Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids

C. Banino[1], O. Beaumont[1], A. Legrand[2], and Y. Robert[2]

[1] LaBRI, UMR CNRS 5800, Domaine Universitaire, 33405 Talence Cedex, France
[2] LIP, UMR CNRS-INRIA 5668 ENS de Lyon, 69364 Lyon Cedex 07, France

Abstract. In this paper, we consider the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous "grid" computing platform. We use a non-oriented graph to model a grid, where resources can have different speeds of computation and communication, as well as different overlap capabilities. We show how to determine the optimal steady-state scheduling strategy for each processor.
Because spanning trees are easier to deal with in practice, a natural question arises: how to extract the best spanning tree, i.e. the one with optimal steady-state throughput, out of a general interconnection graph? We show that this problem is NP-Complete. Still, we introduce and compare several low-complexity heuristics to determine a sub-optimal spanning tree.

1 Introduction

In this paper, we deal with the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous "grid" computing platform. We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of an undirected graph. Each node is a computing resource (a processor, or a cluster, or whatever) capable of computing and/or communicating with its neighbors at (possibly) different rates.

We assume that one specific node, referred to as the master, initially holds (or generates the data for) a large collection of independent, identical tasks to be allocated on the grid. The question for the master is to decide which tasks to execute itself, and how many tasks to forward to each of its neighbors. Due to heterogeneity, the neighbors may receive different amounts of work (maybe none for some of them). Each neighbor faces in turn the same dilemma: determine how many tasks to execute, and how many to delegate to other processors.

The master may well need to send tasks along multiple paths to properly feed a very fast but remote computing resource. The master-slave scheduling problem for a general interconnection graph is to determine a steady state scheduling policy for each processor, i.e. the fraction of time spent computing, and the fraction of time spent sending or receiving tasks along each communication link, so that the (averaged) overall number of tasks processed at each time-step is maximum. In this paper, we solve the master-slave scheduling problem for general graphs,

J. Fagerholm et al. (Eds.): PARA 2002, LNCS 2367, pp. 423–432, 2002.

using a linear programming formulation (which nicely encompasses the situation where there are several masters instead of a single one).

The master-slave scheduling problem is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [14], factoring large numbers [7], the Mersenne prime search [13], and those distributed computing problems organized by companies such as Entropia [8].

Several papers [15,11,9,16,3,2] have recently revisited the master-slave paradigm for processor clusters or grids, and we refer to Section 7 for comparison and discussion.

This paper is a follow-on of recent work by Beaumont et al. [2], who solve the master-slave scheduling problem for a tree-shaped heterogeneous platform. Given an oriented spanning tree rooted at the master, they aim at determining the optimal steady-state scheduling strategy. Interestingly, it turns out that this strategy is *bandwidth-centric*: if enough bandwidth is available to the node, then all children are kept busy; if bandwidth is limited, then tasks should be allocated only to children which have sufficiently fast communication times, in order of fastest communication time. Counter-intuitively, the maximum throughput in the tree is achieved by delegating tasks to children as quickly as possible, and not by seeking their fastest processing.

Given a network topology (that may well include cycles and multiple paths), how to extract the "best" spanning tree, i.e. the spanning tree which allows for the maximum number of tasks to be processed by all the computing resources? Given a tree, the result by Beaumont et al [2] enables to compute the best scheduling strategy for that tree, but is of no help to find the tree. Because there may exist an exponential number of trees rooted at the master, we cannot simply compute the best scheduling strategy for each tree, and then select the best result.

Given a general interconnection graph, we show that the problem of extracting the optimal spanning tree is NP-complete. Even worse, we show that there exist heterogeneous networks for which the optimal spanning tree has a throughput which is arbitrarily bad in front of the throughput that can be achieved by the optimal (multiple-path) solution. Still, we introduce and compare several low-complexity heuristics to determine a sub-optimal spanning tree. Fortunately, we observe that the best heuristics do achieve an excellent performance in most experiments.

The rest of the paper is organized as follows. In Section 2 we introduce our base model of communication and computation, and we formally state the master-slave scheduling problem for a general interconnection graph. We provide the optimal solution to this problem, using a linear programming approach. In Section 3 we discuss various extensions, first with several masters, and then with different hypotheses on the overlapping capabilities. Section 4 is theoretically oriented and provides the negative complexity results for the search of an optimal spanning tree: (i) NP-completeness of the problem and (ii) inapproximability of a general graph by any spanning tree. Section 5 deals with the design of five low-cost (polynomial) heuristics to determine a sub-optimal spanning tree. These

heuristics are experimentally compared in Section 6. Fortunately, we are able to report that the best two heuristics achieve very good performance in most cases, despite the negative theoretical predictions. We briefly survey related work in Section 7. Finally, we give some remarks and conclusions in Section 8.

2 The Master-Slave Scheduling Problem

In this section, we formally state the optimization problem to be solved. We start with the architectural model, next we explain how to compute the steady state, and finally we state the master-slave scheduling problem as a linear programming problem to be solved in rational numbers (hence a polynomial complexity).

2.1 Architectural Model

The target architectural/application framework is represented by a node-weighted edge-weighted graph $G = (V, E, w, c)$. Let $p = |V|$ be the number of nodes. Each node $P_i \in V$ represents a computing resource of weight w_i, meaning that node P_i requires w_i units of time to process one task (so the smaller w_i, the faster the processor node P_i). There is a *master* processor, i.e. a node P_m which plays a particular role. P_m initially holds the data for a large (say infinite) collection of independent tasks to be executed. Tasks are atomic, their computation or communication cannot be preempted. A task represents the granularity of the application.

Each edge $e_{ij} : P_i \to P_j$ is labeled by a value c_{ij} which represents the time needed to communicate the data for one task between P_i and P_j, in either direction: we assume that the link between P_i and P_j is bidirectional and symmetric, i.e. that it takes the same amount of time to send (the data for) one task from P_i to P_j than in the reverse direction, from P_j to P_i. If there is no communication link between P_i and P_j we let $c_{ij} = +\infty$, so that $c_{ij} < +\infty$ means that P_i and P_j are neighbors in the communication graph. Note that we can include in c_{ij} the time needed for the receiving processor to return the result to the sending processor when it is finished. For the purpose of computing steady-state behavior, it does not matter what fraction of the communication time is spent sending a problem and what fraction is spent receiving the results. To simplify the exposition, we will henceforth assume that all the time is spent sending the task data, and no time is needed to communicate the results back. We assume that all w_i are positive rational numbers. We disallow $w_i = 0$ since it would permit node P_i to perform an infinite number of tasks, but we allow $w_i = +\infty$; then P_i has no computing power but can still forward tasks to other processors. Similarly, we assume that all c_{ij} are positive rational numbers (or equal to $+\infty$ if there is no link between P_i and P_j).

We state the communication model more precisely: if P_i sends a task to P_j at time-step t, then P_j cannot start executing this task before time-step $t + c_{ij}$, P_j can neither initiate another receive operation nor start the execution of the task before time-step $t + c_{ij}$ (but it can perform a send operation and independent

computation), P_i cannot initiate another send operation before time-step $t + c_{ij}$ (but it can perform a receive operation and independent computation).

2.2 Steady-State Operation

Given the resources of a weighted graph G operating under the base model, we aim at determining the best steady-state scheduling policy. After a start-up phase, we want the resources to operate in a periodic mode. This makes very good sense if there is a large number of tasks to process, as typical applications would require: otherwise why bother dispatching them on the grid?

To formally define the steady-state, we need a couple of notations. Let $n(i)$ denote the index set of the neighbors of processor P_i. During one time unit: α_i is the fraction of time spent by P_i computing, s_{ij} is the fraction of time spent by P_i sending tasks to each neighbor processor P_j, $j \in n(i)$, i.e. for each $e_{ij} \in E$ and r_{ij} is the fraction of time spent by P_i receiving tasks from each neighbor processor P_j, $j \in n(i)$, i.e. for each $e_{ij} \in E$. We search for rational values of all these variables. The first set of constraints is that all variables α_i, s_{ij} and r_{ij} must belong to the interval $[0, 1]$, as they correspond to the activity during one time unit. The second set of constraints is that the number s_{ij}/c_{ij} of tasks sent by P_i to P_j is equal to the number of tasks r_{ji}/c_{ij} received by P_j from P_i:

$$\forall e_{ij} \in E, \quad s_{ij} = r_{ji} \quad (1)$$

Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation

$$\forall i, \quad \sum_{j \in n(i)} s_{ij} \leq 1 \quad (2)$$

Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation

$$\forall i, \quad \sum_{j \in n(i)} r_{ij} \leq 1 \quad (3)$$

Because of the full overlap hypothesis, there is no further constraint on α_i: $0 \leq \alpha_i \leq 1$, and $\alpha_i = 1$ would mean that P_i is kept processing tasks all the time. We have to ensure that the link bandwidth is not exceeded. The constraint translates into:

$$\forall e_{ij} \in E, \quad s_{ij} + r_{ij} \leq 1 \quad (4)$$

The last constraints deals with *conservation laws*: for every processor P_i which is not the master, the number of tasks received by P_i, i.e. $\sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}}$, should be equal to the number of tasks that P_i consumes itself, i.e. $\frac{\alpha_i}{w_i}$, plus the number of tasks forwarded to its neighbors, i.e. $\sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}}$. We derive the equation:

$$\forall i \neq m, \sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}} \quad (5)$$

It is important to understand that Equation (5) really applies to the steady-state operation. We can assume an initialization phase, during which tasks are forwarded to processors, and no computation is performed. Then, during each time-period in steady-state, each processor can simultaneously perform some computations, and send/receive some other tasks. Equation (5) does not hold for the master processor P_m, because it holds an infinite number of tasks. Without loss of generality, we can enforce that $r_{mj} = 0$ for all $j \in n(m)$: the master does not need to receive any task from its neighbors.

The equations above constitute a linear programming problem, whose objective function is the number of tasks consumed within one unit of time, i.e. the throughput $n_{\text{task}}(G) = \sum_i \frac{\alpha_i}{w_i}$. Here is a summary:

Master Slave Scheduling Problem MSSP(G)
Maximize

$$n_{\text{task}}(G) = \sum_{i=1}^{p} \frac{\alpha_i}{w_i},$$

subject to

$$\begin{cases} \forall i, & 0 \leq \alpha_i \leq 1 \\ \forall i, & \forall j \in n(i), 0 \leq s_{ij} \leq 1 \\ \forall i, & \forall j \in n(i), 0 \leq r_{ij} \leq 1 \\ \forall e_{ij} \in E, & s_{ij} = r_{ji} \\ \forall i, & \sum_{j \in n(i)} s_{ij} \leq 1 \\ \forall i, & \sum_{j \in n(i)} r_{ij} \leq 1 \\ \forall e_{ij} \in E, & s_{ij} + r_{ij} \leq 1 \\ \forall i \neq m, & \sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}} \\ \forall j \in n(m), & r_{mj} = 0 \end{cases}$$

Note that we can enforce $\alpha_m = 1$, because the master will keep on processing tasks all the time, but this condition will automatically be fulfilled by the solution.We can now state the first result of this paper:

Theorem 1. *The solution to the previous linear programming problem provides the optimal solution to* MSSP(G).

Because we have a linear programming problem in rational numbers, we obtain rational values for all variables in polynomial time (polynomial in $|V| + |E|$, the size of the heterogeneous platform). When we have the optimal solution, we take the least common multiple of the denominators, and thus we derive an integer period T for the steady-state operation.

3 Extensions

3.1 With Several Masters

The extension for several masters is straightforward. Assume that there are k masters $P_{m_1}, P_{m_2}, \ldots, P_{m_k}$, each holding (the initial data for) a large collection

of tasks. For each index m_q, $1 \leq q \leq k$: suppress equation (5) for $i = m_q$ (the conservation law does not apply to a master), add the constraints $r_{m_q,j} = 0$ for all $j \in m(q)$ and solve the new MSSP(G) problem.

3.2 With Other Models

We rely on the classification proposed by Beaumont et al [2]. A processor can do three kinds of operations: it can perform some computation, it can receive data from its neighbors and send data to its neighbors. The degree of simultaneity between this three actions indicates the level of performance of a processor. For instance a processor which can execute this three operations simultaneously has the highest level of performance, whereas the one which can only do one thing at a time is the less powerfull. The interested reader may find in [1] the proof of the equivalence of these models. It is important to point out that the processor nodes of a platform may well operate under different modes.

4 Spanning Trees

For a general interconnection graph, the solution of the linear program may lead to the use of multiple paths. As already mentioned, it may be of interest to extract the best spanning tree (the one with maximum throughput) out of the graph. Using a tree greatly simplifies the implementation (because of the unique route from the master to any processor). Also, the bandwidth-centric algorithm presented in [2] is local and demand-driven, therefore is very robust to small variations in resources capabilities.

This section provides "negative" results: first, extracting the best tree is NP-Complete. But even if we are ready to pay a high (exponential) cost to determine the best tree, there exist graphs for which the throughput of the best tree is arbitrarily bad in front of the throughput that can be achieved while the whole graph. In practice, however, low-costs heuristics can be derived to determine sub-optimal but efficient spanning trees: (see Sections 5 and 6).

4.1 Finding the Best Spanning Tree

Our aim is to find the spanning tree that maximizes the throughput, i.e. the number of tasks that can be processed within one unit of time at steady state. Formally, we can state the problem as follows. The interested reader may find in [1] the proofs of theorems 2 and 3.

Definition 1 (BEST-TREE(G)). *Let $G = (V, E, w, c)$ be the node-weighted edge-weighted graph representing the architectural framework. Find the tree $T = (V, E', w, c)$, sub-graph of G, rooted at the master, such that the number of tasks $n_{\text{task}}(T)$ that can be processed in steady-state within one time-unit, using only those edges of the tree, is maximized.*

Theorem 2. BEST-TREE(G,α) *is NP-complete [1].*

4.2 Inapproximability of a Graph by a Tree

One natural and interesting question is the following: how bad may the approximation of a graph by a tree be? The following theorem states the inapproximability of a general graph by a tree, with respect to throughput:

Theorem 3. *Given any positive integer K, there exists a graph G such that for any tree T, sub-graph of G and rooted at the master, we have [1]*

$$\frac{n_{task}(G)}{n_{task}(T)} \geq K.$$

The ratio between the number of tasks that can be processed using the graph and the number of tasks that can be processed using any extracted tree can be arbitrary large. Nevertheless, we show in Sections 5 and 6 that despite both the NP-completeness of the search of the best spanning tree, and the inapproximability of a graph by a spanning tree, it is possible to derive very efficient heuristics in practice.

5 Heuristics

In this section, we present several heuristics to extract a spanning tree with the highest possible throughput out of a general interconnection graph $G = (V, E, w, c)$. Given a spanning tree, we do not need the linear programming approach to compute its throughput: instead, we traverse the tree using the bandwidth-centric algorithm of [2], with a cost linear in $(|V|+|E|)$. Several greedy heuristics come to mind. We have selected (and implemented) the following.

5.1 Greedy Heuristics

Naive MST. Given G, we compute the minimum spanning tree [6]. The edge weights are the communication parameters c_{ij}, and we do not use the computation parameters w_i at all. Given the non-oriented tree, we root it at the master and orient the edges accordingly.

Compute Tree. We start from the master and take all the edges connecting to its neighbors. We sort these neighbors by non-decreasing w_i (faster processors first). For each neighbor P_i in sorted order, we consider its own neighbors and add the corresponding edge if that neighbor does not already belong to the tree. The process goes on until all nodes are included. This is a breadth-first traversal to grow the tree, where the more powerful neighbors of the current node are processed first.

C-to-C Tree. This heuristic grows the tree similarly as the previous one, except that the sorted order is by non-decreasing values of the communication-to-computation ratios c_{ij}/w_j.

BW-centric Tree. This heuristic is a variant of the **Compute Tree** heuristic: if node P_i is the current node, its neighbors P_j are still added according to the order of non-decreasing w_j, but only while the bandwidth-centric condition $\sum_j \frac{c_{ij}}{w_j} \leq 1$ holds. The idea is that the last neighbors will not be added to the tree if the bandwidth is saturated. At the end of the procedure there may remain isolated nodes, which we then connect to their closest neighbor (smallest value of the edge weight).

5.2 Heuristic Based on the Linear Program

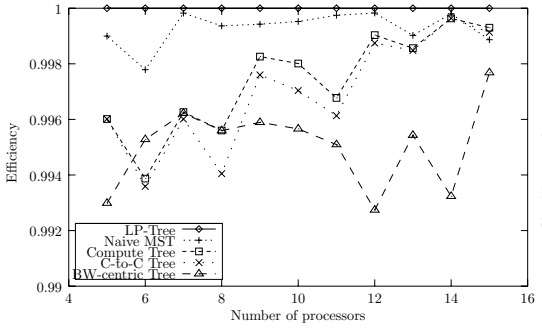
Our last heuristic is more costly, because it requires to solve the linear program for the initial interconnection graph. Once we have the solution, we weight each edge e_{ij} by the value $\frac{s_{ij}}{c_{ij}}$, which represents the average number of tasks which transit on the edge each second (in fact, because edges are bidirectional, we use $\frac{|s_{ij}-s_{ji}|}{c_{ij}}$). Given these weights, we extract a minimum spanning tree, which we call the **LP Tree**, and we compute its throughput as before.

6 Experiments

We have developed a software simulator that executes the heuristic algorithms of Section 5 and computes the throughput for each of them. The inputs of the simulator are the number of nodes in the graph, the minimum degree, the maximum degree, the median degree, a probability function for the communication and the computation costs. A random connected graph based on these parameters is generated. For each graph, the throughput for each heuristic is computed, and is compared to the optimal throughput that can be reached using the whole graph. The following results are averaged values on 50 random graphs whose nodes minimum degree is equal to 3, maximum degree is equal to 5 and average degree is equal to 4. The number of vertices range from 5 to 15. In the following simulation depicted in Fig 1, the probability functions for the communication and the computation costs follow a uniform distribution on the interval $[25, 35]$. We depict the average ratio between the throughput of each heuristic and the optimal throughput of the whole graph. Note that **LP-Tree** and **Naive MST** are the most efficient heuristic and lead to trees whose throughput is very close to the optimal throughput of the whole graph. The reader may refer to [1] to find more simulation results.

7 Related Problems

Scheduling task graphs on heterogeneous platforms. Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors. The reader may refer to [1] to find a complete discussion of scheduling strategies on heterogeneous platforms.



Both computation and communication costs range from 25 to 35. LP-Tree always leads to an optimal tree, and Naive MST behaves surprisingly well.

Fig. 1. Efficiency of the five heuristics : LP-Tree, Naive MST, Compute Tree, C-to-C Tree and BW-centric Tree.

Collective communications on heterogeneous platforms. Several papers deal with the complexity of collective communications on heterogeneous platforms: broadcast and multicast are discussed in [5,12], gather in [10].

Master-slave on the computational grid. Master-slave scheduling on the grid can be based on a network-flow approach [15] or on an adaptive strategy [11].

8 Conclusion

In this paper, we have dealt with master-slave tasking on heterogeneous platforms. We have shown how to determine the best steady-state scheduling strategy for a general interconnection graph, using a linear programming approach. On one hand, we have derived negative theoretical results, namely that general interconnection graphs may be arbitrarily more powerful than spanning trees, and that determining the best spanning tree is NP-Complete.
On the other hand, we have proposed several low-costs heuristics that achieve very good performances on a wide range of simulations. These positive experiments show that in practice, it is safe to rely on spanning trees to implement master-slave tasking.

This work can be extended in the following two directions:

- From a theoretical point of view, it could be interesting to try to solve the complete scheduling problem associated to a general interconnection graph: instead of optimizing the steady-state throughput, how to maximize the total number of tasks processed within T time-units, for any time-bound T? Partial results are available in [3,4], but the general problem looks quite challenging.
- On the practical side, we need to run actual experiments rather than simulations. Indeed, it would be interesting to capture actual architecture and application parameters, and to compare heuristics on a real-life problem.

References

1. C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor grids. Technical Report 2002-12, LIP, ENS Lyon, France, March 2002.
2. O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2002*. IEEE Computer Society Press, 2002. Extended version available as LIP Research Report 2001-25.
3. O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In D.S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, editors, *Cluster'2001*, pages 419–426. IEEE Computer Society Press, 2001. Extended version available as LIP Research Report 2001-13.
4. O. Beaumont, A. Legrand, and Y. Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. Technical Report 2002-07, LIP, ENS Lyon, France, February 2002.
5. P.B. Bhat, V.K. Prasanna, and C.S. Raghavendra. Efficient collective communication in distributed heterogeneous systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*. IEEE Computer Society Press, 1999.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
7. James Cowie, Bruce Dodson, R.-Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Joerg Zayer. A world wide number field sieve factoring record: on to 512 bits. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - Asiacrypt '96*, volume 1163 of *LNCS*, pages 382–394. Springer Verlag, 1996.
8. Entropia. URL: `http://www.entropia.com`.
9. J.P Goux, S.Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society Press, 2000.
10. J.-I. Hatta and S. Shibusawa. Scheduling algorithms for efficient gather operations in distributed heterogeneous systems. In *2000 International Conference on Parallel Processing (ICPP'2000)*. IEEE Computer Society Press, 2000.
11. E. Heymann, M.A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.
12. R. Libeskind-Hadas, J.R.K. Hartline, P. Boothe, G. Rae, and J. Swisher. On multicast algorithms for heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 61(11):1665–1679, 2001.
13. Prime. URL: `http://www.mersenne.org`.
14. SETI. URL: `http://setiathome.ssl.berkeley.edu`.
15. G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.
16. J.B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.