Who needs a scheduler?

Anne Benoit, Loris Marchal, Yves Robert

To cite this version:

Anne Benoit, Loris Marchal, Yves Robert. Who needs a scheduler?. [Research Report] LIP RR-2008-34, Laboratoire de l'informatique du parallélisme. 2008, 11p. hal-02102807

HAL Id: hal-02102807

https://hal-lara.archives-ouvertes.fr/hal-02102807v1

Submitted on 17 Apr 2019

Who needs a scheduler?

Anne Benoit, Loris Marchal and Yves Robert[*]
École Normale Supérieure de Lyon, France
{Anne.Benoit|Loris.Marchal|Yves.Robert}@ens-lyon.fr

October 2008

LIP Research Report RR-2008-34

Abstract

This position paper advocates the need for scheduling. Even if resources at our disposal would become abundant and cheap, not to say unlimited and free (a perspective that is not granted), we would still need to assign the right task to the right device. We give several simple examples of such situations where resource selection and allocation is mandatory. Finally we expose our views on the important algorithmic challenges that need be addressed in the future.

1 Introduction

Already in the former century, scheduling was sometimes considered as a minor, and more or less useless, activity. Young colleagues and talented students could hear through the grapevine that *"Scheduling is this thing that people in academia like to think about but that people who do real stuff sort of ignore"*. Most of them were discouraged and went into bad topics like programming languages, operating systems and networks. Some bravely stayed in the field of algorithm design and scheduling techniques. At least we did.

Today the question is raised much stronger. With so many billions of (mostly idle) computers in the world, all interconnected by these (mostly empty) network pipes, the resources at our disposal become abundant and cheap, not to say unlimited and free. Well, at least there is a chance for this dream to become true. Who would then need a complicated scheduling algorithm while a greedy resource allocation is likely to do the job? Demand-driven approaches like *first-come first-serve* or *round-robin* will perform extremely well in most situations. In short, *who needs a scheduler* with infinite resources handy and ready? Should the scheduling community in general, and ourselves in particular, do something else, more useful, and more trendy, like, say, cloud computing with energy-aware self-programming multicores?

Usually this questioning does not last long. The first reason is that after some time spent with systolic arrays, SIMD and MIMD machines, hypercube algorithms, task graphs and pipelined workflows, you get to like the mathematical beauty of the art of giving complicated solutions to simple problems (as opposed to giving simple solutions to complicated problems, the dominant activity in the afore-mentioned bad topics). The second reason is that we see a bright future for scheduling. This position paper is going to explain why (but expect no more jokes, only technical statements, from now on).

The rest of the paper is organized as follows. First we tentatively define the activity of scheduling in Section 2. Next in Section 3, we explain why the classical macro-dataflow model should be replaced by more realistic communication models such as the one-port and bounded multi-port ones. In Section 4 we present a simple case-study, that of bag-of-tasks applications, in order to mathematically assess the importance of resource selection and load assignment strategies. Then we address more realistic (and also more complicated) problems in Section 5. Finally we state some final remarks in Section 6.

[*] The authors are with LIP, jointly operated by CNRS, ENS Lyon, INRIA and UCB Lyon. Postal address: 46 allée d'Italie, 69364 Lyon Cedex 07, France.

2 Scheduling / demand-driven?

The word *scheduling* has different meanings. In this paper we deal with what is usually called *static scheduling*, an activity which starts with a set of tasks (organized as a precedence graph, or DAG), and with a target computing platform, as input, and consists in mapping the former on the latter with the goal to optimize some objective function (often the total execution time, or *makespan*). Static schedulers need a reasonably good knowledge of the application parameters. More precisely, the structure of the DAG, and estimations of node and edge weights (which correspond to computation costs and communication volumes respectively) are fed into the scheduler.

This is different from *dynamic scheduling*, or better *demand-driven resource allocation*, which consists in mapping jobs onto shared computational resources. Typically, very little is known about the jobs, maybe rough estimates of their execution times in some cases, and nothing is known in advance about their incoming rate. In such situations, there is not much else to do than assigning new loads to currently idle resources, satisfying requests with a simple FIFO policy.

The terms *static* and *dynamic* are misleading, because a scheduler can (dynamically) take new decisions on the fly, based upon newly acquired information on application and platform parameters. We refer to *scheduling* as the activity of designing algorithms and heuristics (e.g., a list schedule) in order to deploy an application onto a platform. On the contrary, a *demand-driven* approach is a system-oriented approach where resources are allocated upon demand to incoming requests. Of course this is an overly simplified classification and there is a continuum. When several applications (rather than one) are simultaneously deployed by a single user on a platform shared by many other users (rather than on a dedicated platform), the difference between both approaches narrows. To our view, the difference goes beyond *off-line* versus *on-line*, or *compile-time* versus *run-time*. Basically, the more we know about what we need to schedule, and the more refined the decisions that the scheduler can take.

Maybe it was only a rhetorical question, or a "philosophical" debate, but we wanted to clarify the context!

3 Kill the bad model

Distributed-memory parallel computing platforms pose many challenges to the algorithm designer and the programmer. An obvious factor contributing to this complexity is the need for network communication, whose performance is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms. Older parallel computers used a store-and-forward approach to communicate messages, which was not efficient but simple to understand and to model. Essentially, the time for sending a message from a processor p to a processor p' is $c(p,p') = \mathit{dist}(p,p') \times (L + s/b)$, where s is the length of the message, $\mathit{dist}(p,p')$ is the distance between p and p' in number of hops, L is the communication start-up cost, and b is the steady-state bandwidth. In modern computers, messages are split into packets that are dynamically routed between processors, possibly using different paths. Messages can be routed efficiently if there are no contentions on the communication links (or "hot spots"). The distance between communicating processors is no longer the single most important factor for communication performance. In fact, if several processors are to exchange data simultaneously, then the more structured the communication patterns, the more efficient they are, making the role of locality on performance at best indirect.

In light of the complexity of performance modeling for network communications, the vast majority of scheduling works and results are for a very simple model, which is as follows. If a task T communicates data to a successor task T', the cost is modeled as

$$\text{cost}(T,T') = \begin{cases} 0 & \text{if } \mathit{alloc}(T) = \mathit{alloc}(T') \\ c(T,T') & \text{otherwise,} \end{cases}$$

where $\mathit{alloc}(T)$ denotes the processor that executes task T, and $c(T,T')$ is defined by the application specification. The above model states that the time for communication between two tasks running on the same processor is negligible. The model also assumes that the processors are part of a fully connected clique. This so-called *macro-dataflow* model makes two main assumptions: (i) communication can occur as soon as data are available; and (ii) there is no contention for network links. Assumption (i) is reasonable as communications can overlap with computations in most modern computers. Assumption (ii) is much more questionable. Indeed, there is no physical device capable of sending, say, $1,000$ messages to $1,000$ distinct processors, at the same speed as if there were a single message. In the worst case, it would take $1,000$ times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources! Nevertheless, this assumption is omnipresent in the traditional scheduling literature. Perhaps it was the price to pay to derive tractable mathematical results on makespan minimization?

Our conviction is that we need to turn to more realistic communication models when modeling concurrent communications. We outline two such models, that account for the interference between concurrent communications.

Bounded multi-port – Assuming an application that runs threads on, say, a node that uses multicore technology, the network link could be shared by several incoming and outgoing communications. Therefore, the sum of the bandwidths allotted by the operating system to all communications cannot exceed the bandwidth of the network card. The bounded multi-port model proposed by Hong and Prasanna [16] assesses that an unbounded number of communications can thus take place simultaneously, provided that they share the total available bandwidth. We point out that recent multi-threaded communication libraries such as MPICH2 [17] now allow for initiating multiple concurrent send and receive operations, thereby providing practical realizations of the multi-port model.

One-port – To avoid unrealistically optimistic results obtained with the multi-port model, a radical option is simply to forbid concurrent communications at a node. In the one-port model, a node can either send data or receive data, but not simultaneously. This model is thus very pessimistic as real-world platforms can achieve some concurrency of communication. On the other hand, it is straightforward to design algorithms that follow this model and thus to determine their performance a priori.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It is used by Bhat et al. [8, 9] for fixed-sized messages. They advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has recently been related by Saif and Parashar [21], who report that asynchronous sends become serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular implementations of the MPI message-passing standard, MPICH on Linux clusters and IBM MPI on the SP2.

There are more complicated models such as those that deal with bandwidth sharing protocols [19, 18]. Such models are very interesting for performance evaluation purposes, but they almost always prove too complicated for algorithm design purposes. For this reason, we prefer to deal with the bounded multi-port or the one-port model. We believe that these models represent a good trade-off between realism and tractability.

4 Case study: bags-of-tasks

In this section we study the deployment of BOINC-like applications [11] under the previous one-port and bounded multi-port models. We start with the simplest problem, that of scheduling a single bag-of-tasks made up of a large number of same-size tasks onto a master-worker platform. Next we proceed with several bag-of-tasks applications onto the same simple master-worker platform. Finally we briefly discuss both problems on general platforms.

4.1 Steady-sate scheduling

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? Often, the motivation for deploying an application on a parallel platform is that the number of tasks is very large. In this case, the optimal execution time with the optimal schedule may be very large and a small deviation from it is likely acceptable. To state this informally: if there is a nice (e.g., 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.

This approach has been pioneered by Bertsimas and Gamarnik [7]. Steady-state scheduling allows one to relax the scheduling problem in many ways. The costs of the initialization and clean-up phases are ne-

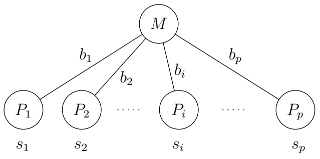


Figure 1: Star-shaped master-worker platform.

glected. The initial integer formulation is replaced by a continuous or rational formulation. The precise scheduling of computations and communications is not required, or at least not before the optimal schedule is outlined. 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. The actual schedule then arises naturally from these quantities.

4.2 One bag-of-tasks

In this section, we target simple heterogeneous star-shaped platforms. The master M initially holds a large collection of atomic tasks. Refer to Figure 1 for notations:

- The master M sends tasks to workers *sequentially*, and without preemption (one-port model).
- There is full computation/communication overlap at each worker.
- A task consists of an input file of size δ (in Bytes), and a computation job of size w (in Flops).
- Worker P_i has a communication bandwidth b_i: it receives a task in δ/b_i time units.
- Worker P_i has a computation speed s_i: it processes a task in w/s_i time units.
- The master M does not compute any task (but a master with computation speed s_0 can be simulated as a worker with the same computation speed and infinite bandwidth).

When dealing with a single bag-of-tasks application, we assume that $\delta = w = 1$ without loss of generality (processor speeds and bandwidths can be scaled).

The optimal steady-state is defined as follows: for each worker, determine the fraction of time spent computing tasks, and the fraction of time spent receiving tasks; for the master, determine the fraction of time spent communicating along each communication link. The objective is to maximize the (average) number of tasks processed per time unit. Formally, after a start-up phase, we want the resources to operate in a periodic mode, with worker P_i executing α_i tasks per time unit. We point out that α_i is a rational number, not an integer, so that there remains some work to reconstruct a feasible schedule, i.e., with an integer number of tasks.

One-port – First we express the constraints for computations: P_i must compute α_i tasks within one time unit, thus we must have $s_i \geq \alpha_i$, and

$$\alpha_i/s_i \leq 1 \ . \tag{1}$$

As for communications, the master M sends tasks sequentially to the workers, and it must send α_i tasks per time unit along the link to P_i. Thus, by summing all communication times we obtain

$$\sum_{i=1}^{p} \alpha_i/b_i \leq 1 \ . \tag{2}$$

Finally, the objective is to maximize the throughput, namely,

$$\rho = \sum_{i=1}^{p} \alpha_i \ .$$

Altogether, we have a linear programming problem with rational unknowns:

$$\begin{array}{l}\textsc{Maximize } \rho, \\ \textsc{subject to} \\ \left\{\begin{array}{lll} & \rho = \sum_{i=1}^{p} \alpha_i & \text{(i)} \\ & \sum_{i=1}^{p} \alpha_i/b_i \leq 1 & \text{(ii)} \\ \forall i, & \alpha_i/s_i \leq 1 & \text{(iii)} \\ \forall i, & \alpha_i \geq 0 & \text{(iv)} \end{array}\right.\end{array} \tag{LP}$$

It turns out that the linear program is so simple that it can be solved analytically. Indeed it is a fractional knapsack problem [12] with value-to-cost ratio b_i. We should start with the "item" (worker) of the largest ratio, i.e., the largest b_i, and take (assign) as many tasks as we can, i.e., min (b_i, s_i). Here is the detailed procedure:

1. Sort the workers by increasing communication times. Re-number them so that $b_1 \geq b_2 \ldots \geq b_k$.

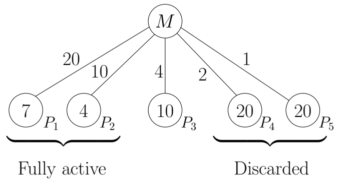
2. Let q be the largest index so that $\sum_{i=1}^{q} \frac{s_i}{b_i} \leq 1$. Workers P_1 to P_q will be fully active (and each of them will execute s_i tasks per time unit). If $q < p$, let $\varepsilon = 1 - \sum_{i=1}^{q} \frac{s_i}{b_i}$, otherwise let $\varepsilon = 0$. Worker P_{q+1} (if it exists) will be only partially active, and will execute $\min(\varepsilon \cdot b_{q+1}, s_{q+1})$ tasks per time unit.

3. Workers P_{q+2} to P_p (if they exist) are discarded; they will not participate in the computation.

4. The optimal throughput is then

$$\rho = \sum_{i=1}^{q} s_i + \min(\varepsilon \cdot b_{q+1}, s_{q+1}) \ .$$

When $q = p$ the result is expected. It basically says that workers can be fed with tasks fast enough so that they are all kept computing steadily. However, if $q < p$, the result is surprising. Indeed, if the communication bandwidth is limited, some workers will partially starve. In the optimal solution these partially starved workers are those with slow communication rates, *regardless* of their processing speeds. In other words, a slow processor with a fast communication link is to be preferred to a fast processor with a slow communication link. This optimal strategy is often called *bandwidth-centric* because it delegates work to the fastest communicating workers, regardless of their computing speeds. Of course, slow workers will not contribute much to the overall throughput.



(a) Platform with bandwidths and speeds.

Tasks	**Communication**	**Computation**
7 tasks to P_1	$7/b_1 = 7/20$	$7/s_1 = 1$
4 tasks to P_2	$4/b_2 = 8/20$	$4/s_2 = 1$
1 tasks to P_3	$1/b_3 = 5/20$	$1/s_3 = \mathbf{1/10}$

(b) Achieved throughput for the bandwidth-centric strategy

Figure 2: Example under the one-port model.

Consider the example shown in Figure 2. Workers are sorted by non-increasing b_i. We see that $\frac{s_1}{b_1} + \frac{s_2}{b_2} = \frac{15}{20} < 1$ and that $\frac{s_1}{b_1} + \frac{s_2}{b_2} + \frac{s_3}{b_3} = \frac{65}{20} \geq 1$, so that $q = 2$ and $\varepsilon = \frac{1}{4}$ in the previous formula. Therefore, P_1 and P_2 will be fully active, contributing $\alpha_1 + \alpha_2 = s_1 + s_2 = 11$ tasks per time unit. P_3 will only be partially active, contributing $\alpha_3 = \min(\varepsilon \cdot b_{q+1}, s_{q+1} = \min(1, 10) = 1$. P_4 and P_5 will be discarded. The optimal throughput is $\rho = 7 + 4 + 1 = 12$. Figure 2(b) shows that 12 tasks are computed every time unit.

It is important to point out that if we had used a purely greedy (demand-driven) strategy, we would have reached a much lower throughput. Indeed, the master would serve the workers in round-robin fashion, and we would execute only 5 tasks every $\frac{1}{20} + \frac{1}{10} + \frac{1}{4} + \frac{1}{2} + 1 = \frac{19}{10}$ time units, therefore achieving a throughput of only $\rho = 10/19 \approx 0.53$. The conclusion is that even when resources are cheap and abundant, resource selection is key to performance.

The good news is that the actual periodic schedule can easily be constructed from the linear program, and that this schedule is asymptotically optimal. See [4] for details.

Bounded multi-port – How can we solve the same problem using the bounded multi-port model instead of the one-port model? Refer to the one-port linear program again. Because messages can now be sent in parallel, we replace Equation (ii) by

$$\forall i, \quad \frac{\alpha_i}{b_i} \leq 1 \ , \qquad \text{(ii-a)}$$

which states that the bandwidth of the link from M to P_i is not exceeded. We also have to enforce a global bound related to the bandwidth B of the master's network card:

$$\frac{\sum_{i=1}^{p} \alpha_i}{B} \leq 1 \ . \qquad \text{(ii-b)}$$

Replacing Equation (ii) by both Equations (ii-a) and (ii-b) is all that is needed to change to the bounded multi-port model.

However, this modification has a dramatic impact on the solution and on the scheduler. Resource selection is not needed any longer. If we enroll all (or sufficiently many) available resources and feed each of them using a pure demand-driven basis (thereby enforcing that $\alpha_i \leq \min(s_i, b_i)$), we end up reaching the maximum throughput $\rho_{\text{opt}} = \min\left(B, \sum_{i=1}^{n} \min(s_i, b_i)\right)$ dictated by the master's outgoing communication capacity.

This is quite contradictory with our initial claim. Is the complexity an artifact of the one-port model? Is it enough to change the communication model for the algorithmician and his static scheduler to disappear? We will see in the following that even in the "simple" multi-port model, static knowledge is required to efficiently schedule several applications.

4.3 Several bags-of-tasks

We now consider that a single scheduler has to cope with tasks belonging to several applications. There are K application $(A_1, \dots, A_K)$, and each application consists of a large number of same-size tasks, to be executed on the same master-worker platform. Some new notations are needed:

- δ_k is the size (in Bytes) of an input file for application A_k; processor P_i receives a task of A_k in $\frac{\delta_k}{b_i}$ time units.
- w_k is the size (in Flops) of a task for application A_k; processor P_i executes a task of A_k in $\frac{w_k}{s_i}$ time units.

When dealing with several applications in steady state mode, α_i^k denotes the local throughput of application A_k on processor P_i. In other words, processor P_i executes α_i^k tasks of applications A_k during one time unit. As previously, α_i^k might be a rational number. The total throughput ρ^k of an application A_k is then given by $\rho^k = \sum_{i=1}^{p} \alpha_i^k$.

Since we have several applications to schedule on the same platform, we have to modify the objective to take all applications into account. We assume that some applications may be more important than others. Each application A_k is provided with a priority π_k, so that if $\pi_k = 2\pi_{k'}$, the throughput of A_k must be twice the throughput of $A_{k'}$. Our objective is then to maximize $\min_k \left\{ \frac{\rho^k}{\pi_k} \right\}$.

One-port – We extend the linear program (LP) to several applications. For the one-port model, we get the following formulation:

$$\textsc{Maximize } \min_k \left\{ \frac{\rho^k}{\pi_k} \right\}$$
$$\text{SUBJECT TO}$$
$$\left\{ \begin{array}{lll} & \rho^k = \sum_{i=1}^{p} \alpha_i^k & \text{(M-i)} \\ & \sum_{i=1}^{p} \sum_{k=1}^{K} \alpha_i^k \frac{\delta_k}{b_i} \leq 1 & \text{(M-ii)} \\ \forall i, & \sum_{k=1}^{K} \alpha_i^k \frac{w_k}{s_i} \leq 1 & \text{(M-iii)} \\ \forall i, \forall k & \alpha_i^k \geq 0 & \text{(M-iv)} \end{array} \right. \qquad \text{(M-LP)}$$

We characterize each application A_k by its communication-to-computation ratio (CCR) δ_k / w_k: the larger the CCR, the more communication-intensive the application. This parameter has a critical influence on the shape of the solution. In an optimal solution, applications with larger CCR should be allocated to processors with larger bandwidth. Resources are split in ordered "slices", each slice being devoted to the processing of application A_k. Figure 3 illustrates the affinity property [5]: if applications are sorted in non-increasing order of CCR ($\frac{\delta_1}{w_1} \geq \frac{\delta_2}{w_2} \geq \dots \geq \frac{\delta_K}{w_K}$) and processors are sorted in non-increasing bandwidth ($b_1 \geq b_2 \geq \dots \geq b_p$), then there exists indices $a_0, a_1, \dots a_K$ such that only processors $P_u, u \in [a_{k-1}, a_k]$ execute tasks of type k in the optimal solution.

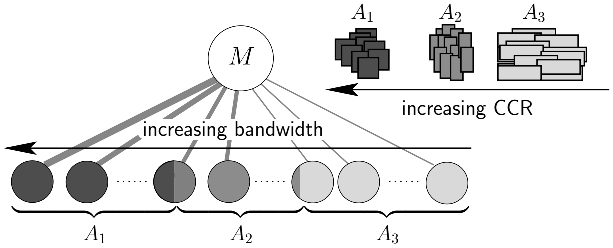


Figure 3: Shape of the optimal one-port solution.

In [5] we have experimentally compared the following three simple algorithms:

- A pure *demand-driven* strategy, where the scheduler sends a task of any application to the first worker posting a request
- A *coarse-grain* strategy: we assemble all applications into a single big one and use the bandwidth-centric algorithm explained above. For example, consider two applications with priorities $\pi_1 = 3$ and $\pi_2 = 1$. We gather the tasks into bundles where each bundle contains three tasks of application A_1 and one task of application A_2. We now have reduced the problem to a single, coarse-grain application to schedule.
- An *affinity-based* strategy, which relies on the above affinity property to pair application tasks and computation/communication resources.

The third strategy dramatically outperformed the first two. We expected the result for the first strategy. But it is insightful that the second strategy, although optimal for a single bag-of-tasks, was not "clever" enough for several ones.

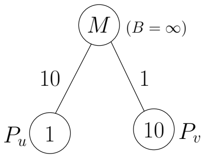
Bounded multi-port – We now move to the multi-port model. As for one application, we can easily adapt the linear program in order to cope with this model. Constraint (M-ii) is replaced by two constraints, one bounding the capacity of each edge:

$$\forall i, \quad \sum_{k=1}^{K} \alpha_i^k \frac{\delta_k}{b_i} \leq 1, \qquad \text{(M-ii-a)}$$

and one bounding the network capacity of the master:

$$\sum_{i=1}^{p} \sum_{k=1}^{K} \alpha_i^k \frac{\delta_k}{B} \leq 1. \qquad \text{(M-ii-b)}$$

Although the affinity between applications and processors does not result in a slicing property as in the one-port model, it still has a big impact on the optimal solution.



A_1	A_2
$\delta_1 = 1$	$\delta_2 = 10$
$w_1 = 10$	$\delta_2 = 1$
$\pi_1 = 1$	$\pi_2 = 1$

(a) Platform and applications.

throughput	for A_1		for A_2		objective
	on P_u	on P_v	on P_u	on P_v	
optimal	0	1	1	0	1
coarse-grain	1/11	1/11	1/11	1/11	1/11

(b) Throughput achieved by both strategies.

Figure 4: Example of multiple applications with multi-port model.

Consider the simple problem described in Figure 4(a), with two processors and two applications of same priority. Processor P_1 has a large computation speed but a small bandwidth, while it is the opposite for P_2. Application A_1 is computation-intensive with CCR 1/10, while A_2 is communication-intensive, with CCR 10. In the optimal solution as given by the linear program, tasks of application A_2 are executed only by processor P_1, while processors P_2 is in charge of all tasks of application A_1. This results in a throughput of one task per time unit for each application.

If we gather both applications into a coarse-grain application, we get tasks composed of one task of A_1 and one task of A_2. The parameters of the coarse-grain application are $\delta_{CG} = 11$ and $w_{CG} = 11$. Each processor is only able to process 1/11 tasks during each time unit, and the throughput is dramatically decreased. Note that in this toy example, we have not bounded the network capacity of the master.

How would a pure demand-driven scheduler behave on this example? This is hard to predict, as it depends on the initial tasks sent to each processor: if by chance, the scheduler sends a task of A_2 to P_1, and one of A_1 to P_2, then both tasks will be completed simultaneously and both processors will request some more work. If the scheduler repeats its initial choice, then it will reach the optimal throughput. On the contrary, if it makes the opposite choice (sending A_1 to P_1 and A_2 to P_2), then the processing of both tasks will be slowed down, leading to a throughput of 1/100.

Why take the risk? In a multi-application setting, demand-driven scheduling can be very unstable. Its seems reasonable to approximate its average performance by considering the coarse-grain scheduler, which represents the case where the fraction of each application sent to a processor does not depend on the target processor. In the multi-port model, the demand-driven strategy gives the best throughput for a single application, so we could expect a good performance with the resulting coarse-grain application. However, we have shown that its performance can be significantly reduced because it does not consider the affinity between processors and applications.

For the one-port model, the demand-driven strategy performs poorly even with a single application. Due to the high complexity of the one-port model, one could argue that the problems come from the limited capacity of the master, and that it would be sufficient to enhance its network capacity, or even to duplicate the server in charge of sending tasks for the scheduling complexity to disappear. However, the example of Figure 4 shows that even with unlimited network resources on the master and with the simple multi-port model, executing multiple applications with a demand-driven strategy leads to sub-optimal performance.

4.4 Platform selection

We have just shown the usefulness of a scheduler that performs resource selection, and assigns the best-suited application load to each enrolled resource: the throughput achieved for one or several bags-of-tasks is higher (and by an arbitrary factor) than that provided by demand-driven strategies. This observation holds true for the simplest possible platform, a single-level tree.

In practice the problem is more complicated. Either the platform is given, most likely in the form of a (hierarchical) multi-level tree, where each participating node has enrolled some neighboring resources. Or the platform is to be built out of, say, widely scattered and distributed resources (a cluster here, a supercomputer there, and a large network of workstations elsewhere). In the latter case, the user needs:

- either to extract the most efficient tree out of the general platform graph, which looks difficult because of the huge combinatorial set of possibilites to explore;
- or to deploy its application using the whole platform, which looks difficult too because the user will face in this case the complexity induced by cycles in the platform graph.

No need to go into technical details here, the reader will be easily convinced that every problem is indeed difficult. For a single bag-of-tasks, the throughput achieved by the best tree can be arbitrarily bad compared to that of the general platform [4]. With several bags-of-tasks on a fixed tree, complex algorithms must be used to achieve a good throughput [5].

5 Multi-criteria scheduling

So far we focused on maximizing the throughput, i.e., the number of tasks processed per time unit. However, even for simple BOINC-like applications, several other important optimization criteria should be considered to fulfill users expectations. In the following, we start by introducing *workflow* applications, which consist of pipelined DAGs (rather than independent tasks). Then we list many possible objectives for these applications, and we discuss how schedulers can deal with several (usually conflicting) objectives simultaneously.

5.1 Structured workflows

A bag-of-tasks application (Section 4) is a collection of identical independent tasks. A workflow is a collection of identical task graphs, or DAGs (hence a single bag-of-tasks is a workflow whose DAG reduces to a single node). Workflows naturally arise in many frameworks. Take the example of a JPEG encoder. You cannot apply the Fast Discrete Cosine Transformation [23] on your JPEG encoder (see http://www.jpeg.org/) before some pre-processing on the image: scaling, color space conversion, and so on. As a consequence, the application graph of the JPEG encoder is a linear chain, to be executed successively on each incoming image. Of course, rather than just chains, we can have fork or fork-join graphs, or series-parallel graphs, or even arbitrary DAGs.

Classical scheduling aims at minimizing the makespan of a single DAG: a single data set goes through the application graph. With workflows we have pipelined DAGs, because we operate on a collection of data sets that are executed in a pipeline fashion. Each data set is input to the application graph and traverses it until its processing is complete. Several data sets can be processed concurrently. Mapping and/or scheduling consists in assigning tasks to resources, so as to minimize one or several objectives. Again, a task (also called a stage) is in fact a collection of identical tasks to be executed for each data set (think of the images entering the JPEG encoder).

5.2 Objective functions

For workflow applications, the first objective that comes to mind is *throughput* maximization: the goal is to process as many data sets per time unit as possible. However, looking back at classical scheduling, makespan minimization was an important objective too. This remains true for workflows, and in particular for real-time applications. The definition must be adapted, and we talk of *latency* rather than of makespan, in order to avoid confusion. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. Note that it may well be the case that different data sets have different latencies (because they are mapped onto different processor sets), hence the latency is defined as the maximum response time over all data sets. Note also that minimizing the latency is antagonistic to maximizing the throughput. For a linear chain application, latency is minimized by assigning the whole application to a single processor, thus working in a fully sequential way: no communication is paid. However throughput can be increased by distributing tasks over processors and working in a pipelined manner. Already we guess that trade-offs will have to be found between these criteria. Indeed, several work dealt with both these criteria, for instance see [22, 6].

With the advent of large-scale heterogeneous platforms, resources may be cheap and abundant, but resource failures (processors/links) are more likely to occur and have an adverse effect on the applications. Not only every user is quite likely to face unrecoverable hardware failures when deploying applications on clusters or grids [14, 15, 1, 13], but unrecoverable interruptions can also take place in other important frameworks, such as loaned/rented computers being suddenly reclaimed by their owners, as during an episode of *cycle-stealing* [2, 10, 20]. Consequently, there is an increasing need for developing reliable schedules. One more optimization criterion that should be maximized is the *reliability* of the schedule, given a failure model for the resources.

Another trendy objective emerges for current platforms, namely the *energy* minimization objective. *Green* scheduling aims at minimizing energy consumption, by running processors at lower frequencies [3], or by reducing the number of processors enrolled. Of course being green often involves running at a slower pace, thereby reducing the application throughput.

Finally, even more objectives will appear in a multi-application setting. It was easy with several bags-of-tasks, because we assumed fixed priority factors (Section 4.3). More generally, some form of fairness must be guaranteed between all the applications. Typical measures are the maximum stretch of an application or the sum of all application stretches. The stretch of an application is the slowdown factor incurred by its execution time when sharing resources with the other applications. Add different release dates and deadlines for each application, and contemplate the difficulty of this scheduling problem!

5.3 Dealing with multi-criteria

How to deal with so many objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But is it natural for the user to maximize the quantity $0.7T + 0.3R$, where T is the throughput and R the reliability? What about adding latency and energy parameters into the story? Obviously, the problem here is that we mix apples and bananas: the criteria are very different in nature and it does not make much sense for a user to make a linear combination of them.

Users are more likely to ask questions like "I want a frame rate T and a response time L for my JPEG encoder, what is the least amount of energy that I will consume?". Thus we advocate the use of multi-criteria with thresholds. To give another example, we would aim at maximizing the throughput of the application, but accepting only schedules whose reliability is at least 99%. Now, each criteria combination can be handled in a natural and meaningful way: one single criterion is optimized, under the condition that a threshold is enforced for all other criteria.

Several interesting trade-offs appear when dealing with multi-criteria optimizations. Let us illustrate one of them with a little case study: the application graph is a linear chain, and we target throughput and reliability objectives. In order to increase reliability, a solution consists in replicating a task, or set of tasks, onto several resources. Then each data set is entirely processed by several resources, and if some of the resources fail during execution, the processing is not interrupted. In the extreme case, we could replicate the whole chain onto each resource, and even if all processors but one fail, we still get the result. However, the throughput would be very low for such a highly reliable schedule. For throughput maximization, we would rather split the chain and assign each task to a different processor, in order to process different data sets in parallel. Moreover, we can also replicate each task onto several processors, but this time to increase the throughput: for instance if we replicate a task on two processors, the first one would process even-numbered data sets, while the second processor would process odd-numbered data sets. If this task was the bottleneck of the application, then the throughput can be doubled. Of course this is much less reliable because a single failure stops the whole application.

In such situations, some knowledge of the application and platform parameters may help the scheduler decide which tasks to group onto the same processor set, and for each set, which processors are doing replication for reliability and which ones are doing replication for throughput. Needless to say, the story becomes even more complex when adding more objectives, and when tackling applications whose graph is an arbitrary DAG. Demand-driven strategies are quite likely to fail, even with an infinity of resources.

6 Conclusion

In this position paper, we have explained what scheduling means to us, and why we do believe that it is a mandatory activity. But first we must forget the macro-dataflow model of the scheduling literature, and use the one-port or bounded multi-port models instead.

We started with a glance at BOINC-like applications, introducing steady-state scheduling. Through this case study, we advocated the importance of going divisible (using non-integer number of tasks). Schedules can then be expressed in a compact periodic manner, as opposed to the full-length schedule descriptions of classical scheduling. Despite the simplicity of the problem, we have shown the importance of resource selection, even if resources are abundant and cheap.

Then, the story got more complicated with the introduction of multi-criteria scheduling: makespan minimization is not relevant enough in most situations. Users also care about throughput, reliability, energy, fairness, and so on. However, rather than linear combinations, it makes much more sense to optimize only one criterion, given that a threshold is enforced for the others. Often the criteria are antagonistic, which leads to many algorithmic challenges to tackle.

Altogether we gave several examples for which the design of a good scheduling algorithm was a "sine-qua-non" to obtain good performances. Of course, problems are even more complicated in real life, and the scheduler gets even more useful. Further techniques can be developed if the knowledge of the platform and/or of the application is only partial or not fully accurate. One can schedule pipelined applications by phases and re-inject currently acquired knowledge for the next phase, thereby exploiting up-to-date parameters. Otherwise, if the platform parameters are subject to variations (not to speak of unrecoverable interruptions), we can design robust algorithms able to react to these variations, through the use of stochastic models.

With the advent of multicores, and more importantly of clusters of multicores, additional problems will arise. Schedulers will have to cope with new locality rules, and to trade-off between (fast but scarce) memory accesses and (slower but unlimited) network communications. Most likely, yet another level of hierarchy (outermost tiling) will be needed. We intend to address these forthcoming algorithmic challenges. As claimed in the beginning, we do view a bright future for schedulers!

Acknowledgement. We thank Arnaud Legrand for several enlightening discussions. Several ideas exposed here matured through joint work with too many people to list here; we express our gratitude to all of them.

References

[1] J. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[2] B. Awerbuch, Y. Azar, A. Fiat, and F. Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time. In A. Press, editor, *28th ACM Symp. on Theory of Computing*, pages 519–530, 1996.

[3] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Power-aware scheduling for periodic real-time systems. *IEEE Trans. Computers*, 53(5):584–600, 2004.

[4] 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(4):319–330, 2004.

[5] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for bag-of-tasks applications. *IEEE Trans. Parallel Distributed Systems*, 19(5):698–709, 2008.

[6] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, October 2008. http://dx.doi.org/10.1007/s00453-008-9229-4.

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

[8] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[9] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[10] S. Bhatt, F. Chung, F. Leighton, and A. Rosenberg. On optimal strategies for cycle-stealing in networks of workstations. *IEEE Trans. Computers*, 46(5):545–557, 1997.

[11] BOINC: Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu`.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C.Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[13] A. Duarte, D. Rexachs, and E. Luque. A distributed scheme for fault-tolerance in large clusters of workstations. In *NIC Series, Vol. 33*, pages 473–480. John von Neumann Institute for Computing, Julich, 2006.

[14] A. H. Frey and G. Fox. Problems and approaches for a teraflop processor. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 21–25. ACM Press, 1988.

[15] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors. `http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf`, 2002.

[16] B. Hong and V. K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE Trans. Parallel Distributed Systems*, 18(10):1420–1435, 2007.

[17] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *J.Parallel and Distributed Computing*, 63(5):551–563, 2003.

[18] S. H. Low. A Duality Model of TCP and Queue Management Algorithms. *IEEE/ACM Trans. Networking*, 4(11):525–536, 2003.

[19] L. Massoulié and J. Roberts. Bandwidth sharing: Objectives and algorithms. *Transactions on Networking*, 10(3):320–328, june 2002.

[20] A. Rosenberg. Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. *IEEE Trans. Parallel and Distributed Systems*, 13(2):179–191, 2002.

[21] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[22] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 62–71. ACM Press, 1996.

[23] C. Wen-Hsiung, C. Smith, and S. Fralick. A Fast Computational Algorithm for the Discrete Cosine Tranfsorm. *IEEE Transactions on Communications*, 25(9):1004–1009, 1977.