Scheduling divisible loads with return messages on heterogeneous master-worker platforms

Olivier Beaumont[1], Loris Marchal[2], and Yves Robert[2]

[1] LaBRI, UMR CNRS 5800, Bordeaux, France
Olivier.Beaumont@labri.fr
[2] LIP, UMR CNRS-INRIA-UCBL 5668, ENS Lyon, France
{Loris.Marchal | Yves.Robert}@ens-lyon.fr

Abstract In this paper, we consider the problem of scheduling divisible loads onto an heterogeneous star platform, with both heterogeneous computing and communication resources. We consider the case where the workers, after processing the tasks, send back some results to the master processor. This corresponds to a more general framework than the one used in many divisible load papers, where only forward communications are taken into account. To the best of our knowledge, this paper constitutes the first attempt to derive optimality results under this general framework (forward and backward communications, heterogeneous processing and communication resources). We prove that it is possible to derive the optimal solution both for LIFO and FIFO distribution schemes. Nevertheless, the complexity of the general problem remains open: we also show in the paper that the optimal distribution scheme may be neither LIFO nor FIFO.

1 Introduction

This paper deals with scheduling divisible load applications on heterogeneous platforms. As their name suggests, divisible load applications can be divided among worker processors arbitrarily, i.e. into any number of independent pieces. This corresponds to a perfectly parallel job: any sub-task can itself be processed in parallel, and on any number of workers. In practice, the *Divisible Load Scheduling* model, or DLS model, is an approximation of applications that consist of large numbers of identical, low-granularity computations.

Quite naturally, we target a master-worker implementation where the master initially holds (or generates data for) a large amount of work that will be executed by the workers. In the end, results will be returned by the workers to the master. Each worker has a different computational speed, and each master-worker link has a different bandwidth, thereby making the platform fully heterogeneous. The scheduling problem is first to decide how many load units the master sends to each worker, and in which order. After receiving its share of the data, each worker executes the corresponding work and returns the results to the master. Again, the ordering of the return messages must be decided by the scheduler.

The DLS model has been widely studied in the last several years, after having been popularized by the landmark book [7]. The DLS model provides a practical framework for the mapping of independent tasks onto heterogeneous platforms, and has been applied to a large spectrum of scientific problems. From a theoretical standpoint, the success of the DLS model is mostly due to its analytical tractability. Optimal algorithms and closed-form formulas exist for important instances of the divisible load problem. A famous example is the closed-form formula given in [4,7] for a bus network. The hypotheses are the following: (i) the master distributes the load to the workers, but no results are returned to the master; (ii) a linear cost model is assumed both for computations and for communications (see Section 2.1); and (iii) all master-worker communication links have same bandwidth (but the workers have different processing speeds). The proof to derive the closed-form formula proceeds in several steps: it is shown that in an optimal solution: (i) all workers participate in the computation, then that (ii) they never stop working after having received their data from the master, and finally that (iii) they all terminate the execution of their load simultaneously. These conditions give rise to a set of equations from which the optimal load assignment α_i can be computed for each worker P_i.

Extending this result to a star network (with different master-worker link bandwidths), but still (1) without return messages and (2) with a linear cost model, has been achieved only recently [5]. The proof basically goes along the same steps as for a bus network, but the main additional difficulty was to find the optimal ordering of the messages from the master to the workers. It turns out that the best strategy is to serve workers with larger bandwidth first, independently of their computing power.

The next natural step is to include return messages in the picture. This is very important in practice, because in most applications the workers are expected to return some results to the master. When no return messages are assumed, it is implicitly assumed that the size of the results to be transmitted to the master after the computation is negligible, and hence has no (or very little) impact on the whole DLS problem. This may be realistic for some particular DLS applications, but not for all of them. For example suppose that the master is distributing files to the workers. After processing a file, the worker will typically return results in the form of another file, possibly of shorter size, but still non-negligible. In some situations, the size of the return message may even be larger than the size of the original message: for instance the master initially scatters instructions on some large computations to be performed by each worker, such as the generation of several cryptographic keys; in this case each worker would receive a few bytes of control instructions and would return longer files containing the keys.

Because it is very natural and important in practice, several authors have investigated the problem with return messages: see the papers [3,8,9,2,1]. However, all the results obtained so far are very partial. Intuitively, there are hints that suggest that the problem with return results is much more complicated. The first hint lies in the combinatorial space that is open for searching the best solution. There is no reason for the ordering of the initial messages sent by the

master to be the same as the ordering for the messages returned to the master by the workers after the execution. In some situations a FIFO strategy (the worker first served by the master is the first to return results, and so on) may be preferred, because it provides a smooth and well-structured pipelining scheme. In other situations, a LIFO strategy (the other way round, first served workers are the last to return results) may provide better results, because faster workers would work a longer period if we serve them first and they send back their results last. True, but what if these fast workers have slow communication links? In fact, and here comes the second hint, it is not even clear whether all workers should be enrolled in the computation by the master. This is in sharp contrast to the case without return messages, where it is obvious that all workers should participate. To the best of our knowledge, the complexity of the problem remains open, despite the simplicity of the linear cost model. In [1], Adler, Gong and Rosenberg show that all FIFO strategies are equally performing on a bus network, but even the analysis of FIFO strategies is an open problem on a star network.

The main contributions of this paper are the characterization of the best FIFO and LIFO strategies on a star network, together with an experimental comparison of them. While the study of LIFO strategies nicely reduces to the original problem without return messages, the analysis of FIFO strategies turns out to be more involved; in fact, the optimal FIFO solution may well not enroll all workers in the computations. Admittedly, the complexity of the DLS problem with return messages remains open: there is no a priori reason that either FIFO or LIFO strategies would be superior to solutions where the ordering of the initial messages and that of return messages are totally uncorrelated (and we give an example of such a situation in Section 2). Still, we believe that our results provide an important step in the understanding of this difficult problem, both from a theoretical and practical perspective. Indeed, we have succeeded in characterizing the best FIFO and LIFO solutions, which are the most natural and easy-to-implement strategies. Due to space limitations, the overview of related work is not included in this paper, please refer to the extended version [6]. Similarly, all proofs are omitted, but they are all detailed in [6].

2 Framework

2.1 Problem parameters

A *star network* $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$ is composed of a master P_0 and of p workers P_i, $1 \leq i \leq p$. There is a communication link from the master P_0 to each worker P_i. In the linear cost model, each worker P_i has a (relative) computing power w_i: it takes $X.w_i$ time units to execute X units of load on worker P_i. Similarly, it takes $X.c_i$ time units to send the initial data needed for computing X units of load from P_0 to P_i, and $X.d_i$ time units to return the corresponding results from P_i to P_0. Without loss of generality we assume that the master has no processing capability (otherwise, add a fictitious extra worker paying no communication cost to simulate computation at the master). Note that a

bus network is a star network such that all communication links have the same characteristics: $c_i = c$ and $d_i = d$ for each worker P_i, $1 \leq i \leq p$.

It is natural to assume that the quantity $\frac{d_i}{c_i}$ is a constant z that depends on the application but not on the selected worker. In other words, workers who communicate faster with the master for the initial message will also communicate faster for the return message. In the following, we keep using both values d_i and c_i, because many results are valid even without the relation $d_i = zc_i$, and we explicitly mention when we use this relation.

Finally, we use the standard model in DLS problem for communications: the master can only send data to, and receive data from, a single worker at a given time-step. A given worker cannot start execution before it has terminated the reception of the message from the master; similarly, it cannot start sending the results back to the master before finishing the computation. However, there is another classic hypothesis in DLS papers which we do not enforce, namely that there is no idle time in the operation of each worker. Under this assumption, a worker starts computing immediately after having received its initial message, which is no problem, but also starts returning the results immediately after having finished its computation: this last constraint does reduce the solution space arbitrarily. Indeed, it may well prove useful for a worker P_i to stay idle a few steps before returning the results, waiting for the master to receive the return message of another worker $P_{i'}$. Of course we could have given more load to P_i to prevent it from begin idle, but this would have implied a longer initial message, at the risk of delaying the whole execution scheme. Instead, we will tackle the problem in its full generality and allow for the possibility of idle times (even if we may end by proving that there is no idle time in the optimal solution).

The objective function is to maximize the number of load units that are processed within T time-units. Let α_i be the number of load units sent to, and processed by, worker P_i within T time-units. Owing to the linear cost model, the quantity $\frac{\sum_{i=1}^{p} \alpha_i}{T} = \rho$ does not depend on T (see Section 2.2 for a proof), and corresponds to the achieved throughput, which we aim at maximizing.

2.2 Linear program for a given scenario

Given a star platform with p workers, and parameters w_i, c_i, d_i, $1 \leq i \leq p$, how can we compute the optimal throughput? First we have to decide which workers are enrolled. Next, given the set of participating workers, we have to decide for the ordering of the initial messages. Finally we have to decide for the ordering of the return messages. Altogether, there is a finite (although exponential) number of scenarios, where a *scenario* refers to a schedule with a given set of participating workers and a fixed ordering of initial and return messages. Then, the next question is: how can we compute the throughput for a given scenario?

Without loss of generality, we can always perform all the initial communications as soon as possible. In other words, the master sends messages to the workers without interruption. If this was not the case, we would simply shift ahead some messages sent by the master, without any impact on the rest of the

schedule. Obviously, we can also assume that each worker initiates its computation as soon as it has received the message from the master. Finally, we can always perform all the return communications as late as possible. In other words, once the master starts receiving data back from the first worker, it receives data without interruption until the end of the whole schedule. Again, if this was not the case, we would simply delay the first messages received by the master, without any impact on the rest of the schedule. Note that idle times can still occur in the schedule, but only between the end of a worker's computation and the date at which it starts sending the return message back to the master.

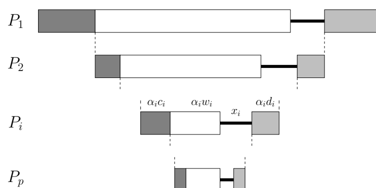


Figure 1. LIFO strategy. Dark grey rectangles (of length $\alpha_q c_q$) represent the initial messages destined to the workers. White rectangles (of length $\alpha_q w_q$) represent the computation on the workers. Light grey rectangles (of length $\alpha_q d_q$) represent the return messages back to the master. Bold lines (of length x_q) represent the idle time of the workers.

The simplest approach to compute the throughput ρ for a given scenario is to solve a linear program. For example, assume that we target a LIFO solution involving all processors, with the ordering $P_1, P_2, \dots, P_p$, as outlined in Figure 1. With the notations of Section 2.1 (parameters w_i, c_i, d_i and unknowns α_i, ρ), worker P_i: (i) starts receiving its initial message at time $t_i^{\text{recv}} = \sum_{j=1}^{i-1} \alpha_j c_j$; (ii) starts execution at time $t_i^{\text{recv}} + \alpha_i c_i$; (iii) terminates execution at time $t_i^{\text{term}} = t_i^{\text{recv}} + \alpha_i c_i + \alpha_i w_i$; (iv) starts sending back its results at time $t_i^{\text{back}} = T - \sum_{j=1}^{i} \alpha_j d_j$. Here T denotes the total length of the schedule. The idle time of P_i is $x_i = t_i^{\text{back}} - t_i^{\text{term}}$, and this quantity must be nonnegative. We derive a linear equation for P_i:

$$T - \sum_{j=1}^{i} \alpha_j d_j \geq \sum_{j=1}^{i-1} \alpha_j c_j + \alpha_i c_i + \alpha_i w_i.$$

Together with the constraints $\alpha_i \geq 0$, we have assembled a linear program, whose objective function is to maximize $\rho(T) = \sum_{i=1}^{p} \alpha_i$. In passing, we check that the value of $\rho(T)$ is indeed proportional to T, and we can safely define $\rho = \rho(1)$ as mentioned before. We look for a rational solution of the linear program, with rational (not integer) values for the quantities α_i and ρ, hence we can use standard tools like Maple or MuPAD.

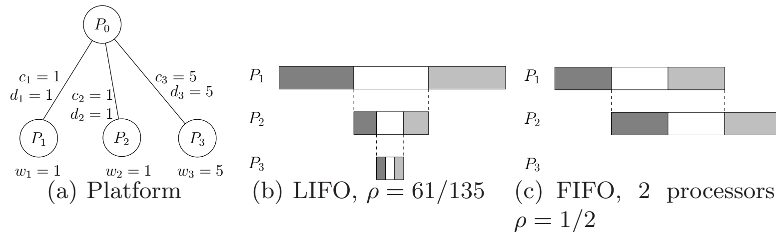
Obviously, this linear programming approach can be applied for any permutation of initial and return messages, not just LIFO solutions as in the above

example. Note that it may well turn out that some α_i is zero in the solution returned by the linear program, which means that P_i is not actually involved in the schedule. This observation reduces the solution space: we can *a priori* assume that all processors are participating, and solve a linear program for each pair of message permutations (one for the initial messages, one for the return messages). The solution of the linear program will tell us which processors are actually involved in the optimal solution for this permutation pair.

For a given scenario, the cost of this linear programming approach may be acceptable. However, as already pointed out, there is an exponential number of scenarios. Worse, there is an exponential number of LIFO and FIFO scenarios, even though there is a single permutation to try in these cases (the ordering of the return messages is the reverse (LIFO) or the same (FIFO) as for the initial messages). The goal of Sections 3 and 4 is to determine the best LIFO and FIFO solution in polynomial time.

2.3 Counter-examples

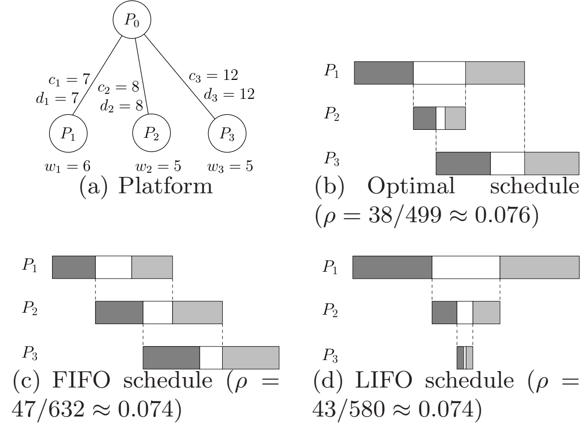
In Figure 2 we outline an example where not all processors participate in the optimal solution. The platform has three workers, as shown in Figure 2(a). The best throughput that can be achieved using all the three workers is obtained via the LIFO strategy represented in Figure 2(b), and is $\rho = 61/135$. However, the FIFO solution which uses only the first two workers P_1 and P_2 achieves a better throughput $\rho = 1/2$. To derive these results, we have used the linear programming approach for each of the 36 possible permutation pairs. It is very surprising to see that the optimal solution does not involve all workers under a linear cost model (and to the best of our knowledge this is the first known example of such a behavior).



(a) Platform (b) LIFO, $\rho = 61/135$ (c) FIFO, 2 processors, $\rho = 1/2$

Figure 2. The best schedule with all the three workers (shown in (b)) achieves a lower throughput than when using only the first two workers (as shown in (c)).

Next, in Figure 3, we outline an example where the best throughput is achieved using neither a FIFO nor a LIFO approach. Instead, the optimal solution uses the initial ordering (P_1, P_2, P_3) and the return ordering (P_2, P_1, P_3). Again, we have computed the throughput of all possible permutation pairs using the linear programming approach.



(a) Platform

(b) Optimal schedule ($\rho = 38/499 \approx 0.076$)

(c) FIFO schedule ($\rho = 47/632 \approx 0.074$)

(d) LIFO schedule ($\rho = 43/580 \approx 0.074$)

Figure 3. An example where the optimal solution is neither FIFO nor LIFO.

3 LIFO strategies

In this section, we concentrate on LIFO strategies, where the processor that receives the first message is also the last processor that sends its results back to the master, as depicted in Figure 1. We keep the notations used in Section 2.2, namely w_i, c_i, d_i and α_i for each worker P_i.

In order to determine the optimal LIFO ordering, we need to answer the following questions: What is the subset of participating processors? What is the ordering of the initial communications? What is the idle time x_i of each participating worker? The following theorem answers these questions and provides the optimal solution for LIFO strategies (see [6] for the proof):

Theorem 1. *In the optimal LIFO solution, then: (i) all processors participate to the execution; (ii) initial messages must be sent by non-decreasing values of* $c_i + d_i$*; and (iii) there is no idle time, i.e.* $x_i = 0$ *for all* i*. Furthermore, the corresponding throughput can be determined in linear time* $O(p)$*.*

4 FIFO strategies

In this section, we concentrate on FIFO strategies, where the processor that receives the first message is also the first processor to send its results back to the master. We keep the notations used in Sections 2.2 and 3, namely w_i, c_i, d_i and α_i for each worker P_i. The analysis of FIFO strategies is much more difficult than the analysis of LIFO strategies: we will show that not all processors are enrolled in the optimal FIFO solution. In this section, we assume that $d_i = zc_i$ for $1 \leq i \leq p$. The proof of the following theorem is long and technical, see [6] for more details:

Theorem 2. *In the optimal FIFO solution, then: (i) initial messages must be sent by non-decreasing values of $c_i + d_i$; (ii) the set of participating processors is composed of the first q processors for the previous ordering, where q can be determined in linear time; and (iii) there is no idle time, i.e. $x_i = 0$ or all i. Furthermore, the optimal LIFO solution and the corresponding throughput can be determined in linear time $O(p)$.*

5 Simulations

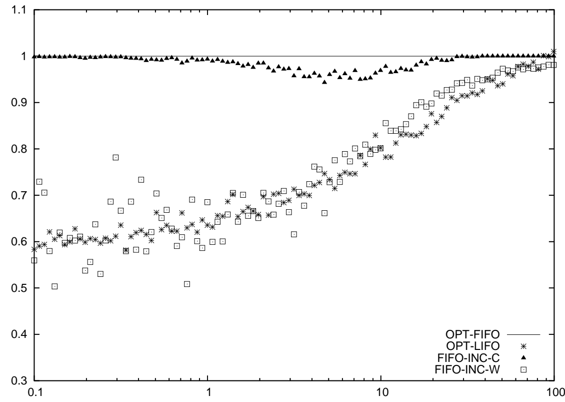
In this section, we present the results of some simulations conducted with the LIFO and FIFO strategies. We cannot compare these results against the optimal schedule, since we are not able to determine the optimal solution as soon as the number of workers exceeds a few units. For instance, for a platform with 100 workers, we would need to solve $(100!)^2$ linear programs of 100 unknowns (one program for each permutation pair). Rather than computing the solution for all permutation pairs, we use the optimal FIFO algorithm as a basis for the comparisons. The algorithms tested in this section are the following: optimal FIFO solution, as determined in Section 4, called OPT-FIFO; optimal LIFO solution, as determined in Section 3, called OPT-LIFO; a FIFO heuristic using all processors, sorted by non-decreasing values of c_i (faster communicating workers first), called FIFO-INC-C; a FIFO heuristic using all processors, sorted by non-decreasing values of w_i (faster computing workers first), called FIFO-INC-W.

We present the relative performance of these heuristics on a master/worker platform with 100 workers. For these experiments, we chose $z = 0.8$, meaning that the returned data represents 80% of the input data. The performance parameters (communication and computation costs) of each worker may vary from 50% around an average value. The ratio of the average computation cost over the average communication cost (called the *w/c-ratio*) is used to distinguish between the experiments, as the behavior of the heuristics highly depends on this parameter.

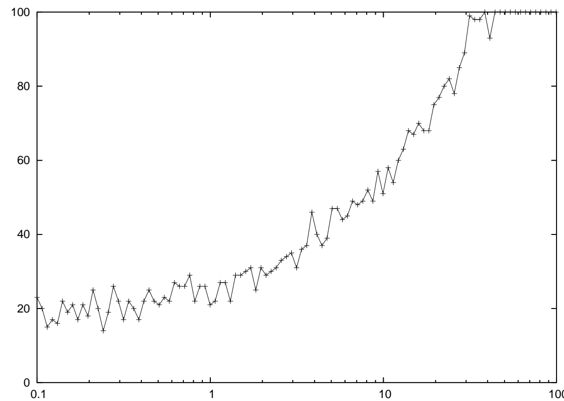
Figure 4(a) presents the throughput of the different heuristics for a w/c-ratio going from 1/10 to 100. These results are normalized so that the optimal FIFO algorithm always gets a throughput of 1. We see that both OPT-FIFO and FIFO-INC-C give good results. The other heuristics (FIFO-INC-W and OPT-LIFO) perform not so well, except when the w/c-ratio is high: in this case, communications have no real impact on the schedule, and almost all schedules may achieve good performances.

In Section 4, we showed that using all processors is not always a good choice. In Figure 4(b) we plot the number of processors used by the OPT-FIFO algorithm for the previous experiments: for small values of the w/c-ratio, a very small fraction of the workers is enrolled in the optimal schedule. Finally, Figure 4(c) presents the relative performance of all heuristics when the size of the data returned is the same as the size of the input data ($z = 1$). With the exception of this new hypothesis ($z = 1$ instead of $z = 0.8$), the experimental settings are the same as for Figure 4(a). We show that for a w/c-ratio less than 10, only the

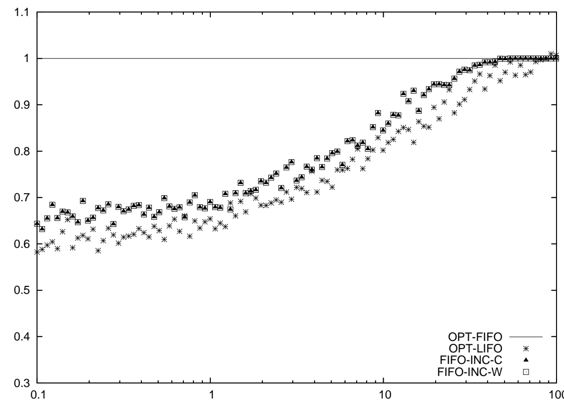
OPT-FIFO algorithm gives good performance: the FIFO-INC-C heuristic is no longer able to reach a comparable throughput. We also observe that when $z = 1$ the ordering of the workers has no importance: FIFO-INC-C and FIFO-INC-W are FIFO strategies involving all the workers but in different orders, and give exactly the same results.



4(a): Performance of the heuristics (optimal FIFO schedule), for different w/c-ratios



4(b): Number of workers (optimal FIFO schedule), for different w/c-ratios



4(c): Performance of the heuristics (optimal FIFO schedule), when $c_i = d_i$ ($z = 1$), for different w/c-ratios

6 Conclusion

In this paper we have dealt with divisible load scheduling on a heterogeneous master-worker platform. We have shown that including return messages from the workers after execution, although a very natural and important extension in practice, leads to considerable difficulties. These difficulties were largely unexpected, because of the simplicity of the linear model.

We have not been able to fully assess the complexity of the problem, but we have succeeded in characterizing the optimal LIFO and FIFO strategies, and in providing an experimental comparison of these strategies against simpler greedy approaches. Future work must be devoted to investigate the general case, i.e. using two arbitrary permutation orderings for sending messages from, and returning messages to, the master. This seems to be a very combinatorial and complicated optimization problem.

References

1. M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.
2. D. Altilar and Y. Paker. Optimal scheduling algorithms for communication constrained parallel processing. In *Euro-Par 2002*, LNCS 2400, pages 197–206. Springer Verlag, 2002.
3. G. Barlas. Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees. *IEEE Trans. Parallel Distributed Systems*, 9(5):429–441, 1998.
4. S. Bataineh, T. Hsiung, and T.G.Robertazzi. Closed form solutions for bus and tree networks of processors load sharing a divisible job. *IEEE Transactions on Computers*, 43(10):1184–1196, Oct. 1994.
5. O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: results and open problems. *IEEE Trans. Parallel Distributed Systems*, 16(3):207–218, 2005.
6. O. Beaumont, L. Marchal, and Y. Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. Research Report 2005-21, LIP, ENS Lyon, France, may 2005. Available at www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-21.pdf.
7. V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.
8. M. Drozdowski and P. Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. In *Proceedings of Euro-Par 2000: Parallel Processing*, LNCS 1900, pages 311–319. Springer, 2000.
9. A. L. Rosenberg. Sharing partitionable workloads in heterogeneous NOws: greedier is not better. In *Cluster Computing 2001*, pages 124–131. IEEE Computer Society Press, 2001.