Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Steady-State Scheduling on Heterogeneous Clusters: Why and How?

Olivier Beaumont,
Arnaud Legrand,
Loris Marchal,
Yves Robert

March 2004

Research Report N° 2004-11

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Steady-State Scheduling on Heterogeneous Clusters: Why and How?

Olivier Beaumont, Arnaud Legrand, Loris Marchal, Yves Robert

March 2004

Abstract

In this paper, we consider steady-state scheduling techniques for heterogeneous systems, such as clusters and grids. We advocate the use of steady-state scheduling to solve a variety of important problems, which would be too difficult to tackle with the objective of makespan minimization. We give a few successful examples before discussing the main limitations of the approach.

Keywords: Scheduling, steady-state, heterogeneous platforms

Résumé

Nous nous intéressons ici aux techniques d'ordonnancement en régime permanent pour les plates-formes hétérogènes de type *"grille de calcul"*. Nous montrons l'utilité de l'approche en régime permanent pour résoudre un certain nombre de problèmes importants, pour lequels chercher à minimmiser le temps d'exécution serait trop complexe. Nous donnons quelques exemples où cette technique a été couronnée de succès, avant de présenter ses principales limites.

Mots-clés: Ordonnancement, régime permanent, plates-formes hétérogènes

1 Introduction

Scheduling computational tasks on a given set of processors is a key issue for high-performance computing. The traditional objective of scheduling algorithms is makespan minimization: given a task graph and a set of computing resources, find a mapping of the tasks onto the processors, and order the execution of the tasks so that: (i) task precedence constraints are satisfied; (ii) resource constraints are satisfied; and (iii) a minimum schedule length is provided. However, makespan minimization turned out to be NP-hard in most practical situations [17, 1]. The advent of more heterogeneous architectural platforms is likely to even increase the computational complexity of the process of mapping applications to machines.

An idea to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective. Instead of aiming at the absolute minimization of the execution time, why not consider asymptotic optimality? After all, the number of tasks to be executed on the computing platform is expected to be very large: otherwise why deploy the corresponding application on computational grids? To state this informally: if there is a nice (meaning, polynomial) way to derive, say, a schedule whose length is two hours and three minutes, as opposed to an optimal schedule that would run for only two hours, we would be satisfied. And if the optimal schedule is not known, there remains the possibility to establish a lower bound, and to assess the polynomial schedule against it.

This approach has been pioneered by Bertsimas and Gamarnik [9]. Steady-state scheduling allows to relax the scheduling problem in many ways. Initialization and clean-up phases are neglected. The initial integer formulation is replaced by a continuous or rational formulation. The precise ordering and allocation of tasks and messages are not required, at least in the first step. The main idea is to characterize the activity of each resource during each time-unit: which (rational) fraction of time is spent computing, which is spent receiving or sending to which neighbor. Such activity variables are gathered into a linear program, which includes conservation laws that characterize the global behavior of the system. We give a few examples below.

The rest of the paper is organized as follows. section 2 summarizes the assumptions on the platform model. Three examples of linear programs characterizing steady-state operation are given in section 3. The target problems are master-slave tasking, pipelined scatter operations and pipelined multicast operations. For the first two problems, the solution of the linear program provides all the information needed to reconstruct a periodic schedule which is asymptotically optimal, as explained in section 4. For the third problem, the situation is more complicated: we discuss this in section 4.3. section 5 aims at discussing several extensions, as well as various limitations, of the steady-state approach. Finally, we state some concluding remarks in section 6.

2 Platform model

The target architectural framework is represented by a node-weighted edge-weighted graph $G = (V, E, w, c)$, as illustrated in Figure 1. 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 time-steps to process one computational unit (so the smaller w_i, the faster the processor node P_i). Each edge $e_{ij} : P_i \rightarrow P_j$ is labeled by a value c_{ij} which represents the time needed to communicate one data unit from P_i to P_j (communication links are oriented). 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 computations, but we allow $w_i = +\infty$; then P_i has no computing power but can still forward data 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

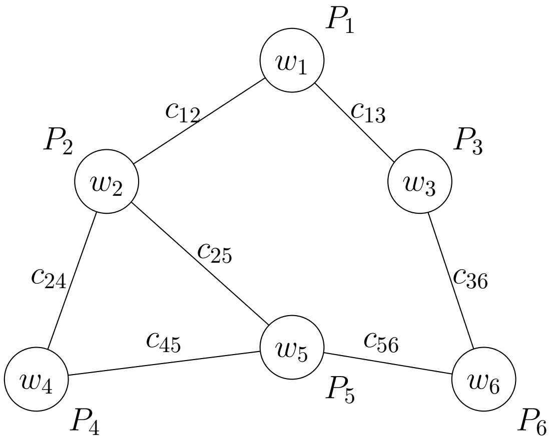


Figure 1: A graph labeled with node (computation) and edge (communication) weights.

and P_j).

Our favorite scenario for the operation mode of the processors is the *full overlap, single-port model* for both incoming and outgoing communications. In this model, a processor node can simultaneously receive data from one of its neighbors, perform some (independent) computation, and send data to one of its neighbors. At any given time-step, there are at most two communications involving a given processor, one sent and the other received. We state the communication model more precisely: if P_i sends a data of size L to P_j at time-step t, then: (i) P_j cannot start executing or sending this task before time-step $t' = t + c_{ij} \cdot L$; (ii) P_j can not initiate another receive operation before time-step t' (but it can perform a send operation and independent computation); and (iii) P_i cannot initiate another send operation before time-step t' (but it can perform a receive operation and independent computation).

Altogether, the model is quite simple: linear costs for computing and communicating, full computation-communication overlap, one-port constraint for sending and receiving. However, no specific assumption is made on the interconnection graph, which may well include cycles and multiple paths, and contention over communication links is taken into account.

3 Linear programs

3.1 Master-slave tasking

In this problem, a special processor $P_m \in V$ (the master) initially holds a large collection of independent, identical tasks to be allocated on the platform. Think of a task as being associated with a file that contains all the data required for the execution of the task. The question for the master is to decide which tasks to execute itself, and how many tasks (i.e. task files) 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. During one time unit, α_i is the fraction of time spent by P_i computing, and s_{ij} is the fraction of time spent by P_i sending tasks to P_j, for $e_{ij} \in E$.

The steady-state equations are summarized in the following linear program:

STEADY-STATE MASTER SLAVE SSMS(G)

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

subject to

$$\begin{cases} \forall i, & 0 \leqslant \alpha_i \leqslant 1 \\ \forall e_{ij} \in E, & 0 \leqslant s_{ij} \leqslant 1 \\ \forall i, & \sum_{j|e_{ij}\in E} s_{ij} \leqslant 1 \\ \forall i, & \sum_{j|e_{ji}\in E} s_{ji} \leqslant 1 \\ \forall e_{jm} \in E, & s_{jm} = 0 \\ \forall i \neq m, & \sum_{j|e_{ji}\in E} \frac{s_{ji}}{c_{ji}} = \frac{\alpha_i}{w_i} + \sum_{j|e_{ij}\in E} \frac{s_{ij}}{c_{ij}} \end{cases}$$

The third and fourth equations enforce the one-port constraints. The fifth equation states that the master does not receive anything. The last equation is the conservation law: the number of tasks received by P_i every time-unit is equal to the number of tasks processed by P_i, plus the number of tasks sent to its neighbors. It is important to see that this equation only holds in steady-state mode. Finally, the objective function is to maximize the number of tasks executed over the platform.

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, during which we execute $T \cdot n_{\text{task}}(G)$ tasks. Because any periodic schedule obeys the equations of the linear program, the previous number is an upper bound of what can be achieved in steady-state mode. It remains to determine whether this bound is tight or not: see section 4.1.

3.2 Pipelined scatter operations

In a scatter operation, one processor P_{source} has to send distinct messages to a set of target processors $\mathcal{P}_{\text{target}} = \{P_0, \ldots, P_N\} \subset V$. In the pipelined version of the problem, P_{source} performs a series of scatter operations, i.e. consecutively sends a large number of different messages to the set $\mathcal{P}_{\text{target}}$.

Let m_k be the type of messages whose destination is P_k, and $send(i,j,k)$ be the fractional number of messages of type m_k which are sent on the edge e_{ij} within a time-unit. Finally, let s_{ij} be the fraction of time spent by P_i sending messages to P_j. We derive the following linear program:

STEADY-STATE PIPELINED SCATTER SSPS(G)

Maximize TP,

subject to

$$\begin{array}{l} \forall i,j, \;\; 0 \leqslant s_{ij} \leqslant 1 \\ \forall i, \;\; \sum_{j|e_{ij}\in E} s_{ij} \leqslant 1 \\ \forall i, \;\; \sum_{j|e_{ji}\in E} s_{ji} \leqslant 1 \\ \forall i,j, \;\; s_{ij} = \sum_k send(i,j,k) \times c_{ij} \\ \forall i, k, k \neq i, \\ \quad \sum_{j|e_{ji}\in E} send(j,i,k) = \sum_{j|e_{ij}\in E} send(i,j,k) \\ \forall P_k \in \mathcal{P}_{\text{target}}, \;\; \sum_{j|e_{jk}\in E} send(j,k,k) = \text{TP} \end{array}$$

As before, the first equations deal with one-port constraints. The fifth equation is the conservation law: a processor forwards all the messages which it receives and whose final destination is another processor. The last equation states that each target processor receives the right number of different messages. This number is indeed the objective function to be maximized. Again, TP is an upper bound of the throughput that can be achieved in steady-state mode.

3.3 Pipelined multicast operations

It looks simpler to multicast than to scatter: indeed, the former operation is the restriction of the latter to the case where all messages are identical. However, we do not know how to write a linear program which would adequately bound the throughput of pipelined multicast operations. For the scatter problem, P_{source} sends messages $x_k^{(t)}$ to each $P_k \in \mathcal{P}$, where (t) denotes the temporal index of the multicast operation. For the multicast problem, all messages are the same for a given operation: $x_k^{(t)} = x_{k'}^{(t)} = x^{(t)}$. Nothing prevents us to use the previous linear program, but the formulation now is pessimistic. If two different messages are sent along a given edge, we do have to sum up the communication times, but if they are the same, there is no reason to count the communication time twice. In other words, we may want to replace the equation $s_{ij} = \sum_k send(i, j, k) \times c_{ij}$ by the equation $s_{ij} = \max_k send(i, j, k) \times c_{ij}$. However, this approach may be too optimistic, as it may well not be possible that messages can be forwarded into groups that obey the new equation: see the discussion in section 4.3.

4 Reconstructing the schedule

Once the linear program is solved, we aim at (i) fully characterizing the schedule during one time-period, and (ii) deriving an actual schedule (with proper initialization and clean-up) whose asymptotic efficiency will hopefully be optimal.

4.1 During a period in steady-state mode

There are several subtle points when reconstructing the actual periodic schedule, i.e. the detailed list of actions of the processors during a time period. Once the linear programming problem is solved, we get the period T of the schedule, and the integer number of messages going through each link. First, because it arises from the linear program, $\log T$ is indeed a number polynomial in the problem size, but T itself is not necessarily polynomial in the problem size. Hence, describing what happens at every time-step during the period might be exponential in the problem size, and we need a more "compact" description of the schedule. Second, we need to exhibit an orchestration of the message transfers where only independent communications, i.e. involving disjoint pairs of senders and receivers, can take place simultaneously.

Both problems are solved as follows. From our platform graph G, and using the result of the linear program, we build a bipartite graph: for each node P_i in G, we create two nodes P_i^{send} and P_i^{recv}. For each communication from P_i to P_j, we insert an edge between P_i^{send} and P_j^{recv}, which is weighted by the length of the communication. We are looking for a decomposition of this bipartite graph into a set of subgraphs where a node (sender or receiver) is occupied by at most one communication task. This means that at most one edge reaches each node in the subgraph. In other words, only communications corresponding to a matching in the bipartite graph can be performed simultaneously, and the desired decomposition of the graph is in fact an edge coloring. The weighted edge coloring algorithm of [15, vol.A,chapter 20] provides in time $\mathcal{O}(|E|^2)$ a polynomial number of matchings (in fact, no more than $|E|$ matchings) which are used to perform the different communications, and which provides the desired polynomial-size description of the schedule. See [6, 4] for further details.

Altogether, this technique leads to the description of an optimal periodic schedule for our first two problems, namely the master-slave tasking and the series of scatters. Nothing prevents us to use the solution of the series of scatters for a series of multicast, but the throughput is no longer shown to be optimal.

4.2 Asymptotic optimality

Because the different instances of the problem (tasks, scatter operations) are independent, it is easy to derive an actual schedule based upon the steady-state operation. We need a fixed number of periods (no more than the depth of the platform graph rooted at P_m or P_{source}) to reach the steady-state: this corresponds to the initialization phase (and similarly for the clean-up phase).

The asymptotic optimality of the final schedule directly follows. In fact, we have a very strong result, both for master-slave tasking [3, 2] and for pipelined scatters [12]: the number of tasks or scatter operations processed within K time-units is optimal, up to a constant number, which only depends upon the platform graph (but not on K).

Note that we can generalize the master-slave tasking to the case of independent task graphs (instead of independent tasks). Then, collections of identical DAGs are to be scheduled in order to execute, say, the same suite of algorithmic kernels, but using different data samples. There are no dependences between successive DAGs, but of course there are some within a DAG. This mixed data and task parallelism problem has not been solved for arbitrary DAGs, but the approach presented in section 3.1 can be extended to any DAG with a polynomial number of simple paths [6, 4].

Finally, the approach for scatters also works for personalized all-to-all and reduce operations [12].

4.3 Optimal throughput for the multicast problem

The news for the pipelined multicasts is not so good: the problem of determining the optimal throughput is NP-hard [7]. In that case, going from makespan minimization to steady-state has not decreased the complexity of the problem.

However, for series of *broadcasts* rather than *multicasts*, the optimal steady-state can be characterized in polynomial time. Contrarily to the case of the multicast, the bound given by the linear program where the $\max$ operator replaces the $\sum$ operator turns out to be achievable [5]. Intuitively, because each intermediate processor participates in the execution, it is not important to decide which messages will be propagated along which path: in the end, everybody has the full information.

This is not the case for the multicast problem, and we illustrate this using an example. Consider the platform graph represented in Figure 2, where values labeling edges are the communication costs of a unit-size message. A solution of the linear program of section 3.2, but with the $\max$ operator instead of the $\sum$ operator, is shown in the following figures, and reaches the throughput of one message per time-unit. Figure 3(a) shows the number of messages sent on each link and whose target processor is P_5, while Figure 3(b) shows similar numbers for target processor P_6. Apparently, looking at Figure 3(c) which shows all the transfers, only one message needs to be sent through edge (P_3, P_4) every second time-unit, which would comply with the edge capacity. However when trying to reconstruct the schedule, we see that message routes differ for odd-numbered indices (label a) and even-numbered indices (label b). Messages are routed along two different trees. To reach P_5, odd-numbered multicast messages, with label a, use the route $P_0 \to P_1 \to P_5$, while even-numbered messages, with label b, use the route $P_0 \to P_2 \to P_3 \to P_4 \to P_5$. Similarly, there are two routes to multicast the messages to P_6: route $r_1 = P_0 \to P_1 \to P_3 \to P_4 \to P_6$ and route $r_2 = P_0 \to P_2 \to P_5$. Label a messages targeted to P_6 must use route r_1, because the edge (P_0, P_2) alone has not the capacity to carry all the messages. Label b messages targeted to P_6 then use route r_2, as shown on Figure 3(c). As a result, the edge $P_3 \rightarrow P_4$ is required to transfer both one a and one b messages every time unit, which is not possible because of its communication cost. Therefore, reconstructing a schedule from the solution of the linear program is not possible, the bound on the throughput cannot be met.

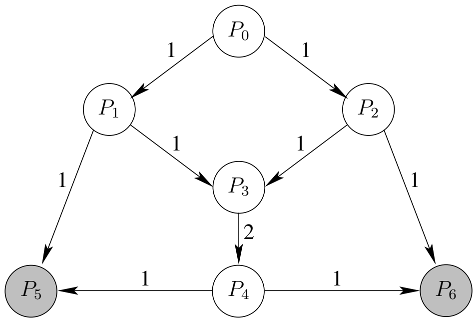


Figure 2: The multicast platform (target processors are shaded).

5 Limitations

5.1 Communication model

5.1.1 Send OR Receive

Surprisingly, the hypothesis that a processor can simultaneously send and receive messages is crucial for the reconstruction of the schedule. As detailed in section 4.1, the solution of the linear program gives a list of actions to be conducted during a period, and the edge-coloring algorithm implements the desired orchestration.

If we assume that a processor can *either* send *or* receive, it is easy to modify the linear program: for each processor, write the constraint that the time spent sending plus the time spent receiving does not exceed one time-unit. However, extracting independent communications (involving disjoint processor pairs) amounts to edge-color an arbitrary graph. The problem becomes NP-hard, but efficient polynomial approximation algorithms can be used [1]. However, for general graphs, we do not know the counterpart of the edge coloring algorithm for bipartite weighted graphs [15, vol.A,chapter 20].

This is bad news, because the fact that the activity variables output by the linear program did lead to a feasible periodic schedule, regardless of their ordering, was a key advantage of the steady-state approach.

5.1.2 Multiport

We can also consider more powerful communication models, where an host, equipped with several network card devices, can be involved in several receptions or emissions simultaneously. In the case where each network card on a given host is used in only one direction (sending or receiving) and is linked to a set of fixed network cards on neighbor hosts, then a linear program can be derived (constraints are written for each network card), and the schedule can be reconstructed (each node in the bipartite graph corresponds to a network card).

In the case where a network card can be used for both sending and receiving operations, then the problem is NP-hard (see above). In the case where a network card is dedicated either to send or to receive data, but can be involved in a communication with any neighbor host, then the complexity of the schedule reconstruction is still open.

5.2 Start-up costs

Linear programs are naturally suited to linear costs, so introducing computation or communication start-ups complicates the story. However, in many situations, the difficulty can be circumvented as follows:
1. Compute a lower bound of the total execution time for n tasks $T_{\text{opt}}(n)$
2. Use the solution of the linear program and the reconstruction of the solution to design a periodic schedule whose time period is large (of order $\sqrt{T_{\text{opt}}(n)}$). A (small) fraction of each period will be wasted due to start-up costs.
3. Design initialization and clean up phases (used before and after the periodic schedule).
4. Prove the asymptotic optimality of the resulting schedule.

The rationale behind the strategy is simple: (i) the length of the period should increase to $+\infty$ together with the total amount of work, so that start-up overheads end up by being negligible; (ii) the work performed during a period should tend to a negligible fraction of the total amount of work, because a fixed number of periods are "wasted" during the initialization and clean-up phases.

The first examples of this strategy were given in [9]. It was successfully applied to divisible load computations in [8]. Let us detail the different phases for master slave tasking when the communication of n_{ij} tasks from P_i to P_j now takes $C_{ij} + n_{ij}c_{ij}$ time-steps, where C_{ij} is the start-up cost.
1. Clearly, the platform with start up costs is less powerful than the platform without start-up costs, so that $T_{\text{opt}}(n) \leqslant \frac{n}{n_{\text{task}}(G)}$.
2. Consider the solution obtained after the reconstruction (without start up costs) and let T denote its period. The idea is to group the communications of m consecutive periods into a new period, so as to diminish the influence of start-up costs. We slightly increase the new period mT in order to take start-up costs into account. Since there are at most $|E|$ communication rounds per period, the overall cost due to start-ups can be bounded by $C|E|$, where $C = \max C_{ij}$. Thus, during each period of duration $mT + C|E|$, exactly $mTn_{\text{task}}(G)$ are processed.
3. The initialization phase consists in sending (sequentially) to each processor the number of tasks it will both compute and send during the first time period. The duration of this initialization phase can be bounded by $A_1 m$, where A_1 only depends on the platform (and not on n). The clean-up phase consists in processing "in place" the tasks that have not been processed yet after the first $\lfloor \frac{n}{mTn_{\text{task}}(G)} \rfloor$ periods. Since $mTn_{\text{task}}(G)$ are processed during each period, at most $mTn_{\text{task}}(G)$ have not been processed yet, so that the duration of the clean up phase can be bounded by $A_2 m$, where A_2 only depends on the platform (and not on n).
4. The overall time $T(n)$ of initialization, steady state and clean up phases is therefore bounded by

$$T(n) \leqslant (A_1 + A_2)m + \frac{n}{n_{\text{task}}(G)} + \frac{C|E|n}{mTn_{\text{task}}(G)},$$

so that if we set $m = \lceil \sqrt{\frac{n}{n_{\text{task}}(G)}} \rceil$, then

$$\frac{T(n)}{T_{\text{opt}}(n)} \leqslant 1 + \frac{\sqrt{n_{\text{task}}(G)}}{\sqrt{n}}(A_1 + A_2 + \frac{C|E|}{T}) + O(\frac{1}{n}),$$

thereby achieving the proof of asymptotic optimality when n becomes arbitrarily large.

5.3 Platform model

The target architectural platform model presented in section 2 takes into account the most important features of actual grid platforms: heterogeneity, link contention, 1-port constraints, overlapping capabilities. We consider it as a good compromise between simplicity (necessary in order to build efficient algorithms) and realism (necessary to build useful algorithms). Nevertheless, the actual topology of a large scale meta-computing platform is usually not known. On one hand, as proposed in [10], we can execute pings between each pair of participating hosts in order to determine the latency and bandwidth between each pair of hosts. This leads to a complete graph where contention are not taken into account. On the other hand, according to Paxson [14], it is very difficult to determine the paths followed by the packets on wide-area networks, and it is almost impossible to deduce the communication performance and the interaction of one data stream on another.

Fortunately, we only need a macroscopic view of the network, showing that some link is shared between some routes, without determining the actual physical topology. Recently, several tools have been designed in order to obtain such a macroscopic view. ENV [16] has been especially designed for master slave tasking, since it provides the view of the platform as seen from the master (i.e. a tree showing shared links). AlNeM [13] provides a platform model which is closer to the model presented in section 2. Both tools perform simultaneous communications between several host pairs in order to determine whether some links are shared on the route between these pairs. The main limitation of both tools is that the search of the topology of a real large scale platform requires a huge amount of time, hence limiting their use to stable platforms.

5.4 Approximation for fixed periods

In the case where the period obtained from the linear program is very large, we may want to restrict to fixed-length periods. The price to pay is that the throughput may be lowered. Again, it is possible to derive fixed-period schedules whose throughputs tend to the optimum as the length of the period increases. See [4] for further details.

5.5 Dynamic versions

A key feature of steady-state scheduling is that it is adaptive. Because the work is divided into periods, it is possible to dynamically adjust to changes in processor speeds or link bandwidths.

Indeed, a classical approach to respond to change in resources capabilities is borrowed from the simple paradigm *"use the past to predict the future"*, i.e. to use the currently observed speed of computation of each machine and of each communication link to decide for the next distribution of work. There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation. The situation is even worse for communication links, because of unpredictable contention problems. When deploying an application on a platform, the idea is thus to divide the scheduling into phases. During each phase, all the machine and network parameters are collected and histogrammed, using a tool like NWS [18]. This information will then guide the scheduling decisions for the next phase.

This approach naturally fits with steady-state scheduling. A first solution is to recompute the solution of the linear program periodically, based upon the information acquired during the current period, and to determine the activity variables for the new period accordingly. A second solution is more dynamic: each processor executes a load-balancing algorithm to choose among several allocations (for instance, various weighted trees for the scatter problem). This technique has been used for scheduling independent tasks on tree-shaped platforms [11].

6 Conclusion

The use of steady-state scheduling techniques turned out to be very helpful to solve a variety of problems, which includes master-slave tasking, divisible load scheduling, and pipelining several types of macro-communications. The list is far from being exhaustive, and we aim at enlarging it in the future. More importantly, we hope to characterize, or at least better understand, which are the situations when the bound output by the solution of the linear program is achievable.

We conclude by stating an open problem: what is the complexity of computing the optimal steady-state for the problem of mapping collections of arbitrary task graphs (with an exponential number of paths, such as the Laplace graph)? We conjecture that determining this optimal throughput is NP-hard.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer, Berlin, Germany, 1999.

[2] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15, 2004.

[3] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor grids. In *PARA'02: International Conference on Applied Parallel Computing*, LNCS 2367, pages 423–432. Springer Verlag, 2002.

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Optimal algorithms for the pipelined scheduling of task graphs on heterogeneous systems. Technical Report RR-2003-29, LIP, ENS Lyon, France, April 2003.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Optimizing the steady-state throughput of broadcasts on heterogeneous platforms heterogeneous platforms. Technical report, LIP, ENS Lyon, France, June 2003.

[6] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Scheduling strategies for mixed data and task parallelism on heterogeneous clusters. *Parallel Processing Letters*, 13(2), 2003.

[7] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Complexity results and heuristics for pipelined multicast operations on heterogeneous platforms. Technical report, LIP, ENS Lyon, France, January 2004.

[8] O. Beaumont, A. Legrand, and Y. Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Computing*, 29:1121–1152, 2003.

[9] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[10] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Adaptive communication algorithms for distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 59(2):252–279, 1999.

[11] L. Carter, H. Casanova, J. Ferrante, and B. Kreaseck. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2003*. IEEE Computer Society Press, 2003.

[12] A. Legrand, L. Marchal, and Y. Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. Technical Report RR-2003-33, LIP, ENS Lyon, France, June 2003.

[13] A. Legrand, F. Mazoit, and M. Quinson. An application-level network mapper. Research Report RR-2003-09, LIP, ENS Lyon, France, feb 2003.

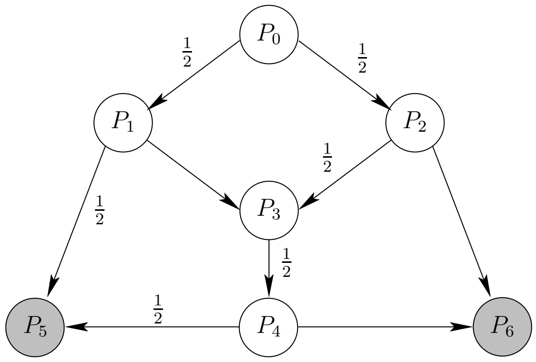
[14] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, 1997.

[15] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.

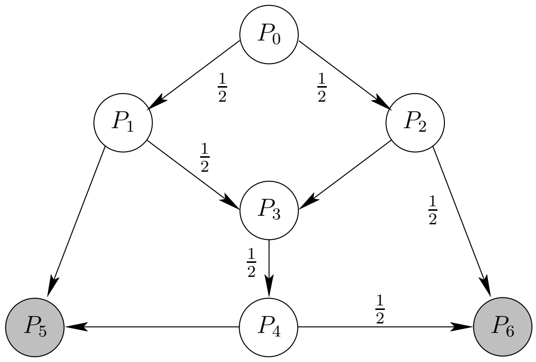
[16] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.

[17] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

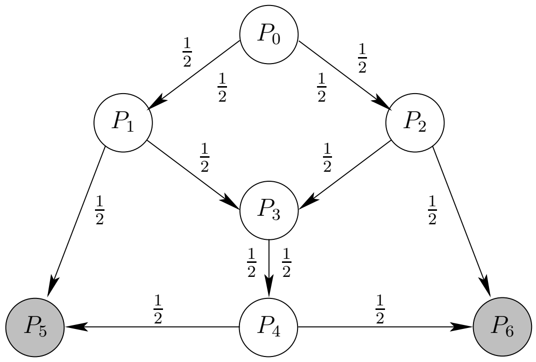
[18] R. Wolski, N.T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(10):757–768, 1999.



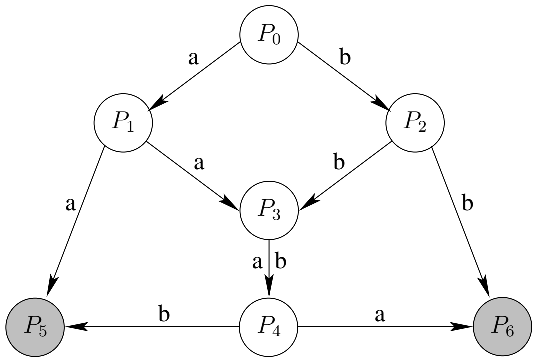
(a) Number of messages transferred through each edge and targeting P_5



(b) Number of messages transferred through each edge and targeting P_6



(c) Total number of messages going through each edge



(d) Conflict between two distinct messages through edge $P_3 \to P_4$

Figure 3: Multicast: Problems while reconstructing a schedule