From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling*

Frédéric Suter[1], Frédéric Desprez[2], and Henri Casanova[1,3]

[1] Dept. of CSE, Univ. of California, San Diego, USA
[2] LIP ENS Lyon, UMR CNRS - ENS Lyon - UCB Lyon - INRIA 5668, France
[3] San Diego Supercomputer Center, Univ. of California, San Diego, USA

Abstract. Mixed-parallelism, the combination of data- and task-parallelism, is a powerful way of increasing the scalability of entire classes of parallel applications on platforms comprising multiple compute clusters. 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.

1 Introduction

Two kinds of parallelism can be exploited in most scientific applications: data- and task-parallelism. 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 [4]. 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.

A number of authors have explored mixed-parallel application *scheduling* in the context of homogeneous platforms [8–10]. However, heterogeneous platforms have become prevalent and are extremely attractive for deploying applications at unprecedented scales. In this paper we build on existing scheduling algorithms for heterogeneous platforms [6, 7, 11, 14] (*i.e.,* specifically designed for task-parallelism) to develop scheduling algorithms for mixed-parallelism on heterogeneous platforms.

* An extended version of this paper is given by [13].

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 230–237, 2004.
© Springer-Verlag Berlin Heidelberg 2004

This paper is organized as follows. Section 2 discusses related work. 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

Most existing mixed-parallel scheduling algorithms [8–10] 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.

In [1], 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. This concept of configuration has been developed to be used in this work and will be detailed in Section 3. Furthermore, the algorithm we present in this paper, unlike that in [1], explicitly accounts for platform heterogeneity, and is thus applicable to real-world multi-cluster platforms.

The problem of scheduling a task graph onto a heterogeneous platform is as follows. Consider a Directed Acyclic Graph (DAG) that models a parallel application. Each node (or task) of the DAG has a computation cost 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 connecting two nodes is associated the amount of data in bytes to communicate. 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 and that several communications may be performed at the same time, possibly leading to contention on the network.

The target architecture is generally a set of heterogeneous processors connected via a fully connected network topology, *i.e.,* as soon as a task has completed, produced data are sent to all its successors simultaneously. We assume that computation can be overlapped with communication and the execution time to be known for each task-processor pair.

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 [2]. 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 *PCT* [6], *BIL* [7], *HEFT* [14], and *DLS* [11]. 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.

3 Towards Heterogeneous Mixed Parallel Scheduling

While the list-scheduling algorithms described in Section 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 [3]. Assuming a homogeneous network among clusters allows us to more easily model redistribution costs. 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. This restriction can be removed in cases in which models for parallel application on heterogeneous platforms are available. In summary, our computing platform consists of several homogeneous clusters of different speeds and sizes interconnected via a homogeneous network.

To utilize list-scheduling algorithms for the purpose of scheduling mixed-parallel application, we have adapted the concept of *configuration* of [1]. A configuration is now defined as a subset of the set of the processors available within, and only within, a cluster. Moreover, we keep only information about the size and shape of the virtual grid represented by the configuration. While in task scheduling the smallest computational element is a processor, in our work the smallest element is a configuration. We limit configurations to contain numbers of processors that are powers of 2, to be rectangular, and to span contiguous processors, which is usual for the vast majority data-parallel applications. These limitations are for simplicity and to reduce the total number of configurations to consider, and can be removed easily if needed. For a given configuration size, we only consider a non-overlapping tiling of the cluster. This ensures that all configurations of the same size can be utilized concurrently. Removing this restriction would mandate sophisticated application performance models that are currently rarely available. For the purpose of scheduling, the target architecture is then abstracted as collections of configurations subject to the restrictions listed above.

Mixed-parallel programs can also be modeled by a DAG, in which nodes represent data-parallel tasks. Whereas in the purely task-parallel case each task was assigned a compute cost, here we assign a *cost vector* to each parallel task, that represents the compute cost of the task when mapped to each configuration.

4 Case Study: HEFT

To illustrate our approach we chose to extend the task-parallel scheduling heuristic proposed in [14], 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.

The priority function used by HEFT is based on "upward ranking". Basically, it is the length of the critical path from a task to the exit task, including the computation cost of this task. The upward rank of a task is the sum of the average execution cost of this task over all available processors and a maximum computed over all its successors. The terms of this maximum are the average communication cost of an edge and the upward rank of the successor.

The Earliest Start Time (EST) is the moment when the execution of a task can actually begin on a processor. An execution can start either when a processor becomes available or when all needed data has arrived on the processor. Adding the execution cost, we obtain the Earliest Finish Time (EFT) of a task.

HEFT uses the EFT as the objective function for selecting the best processor for a node. The rationale is that the schedule length is the EFT of the exit node.

To derive a mixed-parallel version of the HEFT algorithm, called M-HEFT (Mixed-parallel HEFT), the priority and objective functions of HEFT have to be adapted the new compute and communication units. We consider two approaches to define the average mixed-parallel execution cost of a task and average mixed-parallel redistribution cost of an edge.

We first follow exactly the same approach as HEFT, *i.e.,* compute the average mixed-parallel execution cost as the sum of the compute times for the task over all 1-processor configurations, divided by the number of such configurations (*i.e.,* the number of processors in the platform). Similarly we assume serial communications and compute the average mixed-parallel communication cost 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. We denote this heuristic by M-HEFT1.

The second approach is to adapt HEFT directly to the mixed-parallel case. One can compute average mixed-parallel execution cost of task as the sum of each element of the cost vector of the task divided by the number of configurations and the average redistribution cost of an edge as the sum of the redistribution cost for each couple of configurations divided by the number of such couples.

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 execution cost formula 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. For the same reason, we enforce that for a given source configuration size and a given destination configuration size, only one redistribution be taken into account. We denote this heuristic by M-HEFT2.

5 Evaluation Methodology

We use 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 [5, 12] as the foundation of our simulator.

We consider platforms that consist of 1 to 8 clusters. Each cluster contains a random number of processors between 4 and 64. Cluster processor speeds (in GFlop/sec) are sampled from a uniform probability distribution, for various means ([1–1000]) and ranges ([0–1.8]), with the range being proportional to the mean. 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. These parameters generate 280 different platform configurations. We generate several samples for each of them.

We consider two classes of application DAGs, depicted in Figure 1. The left part shows the DAG corresponding to the first level of decomposition of the Strassen algorithm. We instantiate 6 DAGs with different matrix sizes.

Fig. 1. Strassen (left) and Fork-Join (right) DAGs.

We also experiment with simple fork-join DAGs so that we can evaluate our algorithms for graphs 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. 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.

We compare the effectiveness of HEFT, M-HEFT and a third algorithm, HEFT*, which is a simple data-parallel extension of HEFT. For each cluster in the platform, HEFT* first determines the number of processors in the largest feasible configuration, and then computes p^*, the smallest such number over all clusters. The HEFT* algorithm then executes each task on configurations with p^* processors. The rationale behind HEFT* is that, unlike HEFT, it can take advantage of data-parallelism, and is thus a better comparator for evaluating M-HEFT. However, unlike M-HEFT, HEFT* cannot adapt the configuration size to achieve better utilization of compute and network resources.

6 Simulation Results

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 upward rank 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 in M-HEFT1 and M-HEFT2 lead to virtually identical task orderings.

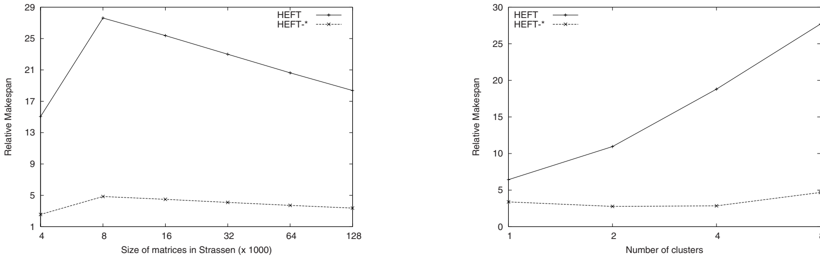

Fig. 2. Relative makespans for HEFT and HEFT* versus the matrix size used in the Strassen DAGs (left) and the number of clusters in the platform (right).

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 more than 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.

Figure 2(left) shows relative makespans for HEFT and HEFT* versus the matrix size used in the Strassen DAGs. 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 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 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 mixed parallel EFT function 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 2(right) shows relative makespans for HEFT and HEFT* versus the number of clusters in the platform. As expected HEFT's relative makespan increases as the number of clusters increases because, while M-HEFT can exploit large computing platforms, HEFT is limited to using 10 processors (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 cluster.

Unlike for the Strassen application, we compared the two variants M-HEFT1 and M-HEFT2 for the fork-join DAG. We found that overall the experiments M-HEFT1 led to better schedules in 33% of the cases, that M-HEFT2 led to better schedules 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 before 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.

7 Conclusion and Future Work

In this paper we have proposed a generic methodology for the conversion of any heterogeneous list-scheduling algorithm for task-parallel application into an algorithm for the mixed-parallel case. 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. Our simulation results showed that

M-HEFT achieves good performance over competitors in the vast majority of the scenarios.

The first future work of this paper is to validate the approach with more significant benchmarks. The first step will be to define a large set of generated DAGs. Parameters such as shape, density, regularity, communication to computation ratio will have to be part of the DAG generation. The second step is to study the data redistribution with heterogeneous configurations (both heterogeneous processors and network). Our approach would be to consider initially 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, we will improve our generic methodology by incorporating redistribution costs in the objective function to reduce redistribution overheads in the application schedule.

References

1. V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Apr 2003.
2. P. Chretienne. Task Scheduling Over Distributed Memory Machines. In *Parallel and Distributed Algorithms*, pages 165–176. North Holland, 1988.
3. F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. *IEEE TPDS*, 9(2):192–205, 1998.
4. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998. ISBN 1-55860-475-8.
5. A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proc. of the 3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, Tokyo, May 2003.
6. M. Maheswaran and H. J. Siegel. A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. In *Proc. of the 7th Heterogeneous Computing Workshop (HCW'98)*, pages 57–69, 1998.
7. H. Oh and S. Ha. A Static Scheduling Heuristic for Heterogeneous Processors. In *Proceedings of Europar'96*, volume 1124 of *LNCS*, pages 573–577, 1996.
8. A. Radulescu, C. Nicolescu, A. van Gemund, and P. Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proc. of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, Apr 2001.
9. S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications.* PhD thesis, Univ. of Illinois at Urbana-Champaign, 1996.
10. T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.
11. G. Sih and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE TPDS*, 4(2):175–187.
12. SimGrid. `http://gcl.ucsd.edu/simgrid/`.
13. F. Suter, H. Casanova, F. Desprez, and V. Boudet. From Heterogeneous Task Scheduling to Heterogeneous Mixed Data and Task Parallel Scheduling. Technical Report RR2003-52, Laboratoire de l'Informatique du Parallélisme (LIP), Nov 2003.
14. H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE TPDS*, 13(3):260–274, 2002.