène de tâches à l'ordonnancement hétérogène mixte

Résumé : Le parallélisme mixte, qui est la combinaison des parallélismes de tâches et de données, est un moyen puissant d'augmenter l'extensibilité de classes entières d'applications parallèles. L'exploitation simultanée des deux types de parallélismes permet de déployer ces applications sur des grappes de grappes. Cependant, des exécutions performantes ne sont seulement possibles que si des stratégies efficaces d'ordonnancement sont disponibles. Alors que les grappes de grappes sont le plus souvent hétérogènes, les travaux précédents dans le domaine de l'ordonnancement en parallélisme mixte ciblent uniquement des plates-formes homogènes. Nous avons développé une méthode pour étendre des algorithmes existants, permettant d'ordonnancer des applications exhibant du parallélisme de tâches sur des plates-formes hétérogènes, afin d'utiliser le parallélisme mixte. Après avoir détaillé les fondations et les hypothèses de notre travail, nous avons présenté une étude de cas dans laquelle nous générons une version «mixte» de l'algorithme d'ordonnancement HEFT. Nous évaluons cette étude par un ensemble complet de simulations.

Mots-clé : Parallélisme mixte, ordonnancement de tâches sur plates-formes hétérogènes.

1 Introduction

Two kinds of parallelism can be exploited in most of scientific application: data- and task-parallelism. The former consists in applying the same operation in parallel on different elements of a data set, while the latter corresponds to concurrent computations on different data sets. One way to maximize the degree of parallelism of a given application is to combine both kinds of parallelism. This approach is called *mixed data and task parallelism* or *mixed-parallelism*. In mixed-parallel applications, several data-parallel computations can be executed concurrently in a task-parallel way. This increases scalability as more parallelism can be exploited when the maximal amount of either data- or task-parallelism has been achieved.

This capability is a key advantage for today's parallel computing platforms. Indeed, to face the increasing computation and memory demands of parallel scientific applications, a recent approach has been to aggregate multiple compute clusters either within or across institutions. These aggregate platforms have been termed *Grids* [7] and Grid middleware [8] provides the necessary software infrastructure for application deployment. Typically, clusters of various sizes are used, and different clusters contain nodes with different capabilities depending on the technology available at the time each cluster was assembled. Therefore, the computing environment is at the same time attractive because of the large computing power, and challenging because it is heterogeneous.

While most mixed-parallelism research has been done in the area of programming languages (see [2] for a broad survey), a fundamental question in that of application *scheduling*. A number of authors have explored this question in the context of homogeneous platforms [14, 15, 16, 17]. However, as explained above, heterogeneous platforms have become prevalent and are extremely attractive for deploying applications at unprecedented scales. Consequently, there is a need for scheduling techniques for mixed-parallel application in the heterogeneous case. In this paper we build on existing scheduling algorithms for heterogeneous platforms [5, 10, 11, 18, 21] (*i.e.,* algorithms specifically designed for task-parallelism) to develop scheduling algorithms for mixed-parallelism on heterogeneous platforms.

This paper is organized as follows. Section 2 discusses related work and formalizes the problem we address. Section 3 shows how scheduling algorithms for task-parallel applications on heterogeneous platforms can be adapted to support mixed-parallelism, which is illustrated with a case study in Section 4. Section 5 presents our evaluation methodology and Section 6 presents our evaluation results. Section 7 concludes the paper with a summary of our contributions and a discussion of future work.

2 Background

2.1 Homogeneous Mixed Data and Task Parallel Scheduling

Most existing mixed-parallel scheduling algorithms [14, 15, 16, 17] proceed in two steps. The first step aims at finding an optimal *allocation* for each task, that is the number of processors

on which the execution time of a task is minimal. The second step determines a *schedule* for the allocated tasks, that is the ordering of tasks that minimizes the total completion time of the application. A classical method used in the allocation step is the minimization of the *critical path* [15]: the longest path in the application task graph, *i.e.,* the path on which the sum of the edge and node weights is maximal. Another possibility is the minimization of the *average area*, which measures the processor-time area required by the application [14]. To be able to estimate the evolution of the execution time of a given task depending on the number of processors allocated to it, these algorithms use either modeling or profiling. Models are principally based on Amdahl's law [1]. For communications, the most common model is to divide the total amount of data to transfer by the bandwidth of the network and add latencies depending on source and destination data distributions.

In [3], we proposed an algorithm that proceeds in only one step. The allocation process is substituted by the association of a list of configurations to tasks, which is also used in this work and detailed in Section 3. However, the algorithm we present in this paper, unlike that in [3], explicitly accounts for platform heterogeneity, and is thus applicable to real-world multi-cluster platforms.

2.2 Heterogeneous Task Scheduling

The problem of scheduling a task graph onto a heterogeneous platform is as follows. Consider a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, that models a parallel application, where $\mathcal{N} = \{N_i : i = 1, \cdots, N\}$ is a set of N nodes (or tasks) and $\mathcal{E} = E_{i,j}$ is a set of edges. Each task has a computation cost, *e.g.,* a number of Flops, which leads to different computation times on different processors. An edge in the DAG corresponds to a task dependency (communication or precedence constraint.) To each edge $E_{i,j}$ is associated D_{ij}, the amount of data in bytes that task N_i sends to task N_j. Each such transfer incurs a communication cost that depends on network capabilities. It is assumed that if two tasks are assigned to the same processor there is no communication cost. Also, we assume that several communications may be performed at the same time, possibly leading to contention on the network.

A task without any input edge is called an *entry* task while a task with no output edge is called an *exit* task. A task is said *ready* when all its predecessors have finished their execution. $Pred(N_i)$ denotes the set of the immediate predecessors of task N_i and $Succ(N_i)$ is the set of the immediate successors of this same task.

The target architecture is generally a set of heterogeneous processors $\mathcal{P} = \{P_k : k = 1, \cdots, p\}$ connected via a fully connected network topology, *i.e.,* as soon as a task N_i has completed, $D_{i,j}$ bytes are sent to all its successors simultaneously. We assume that computation can be overlapped with communication.

We denote by $w_{i,j}$ the execution time of task N_i on processor P_j, which we assume to be known for each task-processor pair.

The communication cost of edge $E_{i,k}$, that is the time to transfer $D_{i,k}$ bytes of data between task N_i (scheduled on P_m) to task N_k (scheduled on P_n) is defined by $c_{i,k} = \beta_{m,n} + D_{i,k} \times \tau_{m,n}$, where $\beta_{m,n}$ and $\tau_{m,n}$ are the communication startup cost and the data transfer rate for the communication link connecting processors P_m and P_n.

The objective is to assign tasks to processors so that the schedule length is minimized, accounting for all interprocessor communication overheads. This problem is NP-complete in the strong sense even when an infinite number of processors are available [5].

A broad class of heuristics for solving the scheduling problem is *list-scheduling*. In the context of heterogeneous platforms popular heuristics list-scheduling heuristics include *minimum Partial Completion Time static priority* (PCT) [10], *Best Imaginary Level* (BIL) [11], *Heterogeneous Earliest Finish Time* (HEFT) [21], and *Dynamic Level Scheduling* (DLS) [18]. All these heuristics are based on the same two components: a *priority* function, which is used to order all nodes in the task graph at compile time; and an objective function, F_{obj}, which must be minimized. A generic list-scheduling algorithm can thus be written as:

Compute the *priority* for each node in the DAG
ReadyTasks = Entry Tasks
While *ReadyTasks* is not empty
    Pick the node N_i that maximizes the *priority* function
    Assign N_i on the processor P_j that minimizes $F_{obj}(N_i, P_j)$
    Update *ReadyTasks*

---

3 Generalization to Heterogeneous Mixed Parallel Scheduling

While the list-scheduling algorithms described in Section 2.2 operate on a fully heterogeneous platform (compute resources and network links), in our first attempt at using these heuristics for mixed-parallelism we impose three restrictions on the platform and its usage: (i) the platform consists of a heterogeneous collection of homogeneous clusters; (ii) the network interconnecting these clusters is fully connected and homogeneous; (iii) data-parallel tasks are always mapped to resources within a single cluster. These restrictions are justified as follows. Restriction (i) clearly makes the problem more tractable but is in fact highly representative of currently available Grid platforms. Indeed, these platforms typically consist of clusters located at different institutions, and institutions typically build homogeneous clusters. Restriction (ii) is more questionable as end-to-end network paths on the wide-area are known to be highly heterogeneous. However, one key issue for scheduling mixed-parallel application is that of *data redistribution*: the process of transferring and remapping application data from one subset of the compute resources to another subset. Data redistribution on heterogeneous networks is a completely open problem, which is known to be NP-complete in the homogeneous case [6]. However, in the homogeneous case a number of algorithms for redistribution have been developed [6, 12, 13, 22, 23]. Consequently, assuming a homogeneous network among clusters allows us to easily model redistribution costs based on these algorithms. Furthermore, no currently available software allows efficient data redistribution in a heterogeneous network setting. Finally, restriction (iii) is just a convenient way to ensure that we can reuse the parallel application models (*e.g.,* speed-up models) traditionally

employed in the mixed-parallelism literature [1, 4]. Conceivably, this restriction could be removed if models for parallel application on heterogeneous platforms were available.

In summary, our computing platform consists of M clusters, where each cluster K_j, for $j = 1, \ldots, M$, contains P_j identical processors. The clusters are interconnected via a homogeneous network. As in Section 2.2, we denote by $\beta_{m,n}$ and by $\tau_{m,n}$ the communication startup cost and the data transfer rate for communication between processor sets S_m and S_n, where each set is a subset of a single cluster.

To utilize list-scheduling algorithms for the purpose of scheduling mixed-parallel application, we introduce the concept of a *configuration*. A configuration is defined as a subset of the set of the processors available within a cluster. Data-parallel tasks are scheduled on configurations. While in task scheduling the smallest computational element is a processor, in our work the smallest element is a configuration. We impose a technical restriction by limiting configurations to contain numbers of processors that are powers of 2. This restriction is convenient (because it reduces the number of possible configurations) and is typical for most data-parallel applications. Also, for the sake of simplicity, we only consider rectangular configurations that span contiguous processors. This is the typical use of parallel compute resources. Finally, for a given configuration size, we only consider configurations that form a non-overlapping tiling of the cluster (*e.g.,* for a cluster with 8 processors we only consider two 1×4 configurations, two 4×1 configurations and two 2×2 configurations). Enforcing non-overlapping configurations ensures that all configurations of the same size can be utilized concurrently by a parallel application. An example of all the configurations we consider for an 8-processor cluster is given in Table 1. All these restrictions are for simplicity and convenience and can be easily removed. However, removing the restriction of rectangular and contiguous configurations may mandate sophisticated application performance models.

Shape	1x1	1x2	2x1	1x4	4x1	2x2	1x8	8x1	2x4	4x2
Number	8	4	4	2	2	2	1	1	1	1

Table 1: List of considered configurations for an 8-processor cluster.

In summary, for the purpose of scheduling, the target architecture is abstracted as collections of c configurations $\mathcal{C} = \{C_k : k = 1, \cdots, c\}$ subject to the restrictions listed above. An upper bound on the number of configurations is:

$$\sum_{i=1}^{M} \sum_{j=0}^{\lfloor log_2(P_i) \rfloor} (j+1) \left\lfloor \frac{P_i}{2^j} \right\rfloor . \tag{1}$$

Similarly to the definition of a parallel program given in Section 2.2, a mixed-parallel program is modeled by a DAG $\mathcal{G} = (\mathcal{N}^*, \mathcal{E})$, where $\mathcal{N}^* = \{N_i^* : i = 1 \cdots N^*\}$ is a set of N^* data-parallel tasks and $\mathcal{E} = E_{i,j}$ is a set of edges representing dependencies among these tasks. Whereas in the purely task-parallel case each task N_i in the DAG was assigned a compute cost, here we assign a *cost vector* to each parallel task. The cost vector represents the compute cost of the task when mapped to each configuration C_k, $k = 1, \ldots, c$. We

denote the compute cost of task N_i^* on configuration C_k by $w_{i,k}^*$. This cost can be set to ∞ if a data-parallel task cannot make use of a particular configuration (*e.g.,* due to memory constraints).

To model $r_{i,s,j,d}$, the time needed to redistribute the amount of data $D_{i,j}$ between task N_i^* (scheduled on the source configuration C_s) and task N_j^* (scheduled on destination configuration C_d), we have to consider two cases. If C_s and C_d are disjoint sets, then the redistribution cost can then be written as:

$$r_{i,s,j,d} = \frac{D_{i,j}}{|C_s|}\tau_{s,d} + \max\left(\frac{|C_d|}{|C_s|}, \frac{|C_s|}{|C_d|}\right)\beta_{s,d}. \quad (2)$$

The first term corresponds to the bandwidth consumption and we assume that all source processors perform communications in parallel. Other communication models can be easily plugged in here without invalidating our approach. The second term corresponds to the latencies. It is based on the maximum number of messages to be exchanged in the data redistribution, which depends on the relative number of source and destination processors and is given by the max clause. If C_s and C_d share processors, some data will be copied on the common processors and these copies can be made in parallel, which saves some bandwidth and one latency cost:

$$r_{i,s,j,d} = \max\left(\frac{D_{i,j}}{|C_s|} - \frac{D_{i,j}}{|C_d|}, \frac{D_{i,j}}{|C_d|} - \frac{D_{i,j}}{|C_s|}\right)\tau_{s,d} + \left(\max\left(\frac{|C_d|}{|C_s|}, \frac{|C_s|}{|C_d|}\right) - 1\right)\beta_{s,d}. \quad (3)$$

With our new representation of the platform as a collection of configuration, a generic heterogeneous mixed data and task parallel scheduling algorithm can then be written as:

Compute the *mixed priority* for each node of the graph
ReadyTasks = Entry Tasks
While *ReadyTasks* is not empty
    Pick the node N_i^* that maximizes the *mixed priority* function
    Assign N_i^* on the configuration C_j that minimizes $F_{obj}(N_i^*, C_j)$
    Update *ReadyTasks*

---

4 Case Study: HEFT

To illustrate our approach we chose to extend the task-parallel scheduling heuristic proposed by Tupcuoglu *et al.* in [21], Heterogeneous Earliest Finish Time (HEFT), to the mixed-parallel case. It is important to note that our approach is general and applicable to other task-parallel scheduling heuristics. We chose HEFT because it is simple, popular, and was shown to be competitive in [21]. In this section we recall the basics of HEFT and show how the priority and objective functions are adapted for mixed-parallelism to develop the M-HEFT (Mixed-HEFT) heuristic.

4.1 HEFT

The priority function used by HEFT is based on "upward ranking". The upward rank of a task N_i is defined recursively by:

$$rank_u(N_i) = \overline{w_i} + \max_{N_j \in Succ(N_i)} \left(\overline{c_{i,j}} + rank_u\left(N_j\right) \right), \tag{4}$$

where $\overline{w_i} = \sum_{j=1}^{p} w_{i,j}/p$ is the average execution cost of task N_i over all available processors. The average communication cost $\overline{c_{i,j}}$ of an edge $E_{i,j}$ is defined as $\overline{c_{i,j}} = \overline{\beta} + D_{i,j}\overline{\tau}$, where $\overline{\tau}$ is the average transfer rate among the processors in the domain and $\overline{\beta}$ is the average communication startup.

The ready tasks are sorted with regard to the decreasing order of the $rank_u$ values. The task with the highest priority is selected. If two candidate nodes have the same priority, one of them is selected randomly.

Before introducing the objective function of HEFT, some notations are needed. The Earliest Start Time (EST) and the Earliest Finish Time (EFT) of task N_i on processor P_j are defined as follows:

$$EST(N_i, P_j) = \max\left(Avail(P_j), \max_{N_k \in Pred(N_i)} \left(AFT(N_k) + c_{k,i} \right) \right), \tag{5}$$

$$EFT(N_i, P_j) = w_{i,j} + EST(N_i, P_j), \tag{6}$$

where $Avail(P_j)$ is the earliest time at which processor P_j is ready for the task execution. The inner max block in the EST equation represents the *ready time* for task N_i, *i.e.,* the time when all needed data has arrived on processor P_j. $AFT(N_k)$ denotes the *Actual Finish Time* of task N_k, *i.e.,* this task has been scheduled on a processor, and its finish time is known.

HEFT uses EFT as the objective function for selecting the best processor for a node. The rationale is that the schedule length is the earliest finish time of the exit node. The node N_i is thus assigned to the processor P_j that minimizes $EFT(N_i, P_j)$. The EST values for each of the remaining node and the set of ready tasks are then updated.

4.2 M-HEFT

In this section we present modifications to the priority and objective functions of the HEFT algorithm to derive a mixed-parallel parallel version, M-HEFT. We first adapt the computation of $rank_u$, the HEFT priority function. We introduce the mixed-parallel upward rank of task N_i^* defined recursively as:

$$rank_u^*(N_i^*) = \overline{w_i^*} + \max_{N_j^* \in Succ(N_i^*)} \left(\overline{r_{i,j}} + rank_u\left(N_j^*\right) \right), \tag{7}$$

where $\overline{w_i^*}$ is the average mixed-parallel execution cost of task N_i^* and $\overline{r_{i,j}}$ is the average redistribution cost of an edge $E_{i,j}$. Several options are possible for defining these averages. In this paper we consider the 2 possible definitions for $\overline{w_i^*}$:

(a) Following exactly the same approach as HEFT, compute $\overline{w_i^*}$ as the sum of the compute times for N_i^* over all 1-processor configurations, divided by the number of such configurations (*i.e.,* the number of processors in the platform).

(b) Adapting HEFT directly to the mixed-parallel case, one can compute $\overline{w_i^*}$ as $\sum_{j=1}^{c} w_{i,j}^*/c$, where c is the number of configurations. But in this case several identical configurations are accounted for in each cluster, which may result in a biasing of the average. For instance, in an 8-processor cluster, the above formula for $\overline{w_i^*}$ will incorporate compute costs for 8 identical 1-processor configurations. To avoid the biasing of the average, we enforce that, for a given configuration size, only one configuration be taken into account for each cluster.

Similarly, we use the following 2 definitions for $\overline{r_{i,j}}$:

(c) As in HEFT, assume serial communications and compute $\overline{r_{i,j}}$ as the sum of the average network latency and of the data size divided by the average network bandwidth. In this paper we only consider homogeneous networks, so the average latency and bandwidth are equal to the latency and bandwidth of any network link.

(d) Adapting HEFT directly to the mixed-parallel case, one can compute $\overline{r_{i,j}}$ as $\sum_{s,d=1}^{c} r_{i,s,j,d}/c^2$. Here again, several identical redistributions are accounted for, which may result in a biasing of the average. We enforce that for a given source configuration size and a given destination configuration size, only one redistribution be taken into account.

In this paper we consider the following two combinations of the above definitions of $\overline{w_i^*}$ and $\overline{r_{i,j}}$ for the M-HEFT priority function: (a) & (c) and (b) & (d), which we denote by M-HEFT1 and M-HEFT2.

To complete our description of M-HEFT, we need to define its objective function. We use the same approach as HEFT, but modify the computation of the EST and the EFT values to support a mixed-parallel execution. We introduce EST^* and EFT^* defined as follows:

$$EST^*(N_i^*, C_j) = \max\left(Avail(C_j), \max_{N_k^* \in Pred(N_i^*)} (AFT(N_k^*, C_l) + r_{k,l,i,j})\right), \tag{8}$$

$$EFT^*(N_i^*, C_j) = w_{i,j}^* + EST^*(N_i^*, C_j), \tag{9}$$

where $Avail(C_j)$ is the earliest time at which all the processors of the configuration C_j are ready for the task execution. Note the use of the redistribution cost $r_{k,l,i,j}$, which replaces the communication cost $c_{k,i}$ in the task-parallel HEFT. As in the task-parallel version, $AFT(N_k^*, C_l)$ denotes the *Actual Finish Time* of task N_k^*, *i.e.,* this task has already been scheduled on a configuration C_l, and its finish time is known.

5 Evaluation Methodology

We resort to simulation for evaluating our approach as it allows us to perform a statistically significant number of experiments and makes is possible to explore a wide range of platform configurations. We use the SimGrid toolkit [9, 19] as the foundation of our simulator. SimGrid provides base abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms. We detail below the platforms, applications, and scheduling algorithms that we use in our simulations.

5.1 Platforms

We consider platforms that consist of 1, 2, 4, or 8 clusters. Each cluster contains a random number of processors between 4 and 64, not limited to powers of 2. Processor speed is homogeneous within each cluster; cluster processor speeds are sampled from a uniform probability distribution, for various means and ranges, with the range being proportional to the mean. We define processor speed as the number of billion floating point operations per second (GFlop/sec). For all experiments we assume that the network has a 5 ms latency and a 10 GBit/sec bandwidth. We keep the network characteristics fixed and vary the processor speeds to experiment with a range of platform communication/computation ratios. Table 2 lists the values we used for these parameters, the combination of which generates 280 different platform configurations. Note that each instantiation has random elements, namely the number of processors per cluster, and each processor speed. Therefore, we generate several samples for each platform configuration.

number of clusters	1, 2, 4, 8
mean processor speed (GFlop/sec)	1, 5, 10, 50, 100, 500, 1000
processor speed range	0, $0.2 \times mean$, ..., $1.8 \times mean$
network latency (ms)	5
network bandwidth (GBit/sec)	10

Table 2: Parameters for simulated platforms.

5.2 Applications

We consider two classes of application DAGs, depicted in Figure 1, and which we detail below.

Strassen – In 1969, Strassen proposed an algorithm to compute the product of two 2×2 matrices that needs 7 multiplications instead of 8 [20]. The eight multiplication is replaced by a set of eighteen additions of lower complexity. This algorithm can be applied recursively to 2×2 block matrices to achieve an asymptotic complexity of $\mathcal{O}(n^{\log(7)})$, where n is the

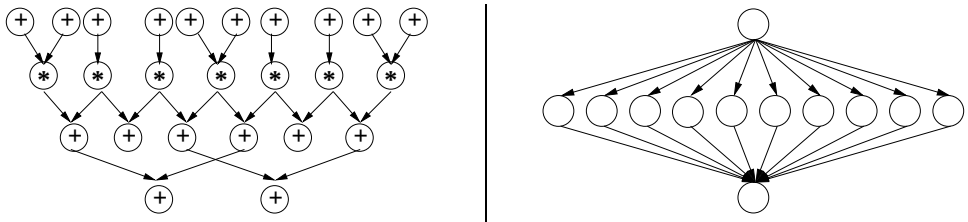


Figure 1: Application DAGs.

dimension of the matrices. The left part of Figure 1 shows the DAG corresponding to the first level of decomposition of the Strassen algorithm, where nodes marked with a '+' represent matrix additions, and those marked with a '*' represent matrix multiplications. We instantiate 6 such DAGs, which we call Strassen-d for $d = 2, ..7$, such that each task in Strassen-d operates on a $(1000 \times 2^d) \times (1000 \times 2^d)$ matrix.

Fork-Join – We also experiment with simple fork-join DAGs so that we can evaluate our algorithms for grahs with high degree of task-parallelism. In particular, the task-parallel HEFT heuristic should be effective for such graphs. We instantiate fork-join DAGs with 10, 50, and 100 tasks. Each task is either a matrix addition or matrix multiplication (which makes it possible to reuse or data-parallel application perform models). We also assume that the entry task send two matrices to each inner task, and that each inner task send the result of the matrix operation, *i.e.,* one matrix, to the end task. We generate DAGs with 25%, 50%, and 75% of multiplication tasks. We have thus 9 possible fork-join graph configurations. For each task we pick the matrix size in a similar fashion as for the Strassen DAGS described above.

5.3 Scheduling Algorithms

For all platform and application configurations described above we compare the effectiveness of 3 scheduling algorithms. The first two are HEFT and M-HEFT (of which we have the M-HEFT1 and M-HEFT2 variants) that were described in Sections 4.1 and 4.2 respectively. We also introduce here a third algorithm, HEFT*, which is a simple data-parallel extension of HEFT and which operates as follows. For each cluster in the platform, HEFT* first determines the number of processors in the largest feasible configuration, and then computes the smallest such number over all clusters, p^*. The HEFT* algorithm then executes each node in the application DAG on configurations with p^* processors. The rationale behind HEFT* is that, unlike HEFT, it can take advantage of data-parallelism. Therefore, while HEFT is limited to using only as many processors as the maximum number of concurrently executable application tasks, HEFT* can make better use available compute resources and is thus a better comparator for evaluating M-HEFT. However, unlike M-HEFT, HEFT* cannot

adapt the configuration size of different data-parallel tasks to achieve better utilization of compute and network resources.

6 Simulation Results

6.1 Strassen

For the Strassen application we only present results for the first variant of M-HEFT, M-HEFT1. The Strassen DAG has a layered structure and the $rank_u^*$ value of two tasks at the same layer differ at most by the cost of an addition, which is generally insignificant. Therefore, the different methods used to compute $rank_u^*$ in M-HEFT1 and M-HEFT2 lead to virtually identical task orderings.

Since we only use 6 DAGs for the Strassen application, we perform simulations over a large number of platforms. For a given number of clusters, M, Table 2 shows that we consider $7 \times 10 = 70$ different platform configurations. For these simulations we instantiate $10 \times M$ samples for each of these platform configurations. The rationale is that we wish to generate more samples for platforms with larger numbers of clusters, as a larger number of cluster implies a larger number of possible platform topologies. We thus performed a total of 21,000 simulation runs.

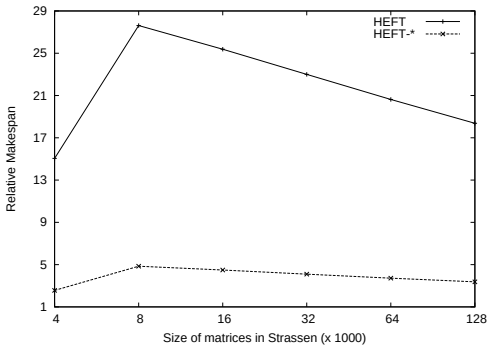


Figure 2: Relative makespans for HEFT and HEFT* versus the matrix size used in the Strassen DAGs.

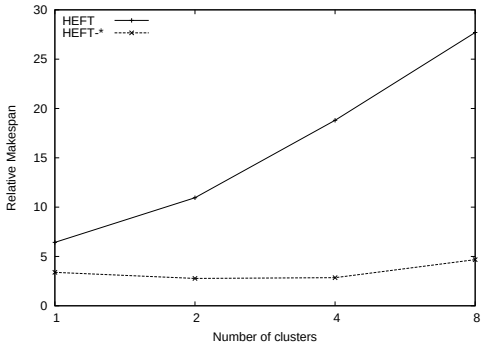


Figure 3: Relative makespans for HEFT and HEFT* versus the number of clusters in the platform.

Over all our simulation results, the average makespans of HEFT and HEFT*, relative to that of M-HEFT, are 21.67 and 3.85 respectively. M-HEFT outperforms HEFT by almost one order of magnitude, which was expected as HEFT cannot take advantage of any data-parallelism and thus is highly limited in the number of resources it can use. More interestingly, the fact that M-HEFT clearly outperforms HEFT* demonstrates that our

scheme for determining appropriate configuration sizes for data-parallel tasks is effective. We now analyze our results in more details.

Figure 2 shows relative makespans for HEFT and HEFT* versus the matrix size used in the Strassen DAGs (from $4,000 \times 4,000$ to $128,000 \times 128,000$). We can see that the relative performance of HEFT versus that of M-HEFT decreases from matrix size 4,000 to size 8,000, and increases thereafter. The original decrease can be explained as follows. M-HEFT maps data-parallel matrix additions to very large configurations as they can be performed without communications. On the other hand, data-parallel matrix multiplications involve inter-processor communications and are thus mapped to smaller configurations. This results in many data redistributions that incur potentially large network latencies. For relatively small matrices, these latencies are thus less amortized over the computation. For the same reason, as matrices get large, redistribution costs increase due to bandwidth consumption. We observe similar trends for HEFT*, but its performance is more on par with that of M-HEFT. Nevertheless, M-HEFT is still a factor 3.85 better on average. This is because HEFT*, unlike M-HEFT, is limited in the size of processor configuration that could be used for expensive data-parallel tasks (*e.g.,* matrix multiplications).

In summary, while M-HEFT clearly outperforms its competitors, its performance is negatively impacted by redistribution costs, which indicates that these costs are not accounted for adequately. This points to a subtle limit of our approach. The first term in the $EFT^\star$ function given in Eq. 9 includes only a computation cost, and no redistribution cost. Therefore, the mapping of the entry tasks of the application graphs are chosen without consideration of initial data-distribution costs. This in turn leads to these tasks being distributed on potentially very large configurations (typically for matrix additions in the case of the Strassen application), and thus to expensive data redistributions for subsequents tasks in the application task graph. This explains why M-HEFT leads to overly expensive data redistributions in some cases. It is important to note that our adaptation of HEFT into M-HEFT is somewhat naïve as our goal in this paper is to provide a generic method for adapting task-parallel list-scheduling heuristics to the mixed-parallel. It would be possible to further improve M-HEFT (and possibly other mixed-parallel list-scheduling heuristics obtained via our methodology) by including redistribution costs explicitly into the objective function.

Figure 3 shows relative makespans for HEFT and HEFT* versus the number of clusters in the platform. As expected HEFT's relative makespan increases steadily as the number of clusters increases because, while M-HEFT can exploit large computing platforms, HEFT is limited to using 10 processors for Strassen DAGs (the maximal number of tasks that can be executed concurrently). We see that HEFT*'s relative makespan also increases after 4 clusters. Recall that the p^* value is computed as a minimum over all clusters. Therefore, even when many clusters are available and some of these clusters are large, HEFT* is limited to using configurations containing at most the number of processors of the smallest available cluster. This hinders performance for expensive data-parallel operations on large matrices. By contrast, M-HEFT has the ability to use large processor configurations on large clusters for these large matrices.

6.2 Fork-join

We performed simulations for 10 samples of each of the 9 possible DAG configurations. In terms of the platform, we simulated 10 samples of each of the $4 \times 70 = 280$ possible platform configurations described in Table 2, for a total of 2,800 simulation runs.

Unlike for the Strassen application, we compared the two variants M-HEFT1 and M-HEFT2. We found that overall the experiments M-HEFT1 led to better schedules than M-HEFT2 in 33% of the cases, that M-HEFT2 led to better schedules than M-HEFT1 in 26% of the cases, and that in the remaining 41% the two led to the same performance. Furthermore, the average performance of M-HEFT1 relative to M-HEFT2 is 1.15, with a maximum of 23.69. These results show that the two variants exhibit roughly similar efficacy. Since M-HEFT2 has the highest complexity we only present results for M-HEFT1 hereafter.

The average performance of HEFT relative to M-HEFT over all experiments is 4.70 and that of HEFT* is 12.11. This is in sharp contrast with the results we obtained for the Strassen application. As mentioned in Section 5.2, some of our fork-joins graphs exhibit high levels of task-parallelism, which can be exploited by HEFT. On the other hand, HEFT* uses data-parallelism for all tasks with the same configuration size for all tasks. This both leads to small tasks being executed in parallel with high overhead, and to a limited ability to exploit high levels of task-parallelism. M-HEFT adapts configuration sizes to avoid this pitfall.

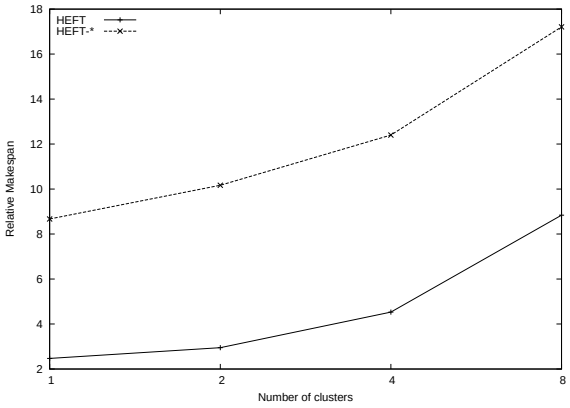


Figure 4: Relative makespans for HEFT and HEFT* versus the number of clusters in the platform.

Figure 4 shows the performance of HEFT and HEFT* relative to M-HEFT as the number of cluster increases. This is the same trend as what we observed for the Strassen application: as more clusters become available, some of which may be large, M-HEFT is able to use large configurations efficiently. Another trend that we observed is that the higher the proportion of matrix multiplications, the worse the relative performance of HEFT when compared to

HEFT* and M-HEFT. This is due to the fact that matrix multiplications are expensive operations to perform on one single processor, and that the use of data-parallelism becomes key for high performance.

7 Conclusion and Future Work

The combination of data- and task-parallelism, *i.e., mixed-parallelism*, is a well-known technique for increasing the scalability of many parallel applications. In particular, mixed-parallelism makes is possible for large-scale applications to efficiently exploit compute platforms that comprise several compute clusters. While mixed-parallel scheduling algorithms have been proposed in the case of homogeneous platforms [14, 15, 16, 17], no work has been conducted in the case of heterogeneous platform. However, the aforementioned platforms are almost always heterogeneous and there is thus a clear need for efficient scheduling algorithms for mixed-parallel applications on heterogeneous platforms. In this paper we have addressed this need with a generic methodology for developing such algorithms.

Our approach allows the conversion of any list-scheduling algorithm for task-parallel application into an algorithm for the mixed-parallel case. This is done by defining the base computational unit as a *configuration*, which is a set of processors within a cluster. Data-parallel tasks can be scheduled on configurations of various sizes. This simple technique makes it possible to fusion the data-parallel task mapping and the task-parallel task scheduling into a single step. We have presented a case study for the popular HEFT scheduling algorithm, which we have extended to obtain the M-HEFT (Mixed-parallel HEFT) algorithm. We presented results evaluating the performance of M-HEFT for a variety of platform scenarios and for two families of application DAGs. We compared M-HEFT to the original HEFT algorithm and to HEFT*, a mixed-parallel version of HEFT that does not use the concept of configurations. Our results showed that M-HEFT achieves good performance in the vast majority of the scenarios, and we have discussed a number of trends apparent in our results. While this case study has value, our main contribution is the fact that our methodology is generic and can be directly applied to other task-parallel scheduling algorithms, conceivably even to those that do not use list-scheduling.

This work can be extended in several directions. First, we plan to apply and further validate our methodology to other list-scheduling algorithms [10, 11, 21, 18]. Second, we have made a number of assumptions about the platform and its utilization, most of which are restrictive and can be relaxed as explained in Section 3. For instance, one could allow for configuration sizes that are not powers of 2. This would imply a larger search space and a higher complexity for our scheduling algorithm, but it would interesting to quantify the trade-off between this increase in scheduling cost and the potential increase in application performance. Also, it would be of great interest to use a data redistribution model for heterogeneous configurations (both heterogeneous processors and network). However, no such model is readily available in the literature and developing a full-fledge one raises an impressive number of challenging issues. At first, an approach could be to consider only special but relevant heterogeneous configuration (*e.g.,* two different sets of homogeneous processors

connected over single network link), rather than the fully heterogeneous case. Third, as explained in Section 6.1, we will improve our generic methodology by incorporating redistribution costs in the objective function to reduce redistribution overheads in the application schedule.

References

[1] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of American Federation of Information Processing Societies (AFIPS) 1967 Spring Joint Computer Conference*, volume 30, pages 483–485, April 1967.

[2] Henri Bal and Matthew Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[3] Vincent Boudet, Frédéric Desprez, and Frédéric Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[4] Eddy Caron and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, Iaşi, Romania, July 2002.

[5] Philippe Chretienne. Task Scheduling Over Distributed Memory Machines. In Michel Cosnard, Patrice Quinton, Michel Raynal, and Yves Robert, editors, *Parallel and Distributed Algorithms*, pages 165–176. North Holland, 1988.

[6] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling Block-Cyclic Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):192–205, February 1998.

[7] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998. ISBN 1-55860-475-8.

[8] Globus. `http://www.globus.org/`.

[9] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the 3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, Tokyo, Japan, May 2003.

[10] Muthucumaru Maheswaran and Howard Jay Siegel. A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. In *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 57–69, Orlando, USA, March 1998. IEEE Computer Society Press.

[11] Hyunok Oh and Soonhoi Ha. A Static Scheduling Heuristic for Heterogeneous Processors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of Europar'96*, volume 1124 of *Lecture Notes in Computer Science*, pages 573–577, Lyon, France, August 1996. Springer-Verlag.

[12] Neungsoo Park, Viktor Prasanna, and Cauligi Raghavendra. Efficient Communication Schedule for Block-Cyclic Array Redistribution between Processor Sets. *IEEE Transactions on Parallel and Distributed Systems*, 10(12), December 1999.

[13] Loïc Prylli and Bernard Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, August 1997.

[14] Andrei Radulescu, Cristina Nicolescu, Arjan van Gemund, and Pieter Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001.

[15] Andrei Radulescu and Arjan van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 15th International Conference on Parallel Processing (ICPP)*, Valencia, Spain, September 2001.

[16] Shankar Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[17] Thomas Rauber and Gudula Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[18] Gilbert Sih and Edward Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[19] SimGrid. `http://gcl.ucsd.edu/simgrid/`.

[20] Volker Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[21] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[22] Akiyosh Wakatani and Michael Wolfe. Optimization of Array Redistribution for Distributed Memory Multicomputers. *Parallel Computing*, 21(9):1485–1490, 1995.

[23] David Walker and Steve Otto. Redistribution of Block-Cyclic Data Distributions Using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399