Mapping pipeline skeletons onto heterogeneous platforms

Anne Benoit, Yves Robert

▶ **To cite this version:**

Anne Benoit, Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. [Research Report] RR-6087, 2007, pp.25. inria-00122884v2

HAL Id: inria-00122884

https://hal.inria.fr/inria-00122884v2

Submitted on 8 Jan 2007 (v2), last revised 13 Feb 2007 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mapping pipeline skeletons onto heterogeneous platforms

Anne Benoit — Yves Robert

N° 6087

January 2007

Thème NUM

apport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6087--FR+ENG

Mapping pipeline skeletons onto heterogeneous platforms

Anne Benoit , Yves Robert

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 6087 — January 2007 — 25 pages

Abstract: Mapping applications onto parallel platforms is a challenging problem, that becomes even more difficult when platforms are heterogeneous –nowadays a standard assumption. A high-level approach to parallel programming not only eases the application developer's task, but it also provides additional information which can help realize an efficient mapping of the application.

In this paper, we discuss the mapping of pipeline skeletons onto different types of platforms: *Fully Homogeneous* platforms with identical processors and interconnection links; *Communication Homogeneous* platforms, with identical links but different speed processors; and finally, *Fully Heterogeneous* platforms. We assume that a pipeline stage must be mapped on a single processor, and we establish new theoretical complexity results for different mapping policies: a mapping can be required to be one-to-one (a processor is assigned at most one stage), or interval-based (a processor is assigned an interval of consecutive stages), or fully general. We provide several efficient polynomial heuristics for the most important policy/platform combination, namely interval-based mappings on *Communication Homogeneous* platforms. These heuristics are compared to the optimal result provided by the formulation of the problem in terms of the solution of an integer linear program, for small problem instances.

Key-words: Algorithmic skeletons, pipeline, scheduling, complexity results, heuristics, heterogeneous clusters.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Mapping de squelettes pipeline sur plate-formes hétérogènes

Résumé : L'ordonnancement d'applications sur des plates-formes parallèles est un problème difficile, et ce d'autant plus si ces plates-formes sont hétérogènes. Une approche de haut niveau à la programmation parallèle, à base de squelettes algorithmiques, permet tout à la fois de faciliter la tâche du développeur, et d'acquérir des informations structurelles supplémentaires sur l'application, qui permettront d'obtenir un meilleur résultat.

Dans ce rapport, on discute l'ordonnancement d'applications sous forme du squelette *pipeline* sur différent types de plates-formes: les plates-formes totalement homogènes (processeurs et liens de communication identiques), les plates-formes à communications homogènes mais avec des vitesses de processeurs différentes, et les plates-formes totalement hétérogènes. On suppose qu'une étape de pipeline doit être placée sur un unique processeur, et on établit de nouveaux résultats de complexité pour différentes stratégies d'allocation: on peut imposer qu'au plus une étape de pipeline soit allouée à chaque processeur, ou bien allouer un intervalle d'étapes consécutives aux processeurs. Une troisième politique ne fixe aucune contrainte d'allocation.

Nous donnons de nombreuses heuristiques polynomiales efficaces pour la combinaison stratégie/plate-forme la plus importante, à savoir le placement par intervalle sur plates-formes à communications homogènes. Ces heuristiques sont comparées au résultat optimal obtenu par une formulation du problème sous forme de programme linéaire, pour de petites instances du problème.

Mots-clés : Squelettes algorithmiques, pipeline, ordonnancement, complexité, heuristiques, grappes de calcul hétérogènes.

Contents

1 Introduction

Mapping applications onto parallel platforms is a difficult challenge. Several scheduling and load-balancing techniques have been developed for homogeneous architectures (see [22] for a survey) but the advent of heterogeneous clusters has rendered the mapping problem even more difficult. Typically, such clusters are composed of different-speed processors interconnected either by plain Ethernet (the low-end version) or by a high-speed switch (the high-end counterpart), and they constitute the experimental platform of choice in most academic or industry research departments.

In this context of heterogeneous platforms, a structured programming approach rules out many of the problems which the low-level parallel application developer is usually confronted to, such as deadlocks or process starvation. Moreover, many real applications draw from a range of well-known solution paradigms, such as pipelined or farmed computations. High-level approaches based on algorithmic skeletons [11, 20] identify such patterns and seeks to make it easy for an application developer to tailor such a paradigm to a specific problem. A library of skeletons is provided to the programmer, who can rely on these already coded patterns to express the communication scheme within its own application. Moreover, the use of a particular skeleton carries with it considerable information about implied scheduling dependencies, which we believe can help to address the complex problem of mapping a distributed application onto a heterogeneous platform.

In this paper, we therefore consider applications that can be expressed as algorithmic skeletons, and we focus on the pipeline skeleton, which is one of the most widely used. In such applications, a series of tasks enter the input stage and progress from stage to stage until the final result is computed. Each stage has its own communication and computation requirements: it reads an input file from the previous stage, processes the data and outputs a result to the next stage. For each task, initial data is input to the first stage, and final results are output from the last stage. The pipeline operates in synchronous mode: after some latency due to the initialization delay, a new task is completed every period. The period is defined as the longest cycle-time to operate a stage, and is the inverse of the throughput that can be achieved.

The problem of mapping pipeline skeletons onto parallel platforms has received some attention, and we survey related work in Section 7. In this paper, we target heterogeneous clusters, and aim at deriving optimal mappings, i.e. mappings which minimize the period, or equivalently maximize the throughput, of the system. Each pipeline stage can be seen as a sequential procedure which may perform disc accesses or write data in the memory for each task. This data may be reused from one task to another, and thus the rule of the game is always to process the tasks in a sequential order within a stage. Moreover, due to the possible local memory accesses, a given stage must be mapped onto a single processor: we cannot process half of the tasks on a processor and the remaining tasks on another without exchanging intra-stage information, which might be costly and difficult to implement.

In this paper, we focus on pipeline skeletons and thus we enforce the rule that a given stage is mapped onto a single processor. In other words, a processor that is assigned a stage will execute the operations required by this stage (input, computation and output) for all the tasks fed into the pipeline. The optimization problem can be stated informally as follows: which stage to assign to which processor? We consider several variants, in which we require the mapping to be one-to-one (a processor is assigned at most one stage), or interval-based (a processor is assigned an interval of consecutive stages), or fully general.

In addition to these three mapping categories, we target three different platform types. First, *Fully Homogeneous* platforms have identical processors and interconnection links. Next, *Communication Homogeneous* platforms, with identical links but different speed processors, introduce a first degree of heterogeneity. Finally, *Fully Heterogeneous* platforms constitute the most difficult instance, with different speed processors and different capacity links. The main objective of the paper is to assess the complexity of each mapping variant onto each platform type. We establish several new complexity results for this important optimization problem, and we derive efficient polynomial heuristics for interval-based mappings onto *Communication Homogeneous* platforms. These heuristics are compared through simulation; moreover, their absolute performance is assessed owing to the formulation of the problem in terms of an integer linear program, whose solution returns the optimal result for small problem instances.

The rest of the paper is organized as follows. Section 2 is devoted to a detailed presentation of the target optimization problems. Next in Section 3 we proceed to the complexity results. In Section 4 we introduce several polynomial heuristics to solve the mapping problem. These heuristics are compared through simulations, whose results are analyzed in Section 5. Section 6 introduces the linear formulation of the problem and assesses the absolute performance of the heuristics when the optimal solution can be found. Section 7 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 8.

2 Framework

We outline in this section the characteristics of the applicative framework, as well as the model for the target platform. Next we detail the objective function, chosen as the maximum period of a processor to execute all the pipeline stages assigned to it.

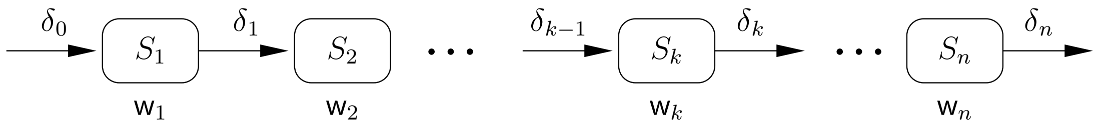
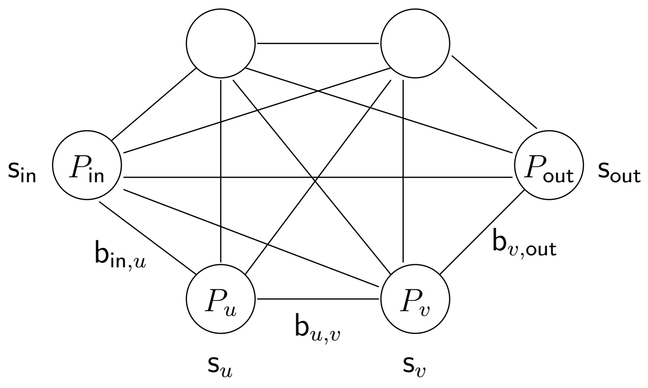


Figure 1: The application pipeline.



s_{in}
P_{in}
P_{out}
$\mathsf{s}_{\mathsf{out}}$
$\mathsf{b}_{\mathsf{in},u}$
$\mathsf{b}_{v,\mathsf{out}}$
P_u
P_v
$\mathsf{b}_{u,v}$
s_u
s_v

Figure 2: The target platform.

2.1 Applicative framework

We consider a pipeline of n stages $\mathcal{S}_k$, $1 \leq k \leq \mathsf{n}$, as illustrated on Figure 1. Tasks are fed into the pipeline and processed from stage to stage, until they exit the pipeline after the last stage.

The k-th stage $\mathcal{S}_k$ receives an input from the previous stage, of size δ_{k-1}, performs a number of w_k computations, and outputs data of size δ_k to the next stage. The first stage $\mathcal{S}_1$ receives an input of size δ_0 from the outside world, while the last stage $\mathcal{S}_\mathsf{n}$ returns the result, of size δ_n, to the outside world.

2.2 Target platform

We target a heterogeneous platform (see Figure 2), with p processors P_u, $1 \leq u \leq \mathsf{p}$, fully interconnected as a (virtual) clique. There is a bidirectional link $\mathsf{link}_{u,v} : P_u \rightarrow P_v$ between any processor pair P_u and P_v, of bandwidth $\mathsf{b}_{u,v}$. Note that we do not need to have a physical link between any processor pair. Instead, we may have a switch, or even a path composed of several physical links, to interconnect P_u and P_v; in the latter case we would retain the bandwidth of the slowest link in the path for the value of $\mathsf{b}_{u,v}$.

Communications contention is taken care of by enforcing the *one-port* model [9, 10]. In this model, a given processor can be involved in a single communication at any time-step, either a send or a receive. However, independent communications between distinct processor pairs can take place simultaneously. The one-port model seems to fit the performance of some current MPI implementations, which serialize asynchronous MPI sends as soon as message sizes exceed a few megabytes [21].

In the most general case, we have fully heterogeneous platforms, with different processors speeds and link capacities. The speed of processor P_u is denoted as s_u, and it takes X/s_u time-units for P_u to execute X floating point operations. We also enforce a linear cost model for communications, hence it takes $X/\mathsf{b}_{u,v}$ time-units to send (resp. receive) a message of size X to (resp. from) P_v. We classify below particular cases which are important, both from a theoretical and practical perspective:

Fully Homogeneous– These platforms have identical processors ($\mathsf{s}_u = \mathsf{s}$) and links ($\mathsf{b}_{u,v} = \mathsf{b}$). They represent typical parallel machines.

Communication Homogeneous– These platforms have different-speed processors ($\mathsf{s}_u \neq \mathsf{s}_v$) interconnected by links of same capacities ($\mathsf{b}_{u,v} = \mathsf{b}$). They correspond to networks of workstations with plain TCP/IP interconnects or other LANs.

Fully Heterogeneous– These are the most general, fully heterogeneous architectures, with $\mathsf{s}_u \neq \mathsf{s}_v$ and $\mathsf{b}_{u,v} \neq \mathsf{b}_{u',v'}$. Hierarchical platforms made up with several clusters interconnected by slower backbone links can be modeled this way.

Finally, we assume that two special additional processors P_{in} and P_{out} are devoted to input/output data. Initially, the input data for each task resides on P_{in}, while all results must be returned to and stored in P_{out}. Of course we may have a single processor acting as the interface for the computations, i.e. $P_{\mathsf{in}} = P_{\mathsf{out}}$.

2.3 Mapping problem

The general mapping problem consists in assigning application stages to platform processors. However, some constraints can be added to the mapping to ease the implementation of the application, for instance by imposing to map a single stage onto each processor. Different instances of the mapping problem are discussed below.

2.3.1 One-to-one Mapping

Assume temporarily, for the sake of simplicity, that each stage $\mathcal{S}_k$ of the application pipeline is mapped onto a distinct processor $P_{\mathsf{alloc}(k)}$ (which is possible only if $\mathsf{n} \leq \mathsf{p}$). For convenience, we create two fictitious stages $\mathcal{S}_0$ and $\mathcal{S}_{\mathsf{n}+1}$, and we assign $\mathcal{S}_0$ to P_{in} and $\mathcal{S}_{\mathsf{n}+1}$ to P_{out}.

What is the period of $P_{\mathsf{alloc}(k)}$, i.e. the minimum delay between the processing of two consecutive tasks? To answer this question, we need to know which processors the previous and next stages are assigned to. Let $t = \mathsf{alloc}(k-1)$, $u = \mathsf{alloc}(k)$ and $v = \mathsf{alloc}(k+1)$. P_u needs $\delta_{k-1}/\mathsf{b}_{t,u}$ to receive the input data from P_t, $\mathsf{w}_k/\mathsf{s}_u$ to process it, and $\delta_k/\mathsf{b}_{u,v}$ to send the result to P_v, hence a cycle-time of $\delta_{k-1}/\mathsf{b}_{t,u} + \mathsf{w}_k/\mathsf{s}_u + \delta_k/\mathsf{b}_{u,v}$ for P_u. The *period* achieved with the mapping is the maximum of the cycle-times of the processors, this corresponds to the rate at which the pipeline can be activated.

In this simple instance, the optimization problem can be stated as follows: determine a one-to-one allocation function $\mathsf{alloc} : [1, \mathsf{n}] \to [1, \mathsf{p}]$ (augmented with $\mathsf{alloc}(0) = \mathsf{in}$ and $\mathsf{alloc}(\mathsf{n} + 1) = \mathsf{out}$) such that

$$T_{\mathsf{period}} = \max_{1 \le k \le \mathsf{n}} \left\{ \frac{\delta_{k-1}}{\mathsf{b}_{\mathsf{alloc}(k-1),\mathsf{alloc}(k)}} + \frac{\mathsf{w}_k}{\mathsf{s}_{\mathsf{alloc}(k)}} + \frac{\delta_k}{\mathsf{b}_{\mathsf{alloc}(k),\mathsf{alloc}(k+1)}} \right\} \tag{1}$$

is minimized. We denote by One-to-one Mapping the previous optimization problem.

2.3.2 Interval Mapping

However, one-to-one mappings may be unduly restrictive. A natural extension is to search for interval mappings, i.e. allocation functions where each participating processor is assigned an interval of consecutive stages. Intuitively, assigning several consecutive tasks to the same processors will increase their computational load, but may well dramatically decrease communication requirements. In fact, the best interval mapping may turn out to be a one-to-one mapping, or instead may enroll only a very small number of fast computing processors interconnected by high-speed links.

Interval mappings constitute a natural and useful generalization of one-to-one mappings (not to speak of situations where $\mathsf{p} < \mathsf{n}$, where interval mappings are mandatory). A major objective of this paper is to assess the performance of general interval mappings as opposed to pure one-to-one allocations.

For the sake of completeness, we formally write the optimization problem associated to interval mappings. We need to express that the intervals achieve a partition of the original set of stages $\mathcal{S}_1$ to $\mathcal{S}_\mathsf{n}$. We search for a partition of $[1..\mathsf{n}]$ into m intervals $I_j = [d_j, e_j]$ such that $d_j \le e_j$ for $1 \le j \le m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \le j \le m - 1$ and $e_m = \mathsf{n}$. Interval I_j is mapped onto processor $P_{\mathsf{alloc}(j)}$, and the period is expressed as

$$T_{\mathsf{period}} = \max_{1 \le j \le m} \left\{ \frac{\delta_{d_j-1}}{\mathsf{b}_{\mathsf{alloc}(j-1),\mathsf{alloc}(j)}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{\mathsf{s}_{\mathsf{alloc}(j)}} + \frac{\delta_{e_j}}{\mathsf{b}_{\mathsf{alloc}(j),\mathsf{alloc}(j+1)}} \right\} \tag{2}$$

Here, we assume that $\mathsf{alloc}(0) = \mathsf{in}$ and $\mathsf{alloc}(m + 1) = \mathsf{out}$. The optimization problem Interval Mapping is to determine the best mapping, over all possible partitions into intervals, and over all processor assignments.

2.3.3 General Mapping

The most general mappings may be more complicated than interval-based mappings: a processor P_u can be assigned any subset of stages.

Let us consider the following example with $\mathsf{n} = 3$ stages and $\mathsf{p} = 2$ processors. We also use a third processor $P_0 = P_{\mathsf{in}} = P_{\mathsf{out}}$ for input/output, with no processing capacity ($\mathsf{s}_0 = 0$).

If the platform is *Fully Homogeneous*, let us consider a case in which the second stage requires two times more computation than the other stages: $\mathsf{w}_1 = \mathsf{w}_3 = 1$ and $\mathsf{w}_2 = 2$, and where the communications are negligible ($\delta_i = 0$). The platform parameters (s, b) are set to 1. An interval-based mapping needs to map either the two first stages or the two last ones on a same processor, which will have a cycle time of $1 + 2 = 3$ (computations of the two stages). An intuitive solution would however be to map the first and third stages on one processor, and the second stage alone, relaxing the interval constraint, hoping to reach a cycle time of 2 for both processors (with a good load-balancing), which will work in parallel. Figure 3 illustrates such a mapping, and the desired behavior of the application.

During a cycle, P_1 processes a task for S_3 and then a task for S_1 (communications from/to P_0 can be neglected), while P_2 processes a task for S_2. Communications occur between each cycle, when P_1 sends output from S_1 and P_2 sends output from S_2.

However, we always assumed in this paper that a stage is implemented in a static and synchronous way, using the one-port model, which is a realistic hypothesis on the implementation of

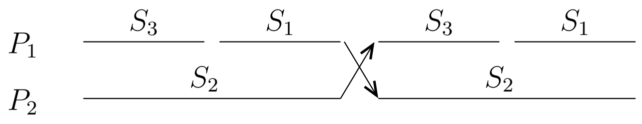


Figure 3: Example of general mapping.

the application. With such a model, if P_1 starts to send the output from S_1, it will be blocked until P_2 completes the receive, while P_2 is sending its own output before receiving the next task for S_2. This leads to a deadlock with a straightforward implementation.

The solution would be to create a process for each stage, and map the independent processes on processors, possibly several processes on a single processor. In order to obtain asynchronous communications, each process should run three threads, one for input, one for computation and one for output. Each thread knows where to expect its input from, and where to redirect its output to, so it can post asynchronous messages and poll for completion. When an input is completed, it hands over the data to the computation thread, and so on. Overall, the computation advances asynchronously, in a greedy fashion: each action is executed as soon as it is ready. Computations and communications are preemptive. On each processor and at any time step, several processes share the CPU for computations related to different stages, and several communications take place simultaneously and share the bandwidth (of the link and/or the network card). This scheme requires as many communication buffers as there are stages/processes assigned to the processor.

With such an implementation, we would like to define the cycle-time of a processor by the time needed to execute one instance of each of its stages, and the period as the maximum of the processor cycle-times. If P_u handles the set of stages $\mathsf{stages}(u)$, and stage i is mapped on processor $\mathsf{alloc}(i)$, the period is defined by:

$$T_{\mathsf{period}} = \max_{1 \le u \le \mathsf{p}} \left\{ \sum_{i \in \mathsf{stages}(u)} \left(\Delta^{u}_{\mathsf{alloc}(i-1)} \frac{\delta_{i-1}}{\mathsf{b}_{\mathsf{alloc}(i-1),u}} + \frac{\mathsf{w}_i}{\mathsf{s}_u} + \Delta^{\mathsf{alloc}(i+1)}_{u} \frac{\delta_i}{\mathsf{b}_{u,\mathsf{alloc}(i+1)}} \right) \right\} \tag{3}$$

where $\Delta^v_u = 1$ if $u \neq v$ and 0 otherwise. In equation (3), we pay the input communication for a stage $\mathcal{S}_i$ only if stage $\mathcal{S}_{i-1}$ is not mapped on the same processor (and similarly for the output communication).

However, it is very difficult to assess whether the above period will actually be achieved in a real-life implementation. Delays may occur due to the dependency paths (P_u sending data to P_v that returns it to P_u, and the like), leading to idle times (waiting on polling). Races may occur and slow down, say, two simultaneous communications involving non-disjoint processor pairs. Finally, control overhead and cost of switching processes may prove very high. At least an optimistic view is that the achieved period will be only slightly larger than the above theoretical bound, but this would require to be validated experimentally.

If we restrict ourselves to the one-port model, and consider the 3-stages example again with the first and third stages mapped on the first processor, the straightforward implementation leads to a period of 4 because P_1 always wait for a task to go through S_2 on P_2 before it goes through P_3. This leads to the execution scheme of Figure 4.

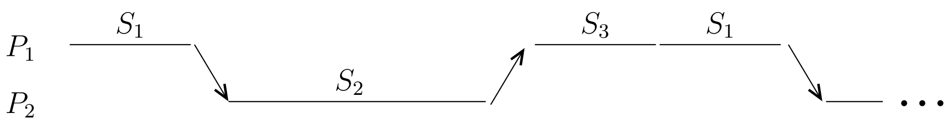


Figure 4: Example of general mapping with the one-port model.

In this execution scheme, we enforced a natural rule that is implied by a non-preemptive implementation. Let $\mathcal{S}_i(k)$ denote the processing in stage $\mathcal{S}_i$ for the incoming task number k. Then, if $\mathcal{S}_i$ and $\mathcal{S}_j$ are two stages assigned to P_u with $i < j$, then P_u executes $\mathcal{S}_j(k)$ before

$\mathcal{S}_i(k+1)$. With this rule, the cycle-time of a processor is the duration of the path that goes from the input of its first stage all the way to the output of its last stage. Let us define $\mathsf{first}(u)$ and $\mathsf{last}(u)$ as the first and last index, respectively, which are assigned to P_u.

$$\mathsf{cycle\text{-}time}(P_u) = \frac{\delta_{\mathsf{first}(u)-1}}{\mathsf{b}_{\mathsf{alloc}(\mathsf{first}(u)-1),u}} + \sum_{i=\mathsf{first}(u)}^{\mathsf{last}(u)} \left(\frac{\mathsf{w}_i}{\mathsf{s}_{\mathsf{alloc}(i)}} + \Delta_{\mathsf{alloc}(i)}^{\mathsf{alloc}(i+1)} \frac{\delta_i}{\mathsf{b}_{\mathsf{alloc}(i),\mathsf{alloc}(i+1)}} \right) \quad (4)$$

$$T_{\mathsf{period}} = \max_{1 \le u \le \mathsf{p}} \mathsf{cycle\text{-}time}(P_u) \quad (5)$$

In equation (4), we always pay the first communication and the last one, by definition of $\mathsf{first}(u)$ and $\mathsf{last}(u)$, but we need to take care that some consecutive stages might be mapped onto the same processor.

Obviously, this period is longer than the one obtained with equation (3), but with these assumptions, we know how to implement the application in a natural way, and we can ensure that this (longer) period will be achieved. All actions can be organized (scheduled statically) and the period can be characterized analytically. This allows us to avoid the complex dynamic scheme.

In this case, it does not seem very interesting to map non consecutive stages on a same processor, because it has to wait for the processing of the intermediate stages, which leads to idle time. Actually, we will see in Section 3 that interval-based mappings are always as good as general mappings on *Communication Homogeneous* platforms. We still consider the general optimization problem, since general mappings may outperform interval-based ones in some particular cases, even with the definition of the period of equation (5).

Consider the following example with $\mathsf{n} = 3$ stages and $\mathsf{p} = 2$ processors. The target architecture is characterized by $\mathsf{s}_1 = \mathsf{b}_{0,2} = \mathsf{b}_{1,2} = 1$ and $\mathsf{s}_2 = \mathsf{b}_{0,1} = 10$. As for the application, we have $\delta_0 = \delta_3 = \mathsf{w}_2 = 10$ and $\delta_1 = \delta_2 = \mathsf{w}_1 = \mathsf{w}_3 = 1$. If we map $\mathcal{S}_2$ on P_1, then $T_{\mathsf{period}} \ge \frac{\mathsf{w}_2}{\mathsf{s}_1} = 10$. If we map $\mathcal{S}_1$ on P_2, then $T_{\mathsf{period}} \ge \frac{\delta_0}{\mathsf{b}_{0,2}} = 10$. Similarly, if we map $\mathcal{S}_3$ on P_2, then $T_{\mathsf{period}} \ge \frac{\delta_3}{\mathsf{b}_{0,2}} = 10$. There exists a single mapping whose period is smaller than 10: indeed, we can map $\mathcal{S}_1$ and $\mathcal{S}_3$ on P_1 and $\mathcal{S}_2$ on P_2, and the period becomes

$$T_{\mathsf{period}} = \frac{\delta_0}{\mathsf{b}_{0,1}} + \frac{\mathsf{w}_1}{\mathsf{s}_1} + \frac{\delta_1}{\mathsf{b}_{1,2}} + \frac{\mathsf{w}_2}{\mathsf{s}_2} + \frac{\delta_2}{\mathsf{b}_{1,2}} + \frac{\mathsf{w}_3}{\mathsf{s}_1} + \frac{\delta_3}{\mathsf{b}_{0,1}} = 7 \quad (6)$$

In equation (6), we have computed the length of the cycle that P_1 repeats for every incoming task:

- read data for task number k from P_{in}
- compute $\mathcal{S}_1(k)$, i.e. stage 1 for that task
- send data to P_2, wait for P_2 to process it (stage $\mathcal{S}_2(k)$) and to return the results
- compute stage $\mathcal{S}_3(k)$, send output data to P_{out}
- proceed to task number $k+1$ and repeat entire cycle, starting with input data for $\mathcal{S}_1(k+1)$

Because the operation of P_2 is entirely included into that of P_1, P_1 has the longest cycle-time, thereby defining the period. This is how we derived that $T_{\mathsf{period}} = 7$ in the example, in full accordance to equations (4) and (5).

This little example shows that general mappings may be superior to interval-based mappings, and provides a motivation to study the corresponding optimization problem, denoted as GENERAL MAPPING.

In this paper, we mostly concentrate on interval-based mappings, because they realize the best trade-off between efficiency and simplicity. One-to-one mappings are not general enough, in particular when resources are in limited number, or when communications have a higher cost. General mappings require important changes in the model and its implementation, and may be too complex to deal with efficiently for the application programmer. We still address the complexity of all mapping problems, including general mappings, but for practical developments we limit ourselves to one-to-one and interval-based mappings. Also, we privilege *Communication Homogeneous* platforms, which are the most representative of current experimental architectures.

	Fully Homogeneous	***Comm. Homogeneous***	***Fully Heterogeneous***
One-to-one Mapping	polynomial (bin. search)	polynomial (bin. search)	NP-complete
Interval Mapping	polynomial (dyn. prog.)	open	NP-complete
General Mapping	same complexity as INTERVAL MAPPING		NP-complete

Table 1: Complexity results for the different instances of the mapping problem.

3 Complexity results

To the best of our knowledge, this work is the first to study the complexity of the various mapping strategies (ONE-TO-ONE MAPPING, INTERVAL MAPPING and GENERAL MAPPING), for each of the different platform categories (*Fully Homogeneous*, *Communication Homogeneous* and *Fully Heterogeneous*). Table 1 summarizes all our new results. All the entries are filled, except for the complexity of the *Communication Homogeneous*/INTERVAL MAPPING combination, which remains open.

For *Fully Homogeneous* or *Communication Homogeneous* platforms, determining the optimal ONE-TO-ONE MAPPING can be achieved through a binary search over possible periods, invoking a greedy algorithm at each step. The problem surprisingly turns out to be NP-hard for *Fully Heterogeneous* platforms. The binary search algorithm for ONE-TO-ONE MAPPING is outlined in Section 3.1.

The INTERVAL MAPPING problem is more complex to deal with. We have designed a dynamic programming algorithm for *Fully Homogeneous* platforms (see Section 3.2), but the complexity of the problem remains open for *Communication Homogeneous* platforms. However, in Section 3.2, we also prove the nice theoretical result that interval-based mappings are dominant for *Communication Homogeneous* platforms: no need to consider general mappings for such platforms, we can restrict the search to interval-based mappings.

Finally, all three optimization problems are NP-hard for *Fully Heterogeneous* platforms. The proof of these results is provided in Section 3.3.

3.1 One-to-one Mapping

Theorem 1. *For Fully Homogeneous and Communication Homogeneous platforms, the optimal* ONE-TO-ONE MAPPING *can be determined in polynomial time.*

Proof. We provide a constructive proof: we outline a binary-search algorithm that iterates until the optimal period is found. At each step, a greedy algorithm is used to assign stages to processors in a one-to-one fashion. The greedy algorithm succeeds if and only if the period is feasible. If the algorithm does succeed, we decrease the target period, otherwise we increase it, and then proceed to the next step. For theory-oriented readers, we easily check that the number of steps is indeed polynomial in the problem size: in a word, we use a binary search over an interval whose length can be bounded by the maximum values of the application/platform parameters, hence a number of steps proportional to the logarithm of this value, which in turn is the encoding size of the problem. For more practice-oriented readers, only a few seconds are needed to obtain the period with a precision of 10^{-4} for a reasonable problem size. See for instance the practical experiments reported in Section 5.

Let us now describe the most interesting part of the procedure, namely the greedy assignment algorithm for a prescribed value T_{period} of the achievable period. Recall that there are n stages to map onto $\mathsf{p} \geq \mathsf{n}$ processors in a one-to-one fashion. Also, we target *Communication Homogeneous* platforms with different-speed processors ($\mathsf{s}_u \neq \mathsf{s}_v$) but with links of same capacities ($\mathsf{b}_{u,v} = \mathsf{b}$).

First we retain only the fastest n processors, which we rename P_1, P_2, ..., P_n such that $\mathsf{s}_1 \leq \mathsf{s}_2 \leq \ldots \leq \mathsf{s}_\mathsf{n}$. Then we consider the processors in the order P_1 to P_n, i.e. from the slowest to the fastest, and greedily assign them any free (non already assigned) task that they can process within the period. Algorithm 1 details the procedure.

```
procedure Greedy Assignment
begin
    Work with fastest $\mathsf{n}$ processors, numbered $P_1$ to $P_{\mathsf{n}}$, where $\mathsf{s}_1 \leq \mathsf{s}_2 \leq \ldots \leq \mathsf{s}_{\mathsf{n}}$
    Mark all stages $\mathcal{S}_1$ to $\mathcal{S}_{\mathsf{n}}$ as free
    for $u = 1$ to $n$ do
        Pick up any free stage $\mathcal{S}_k$ s.t. $\delta_{k-1}/\mathsf{b} + \mathsf{w}_k/\mathsf{s}_u + \delta_k/\mathsf{b} \leq T_{\mathsf{period}}$
        Assign $\mathcal{S}_k$ to $P_u$. Mark $\mathcal{S}_k$ as already assigned
        If no stage found return "failure"
    end
end
```

Algorithm 1: Greedy assignment algorithm for a given period T_{period}.

The proof that the greedy algorithm returns a solution if and only if there exists a solution of period T_{period} is done by a simple exchange argument. Indeed, consider a valid one-to-one assignment of period T_{period}, denoted A, and assume that it has assigned stage S_{k_1} to P_1. Note first that the greedy algorithm will indeed find a stage to assign to P_1 and cannot fail, since S_{k_1} can be chosen. If the choice of the greedy algorithm is actually S_{k_1}, we proceed by induction with P_2. If the greedy algorithm has selected another stage S_{k_2} for P_1, we find which processor, say P_u, has been assigned this stage in the valid assignment A. Then we exchange the assignments of P_1 and P_u in A. Because P_u is faster than P_1, which could process S_{k_1} in time in the assignment A, P_u can process S_{k_1} in time too. Because S_{k_2} has been mapped on P_1 by the greedy algorithm, P_1 can process S_{k_1} in time. So the exchange is valid, we can consider the new assignment A which is valid and which did the same assignment on P_1 than the greedy algorithm. The proof proceeds by induction with P_2 as before. We point out that the complexity of the greedy algorithm is bounded by $O(\mathsf{n}^2)$, because of the two loops over processors and stages. □

3.2 Interval Mapping

Theorem 2. *For Fully Homogeneous platforms, the optimal* Interval Mapping *can be determined in polynomial time.*

Proof. We provide a constructive proof: we outline a dynamic programming algorithm that returns the optimal period. Consider an application with n stages $\mathcal{S}_1$ to $\mathcal{S}_{\mathsf{n}}$ to be mapped onto a fully-homogeneous platform composed of p (identical) processors. Let s and b respectively denote the processor speed and the link bandwidth.

We compute recursively the value of $c(i, j, k)$, which is the optimal period that can be achieved by any interval-based mapping of stages S_i to S_j using exactly k processors. The goal is to determine

$$\min_{1 \leq k \leq \mathsf{p}} c(1, \mathsf{n}, k)$$

The recurrence relation can be expressed as

$$c(i,j,k) = \min_{\substack{q+r=k \\ 1 \leq q \leq k-1 \\ 1 \leq r \leq k-1}} \left\{ \min_{i \leq \ell \leq j-1} \left\{ \max\left(c(i,\ell,q), c(\ell+1,j,r)\right) \right\} \right\}$$

with the initialization

$$c(i,j,1) = \frac{\delta_{i-1}}{\mathsf{b}} + \frac{\sum_{k=i}^{j} \mathsf{w}_k}{\mathsf{s}} + \frac{\delta_j}{\mathsf{b}}$$

$$c(i,j,k) = +\infty \quad \text{if} \quad k > j - i + 1$$

The recurrence is easy to justify: to compute $c(i, j, k)$, we search over all possible partitionings into two subintervals, using every possible number of processors for each interval. The complexity of this dynamic programming algorithm is bounded by $O(\mathsf{n}^3\mathsf{p}^2)$. □

We have not been able to extend the previous dynamic programming algorithm to deal with *Communication Homogeneous* platforms. This is because the algorithm intrinsically relies on identical processors in the recurrence computation. Different-speed processors would execute sub-intervals with different cycle-times. Because of this additional difficulty, the Interval Mapping problem for *Communication Homogeneous* platforms seems to be very combinatorial, and we conjecture that it is NP-hard. At least we can prove that interval-based mappings are dominant for such platforms, which means that there exist interval-based mappings which are optimal among all possible general mappings (hence the entry titled *same complexity* in Table 1):

Theorem 3. *For Communication Homogeneous platforms, interval-based mappings are dominant.*

Proof. Consider an application with n stages $\mathcal{S}_1$ to $\mathcal{S}_n$ to be mapped onto a communication-homogeneous platform composed of p processors P_1 to P_{p}. The speed of P_i is s_i, and all links have same bandwidth b. Consider an optimal mapping for this application. If this mapping is interval-based, we are done. Otherwise, there exist processors which are not assigned a single interval of stages. Let i_1 be the index of the first stage $\mathcal{S}_{i_1}$ that is mapped onto such a processor, say P_u, and let i_2 be the last index such that the interval $[i_1, i_2]$ is mapped on P_u. Stages $\mathcal{S}_1$ to $\mathcal{S}_{i_1-1}$ are mapped by intervals onto other processors.

Rather than going on formally, let us illustrate the algorithm through the following example, where we have $i_1 = 6$, $i_2 = 7$ and $u = 4$:

$$\begin{array}{lcccccccccc}
P_1 & \underline{\mathcal{S}_1\mathcal{S}_2} & & & & & & & & & \\
P_2 & & \underline{\mathcal{S}_3} & & & & & & & & \\
P_3 & & & \underline{\mathcal{S}_4\mathcal{S}_5} & & & & & & & \\
P_4 & & & & \underline{\mathcal{S}_6\mathcal{S}_7} & & & & \underline{\mathcal{S}_{13}\mathcal{S}_{14}\mathcal{S}_{15}} & & \dots \\
P_5 & & & & & \underline{\mathcal{S}_8} & & & & & \\
P_6 & & & & & & \underline{\mathcal{S}_9\mathcal{S}_{10}} & & & \dots & \\
P_7 & & & & & & & \underline{\mathcal{S}_{11}\mathcal{S}_{12}} & & &
\end{array}$$

The idea is to transform the mapping so that P_4 will be assigned a single interval, without increasing the cycle-time of any processor. Recall that we use equations (4) and (5) to compute the period of a general mapping. There are two cases to consider:

- If the speed of P_4 is larger than (or at least equal to) the speed of P_5, P_6 and P_7, then we assign stages $\mathcal{S}_8$ to $\mathcal{S}_{12}$ to P_4. This will indeed decrease its cycle-time, because these stages are processed in shorter time and no more communication is paid in between. Then we iterate the transformation with $i_2 = 15$.

- Otherwise, choose the fastest processor among P_4, P_5, P_6 and P_7. Assume that it is P_6. Then P_6 is assigned the extra stages $\mathcal{S}_{11}$ to $\mathcal{S}_{15}$, and all the other subsequent stages that were assigned to P_4. Because the communications are homogeneous, nothing is changed in the price paid by the following communications between P_4 and other processors that now go between P_6 and these processors. The cycle-time of P_4 has been reduced since we removed stages from it. The cycle-time of P_6 has not increased the period of the new mapping compared to the initial mapping. To see this, note that its first assigned stage $\mathsf{first}(6)$ is the same. If its last assigned stage was already assigned to it before the transformation ($\mathsf{last}(6)$), then its cycle-time has decreased (because some stages are processed faster in between) or is unchanged. If its last assigned stage was originally assigned to P_4 ($\mathsf{last}(4)$), then its new cycle-time is smaller than the old cycle-time of P_4 since $\mathsf{first}(6) > \mathsf{first}(4)$ and some stages are eventually processed faster between $\mathsf{first}(6)$ and $\mathsf{last}(4)$. In both cases, we safely increase the value of i_1 and proceed to the next transformation, without increasing the period of the mapping.

After a finite number of such transformations, we obtain an interval-based mapping, whose period does not exceed the period of the original mapping, which concludes the proof. □

3.3 *Fully Heterogeneous* platforms

Theorem 4. *For Fully Heterogeneous platforms, the (decision problems associated to the) three mapping strategies (*ONE-TO-ONE MAPPING*,* INTERVAL MAPPING *and* GENERAL MAPPING*) are all NP-complete.*

Proof. For all strategies, the problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that all stages are processed, and that the maximum cycle-time of any processor does not exceed the bound on the period. To establish the completeness, we use a reduction from MINIMUM METRIC BOTTLENECK WANDERING SALESPERSON PROBLEM (MMBWSP) [1, 12]. We consider an instance $\mathcal{I}_1$ of MMBWSP: given a set $C = \{c_1, c_2, \ldots, c_m\}$ of m cities, an initial city $s \in C$, a final city $f \in C$, distances $d(c_i, c_j) \in \mathbb{N}$ satisfying the triangle inequality, and a bound $K \geq 2$ on the largest distance, does there exist a simple path from the initial city s to the final city f passing through all cities in C, i.e. a permutation $\pi : [1..m] \to [1..m]$ such that $c_{\pi(1)} = s$, $c_{\pi(m)} = f$, and $d(c_{\pi(i)}, c_{\pi(i+1)}) \leq K$ for $1 \leq i \leq m-1$? To simplify notations, and without loss of generality, we renumber cities so that $s = c_1$ and $f = c_m$ (i.e. $\pi(1) = 1$ and $\pi(m) = m$).

We build the following instance $\mathcal{I}_2$ of our mapping problem (note that the same instance will work out for the three variants ONE-TO-ONE MAPPING, INTERVAL MAPPING and GENERAL MAPPING):

- For the application: $\mathsf{n} = 2m - 1$ stages which for convenience we denote as

$$\to \mathcal{S}_1 \to \mathcal{S}'_1 \to \mathcal{S}_2 \to \mathcal{S}'_2 \to \ldots \to \mathcal{S}_{m-1} \to \mathcal{S}'_{m-1} \to \mathcal{S}_m \to$$

  For each stage $\mathcal{S}_i$ or $\mathcal{S}'_i$ we set $\delta_i = \mathsf{w}_i = 1$ (as well as $\delta_0 = \delta_{\mathsf{n}} = 1$), so that the application is perfectly homogeneous.

- For the platform (see Figure 5):
  - we use $\mathsf{p} = m + \binom{m}{2}$ processors which for convenience we denote as P_i, $1 \leq i \leq m$ and $P_{i,j}$, $1 \leq i < j \leq m$. We also use an input processor P_{in} and an output processor P_{out}. The speed of each processor P_i or P_{ij} has the same value $s = \frac{1}{2K}$ (note that we have a computation-homogeneous platform), except for P_{in} and P_{out} whose speed is 0.
  - the communication links shown on Figure 5 have a larger bandwidth than the others, and are referred to as *fast* links. More precisely, $b_{P_{\mathsf{in}},P_1} = b_{P_m,P_{\mathsf{out}}} = 1$, and $b_{P_i,P_{ij}} = b_{P_{ij},P_j} = \frac{2}{d(c_i,c_j)}$ for $1 \leq i < j \leq m$. All the other links have a very small bandwidth $\mathsf{b} = \frac{1}{5K}$ and are referred to as *slow* links. The intuitive idea is that slow links are too slow to be used for the mapping.

Finally, we ask whether there exists a solution with period $3K$. Clearly, the size of $\mathcal{I}_2$ is polynomial (and even linear) in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. We map stage $\mathcal{S}_i$ onto $P_{\pi(i)}$ for $1 \leq i \leq m$, and stage $\mathcal{S}'_i$ onto processor $P_{\pi(i),\pi(i+1)}$ for $1 \leq i \leq m-1$. The cycle-time of P_1 is $1 + 2K + \frac{d(c_{\pi(1)},c_{\pi(2)})}{2} \leq 1 + 2K + \frac{K}{2} \leq 3K$. Quite similarly, the cycle-time of P_m is smaller than $3K$. For $2 \leq i \leq m-1$, the cycle-time of $P_{\pi(i)}$ is $\frac{d(c_{\pi(i-1)},c_{\pi(i)})}{2} + 2K + \frac{d(c_{\pi(i)},c_{\pi(i+1)})}{2} \leq 3K$. Finally, for $1 \leq i \leq m-1$, the cycle-time of $P_{\pi(i),\pi(i+1)}$ is $\frac{d(c_{\pi(i)},c_{\pi(i+1)})}{2} + 2K + \frac{d(c_{\pi(i)},c_{\pi(i+1)})}{2} \leq 3K$. The mapping does achieve a period not greater than $3K$, hence a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution, i.e. a mapping of period not greater than $3K$. We first observe that each processor is assigned at most one stage by the mapping, because executing two stages would require at least $2K + 2K$ units of time, which would be too large to match the period. This simple observation explains why the same reduction works for the three strategies, ONE-TO-ONE MAPPING, INTERVAL MAPPING and GENERAL MAPPING. Next, we observe that any slow link of bandwidth $\frac{1}{5K}$ cannot be used in the solution: otherwise the period would exceed $5K$.

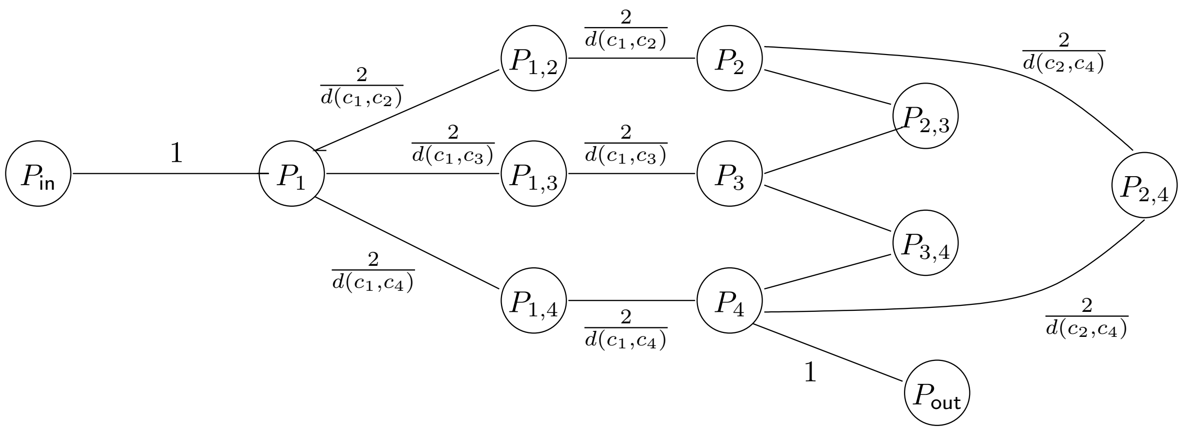


Figure 5: The platform used in the reduction for Theorem 4.

The input processor P_{in} has a single fast link to P_1, so necessarily P_1 is assigned stage $\mathcal{S}_1$ (i.e. $\pi(1) = 1$). As observed above, P_1 cannot execute any other stage. Because of fast links, stage $\mathcal{S}'_1$ must be assigned to some $P_{1,j}$; we let $j = \pi(2)$. Again, because of fast links and of the one-to-one constraint, the only choice for stage $\mathcal{S}_2$ is $P_{\pi(2)}$. Necessarily $j = \pi(2) \neq \pi(1) = 1$, otherwise P_1 would execute two stages.

We proceed similarly for stage S'_2, assigned to some P_{2k} (let $k = \pi(3)$) and stage S_3 assigned to $P_{\pi(3)}$. Owing to the one-to-one constraint, $k \neq 1$ and $k \neq j$, i.e. $\pi : [1..3] \rightarrow [1..m]$ is a one-to-one mapping. By induction, we build the full permutation $\pi : [1..m] \rightarrow [1..m]$. Because the output processor P_{out} has a single fast link to P_m, necessarily P_m is assigned stage $\mathcal{S}_m$, hence $\pi(m) = m$.

We have built the desired permutation, and there remains to show that $d(c_{\pi(i)}, c_{\pi(i+1)}) \leq K$ for $1 \leq i \leq m-1$. The cycle time of processor $P_{\pi(i)}$ is $\frac{d(c_{\pi(i)},c_{\pi(i+1)})}{2} + 2K + \frac{d(c_{\pi(i)},c_{\pi(i+1)})}{2} \leq 3K$, hence $d(c_{\pi(i)}, c_{\pi(i+1)}) \leq K$. Altogether, we have found a solution for $\mathcal{I}_1$, which concludes the proof.

□

4 Heuristics

In this section several heuristics for *Communication Homogeneous* platforms are presented. We restrict to such platforms because, as already pointed out in Section 1, clusters made of different-speed processors interconnected by either plain Ethernet or a high-speed switch constitute the typical experimental platforms in most academic or industry research departments.

Because of Theorem 3, we can restrict to interval-based mappings without any loss of generality. However, the complexity of determining the optimal interval-based mapping for *Communication Homogeneous* platforms is still open, this is why we propose several polynomial heuristics to tackle the problem. In the following, we denote by n the number of stages, and by p the number of processors.

4.1 Greedy heuristics

The first heuristics are greedy, i.e. they assign a stage, or an interval of stages, to a given processor, and this choice is independent of further choices. In these heuristics, we systematically assign $L = \lceil \mathsf{n}/\mathsf{p} \rceil$ consecutive stages per processor (except for the last processor if n is not divisible by p). In other words, the heuristics always assign the same set of intervals, the difference between them lies in the allocations that they enforce. In fact, there is a single exception to this rule: for one of the random heuristics, the size of the interval is randomly determined.

The number of processors which will be used in the final solution is thus $\lceil \mathsf{n}/L \rceil$. Note that if $\mathsf{n} \leq \mathsf{p}$, then $L = 1$ and these heuristics perform a One-to-one Mapping algorithm.

H1a-GR: random – For each interval that needs to be assigned, we randomly choose a free processor which will handle it. As soon as a processor is handling an interval, it is not free any more and cannot be used to process another interval.

H1b-GRIL: random interval length – This variant of the greedy random **H1a-GR** is working similarly, but it further chooses the size of the interval to be assigned randomly. The distribution used is homogeneous, with an average length of L and a length between 1 and $2L - 1$. Notice that this heuristic is identical to H1a when $L = 1$.

H2-GSW: biggest $\sum w$ – In this heuristic, the choice is more meaningful since we select the interval with the most demanding computing requirement, and place it on the fastest processor. Intervals are sorted by decreasing values of $\sum_{i \in Interval} w_i$, processors are sorted by decreasing speed s_u, and the matching is achieved.

H3-GSD: biggest $\delta_{in} + \delta_{out}$ – This heuristic is quite similar to **H2-GSW** except that the intervals are sorted according to their communication requirements, $\delta_{in} + \delta_{out}$, where in is the first stage of the interval, and $out - 1$ the last one.

H4-GP: biggest period on fastest processor – This heuristic is balancing the computation and communication requirements of each interval: the processors are sorted by decreasing speed s_u and, for the current processor u, we choose the interval with the biggest period $(\delta_{in} + \delta_{out})/\mathsf{b} + \sum_{i \in Interval} w_i/s_u$. Then we keep going with the remaining processors and intervals.

4.2 Sophisticated heuristics

This second set of heuristics presents more elaborated heuristics, trying to make clever choices.

H5-BS121: binary search for One-to-one Mapping – This heuristic implements the optimal algorithm for the One-to-one Mapping case, described in Section 3.1. When $p < n$, we cut the application in intervals, just as we did for the greedy heuristics. There are now p intervals, for which we use a One-to-one Mapping. Because the partition in intervals is the same for the previous four greedy heuristics and for **H5-BS121**, this latter heuristic should always perform better.

H6-SPL: splitting intervals – This heuristic sorts the processors by decreasing speed, and starts by assigning all the stages to the first processor in the list. This processor becomes used. Then, at each step, we select the used processor j with the largest period and we try to split its interval of stages, giving some stages to the next fastest processor j' in the list (not yet used). This can be done by splitting the interval at any place, and either placing the first part of the interval on j and the remainder on j', or the other way round. The solution which minimizes $max(period(j), period(j'))$ is chosen if it is better than the original solution. Splitting is performed as long as we improve the period of the solution.

H7a-BSL and H7b-BSC: binary search (longest/closest) – The last two heuristics perform a binary search on the period of the solution. For a given period P, we study if there is a feasible solution, starting with the first stage ($s = 1$) and constructing intervals (s, s') to fit on processors. For each processor u, and each $s' \geq s$ we compute the period (s, s', u) of stages $s..s'$ running on processor u and check whether it is smaller than P (then it is a possible assignment). The first variant **H7a-BSL** choose the longest possible interval (maximizing s') fitting on a processor for a given period, and in case of equality, the interval and processor with the closest period to the solution period. The second variant **H7b-BSC** does not take into account the length of the interval, but only finds out the closest period.

The code for all these heuristics can be found on the Web at:

http://graal.ens-lyon.fr/~abenoit/code/skeleton-heuristics.c

5 Experiments

Several experiments have been conducted in order to assess the performance of the heuristics described in Section 4. We have generated a set of random applications with $\mathsf{n} = 1$ to 50 stages, and two sets of random platforms, one set with $\mathsf{p} = 10$ processors and the other with $\mathsf{p} = 100$ processors. The first case corresponds to a situation in which several stages are likely to be mapped on the same processor because there are much fewer processors than stages. However, in the second case, we expect the mapping to be a ONE-TO-ONE MAPPING, except when communications are really costly.

The heuristics have been designed for *Communication Homogeneous* platforms, so we restrict to such platforms in these experiments. In all the experiments, we fix $\mathsf{b} = 10$ for the link bandwidths. Moreover, the speed of each processor is randomly chosen as an integer between 1 and 20. We keep the latter range of variation throughout the experiments, while we vary the range of the application parameters from one set of experiments to the other. Indeed, although there are four categories of parameters to play with, i.e. the values of δ, w, s and b, we can see from equation (2) that only the relative ratios $\frac{\delta}{\mathsf{b}}$ and $\frac{\mathsf{w}}{\mathsf{s}}$ have an impact on the performance.

Each experimental value reported in the following has been calculated as an average over 100 randomly chosen application/platforms pairs. For each of these pairs, we report the performance of the 9 heuristics described in Section 4.

We report four main sets of experiments. For each of them, we vary some key application/platform parameter to assess the impact of this parameter on the performance of the heuristics. The first two experiments deal with applications where communications and computations have the same order of magnitude, and we study the impact of the degree of heterogeneity of the communications, i.e. of the variation range of the δ parameter: in the first experiment the communication are homogeneous, while in the second one δ varies between 1 to 100. The last two experiments deal with imbalanced applications: the third experiment assumes large computations (large value of the w to δ ratio), and the fourth one reports results for small computations (small value of the w to δ ratio).

5.1 Experiment 1: balanced communication/computation, and homogeneous communications

In the first set of experiments, the application communications are homogeneous, we fix $\delta_i = 10$ for $i = 0..\mathsf{n}$. The computation time required by each stage is randomly chosen between 1 and 20. Thus, the communications and computations are balanced within the application.

We notice that the sophisticated heuristics perform much better than the greedy ones when $\mathsf{p} = 10$: they correspond to the lowest three curves. Heuristic H7b is the best for this configuration, before H7a and H6. H2 and H4 give the same result as H5 (binary search returning the optimal algorithm for a ONE-TO-ONE MAPPING), since the communications are homogeneous, and H3 is less good because it tries to make a choice among communications which are all identical. Finally, we see that all our heuristics are largely outperforming the random ones.

When $\mathsf{p} = 100$, the optimal ONE-TO-ONE MAPPING algorithm returns the best solution (H5), obtained similarly by H2 and H4. The sophisticated heuristics are less accurate (but H7 gives a result very close to H5), and the random ones are not even represented on the plot: they always return a maximum period greater than 3.5, while we decided to zoom on the interesting part of the plots.

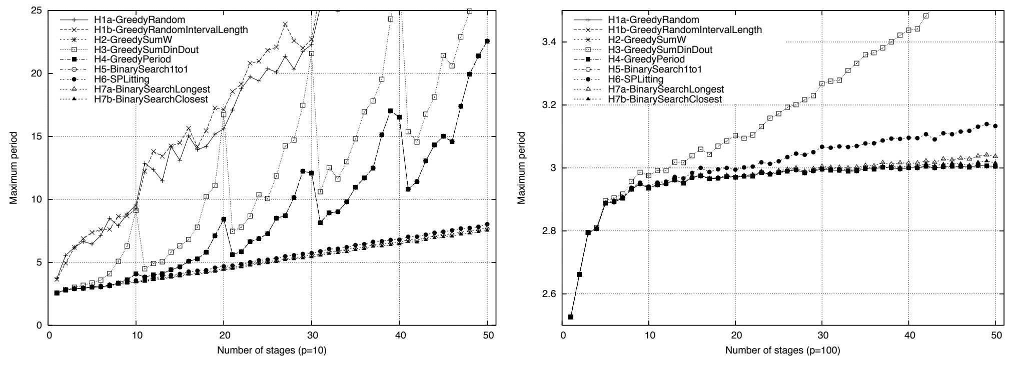


Figure 6: Experiment 1: homogeneous communications.

5.2 Experiment 2: balanced communication/computation, and heterogeneous communications

In this second set of experiments, the application communications are heterogeneous, chosen randomly between 1 and 100. Similarly to Experiment 1, the computation time required by each stage is randomly chosen between 1 and 20. Thus, the communications and computations are still relatively balanced within the application.

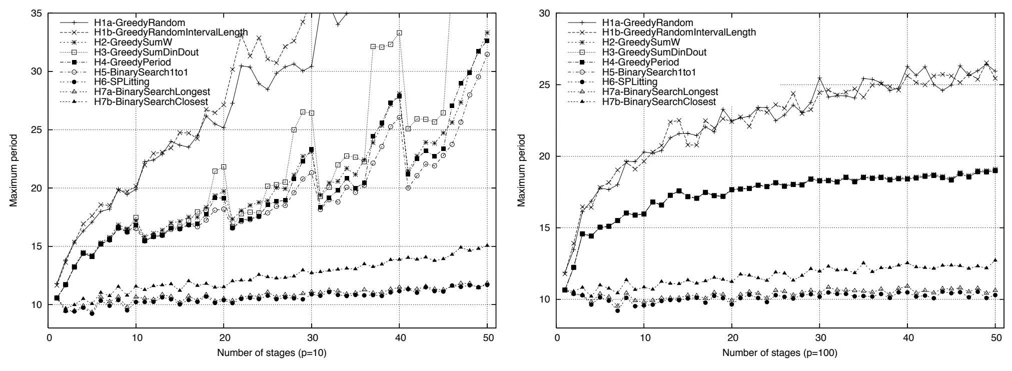


Figure 7: Experiment 2: heterogeneous communications.

In the case with $\mathsf{p} = 10$, the sophisticated heuristics are the best ones, and H6 outperforms the binary search heuristics H7a and H7b. Also, the first binary search H7a is better than H7b, while it was the other way round with homogeneous computations. The splitting heuristic H6 is definitively better in this heterogeneous case. We also notice that H5 is better than all the greedy heuristics, which is due to the fact that it implements the optimal algorithm for a One-to-one Mapping (or fixed-length Interval Mapping).

For $\mathsf{p} = 100$, the sophisticated heuristics behave in the same way, but we notice that all the greedy heuristics find the optimal One-to-one Mapping solution, similarly to H5. In both cases, all our heuristics are much more efficient than the random ones.

5.3 Experiment 3: large computations

In this experiment, the applications are much more demanding on computations than on communications, making communications negligible compared to the computation requirements. We choose the communication time between 1 and 20, while the computation time of each application is chosen between 10 and 1000.

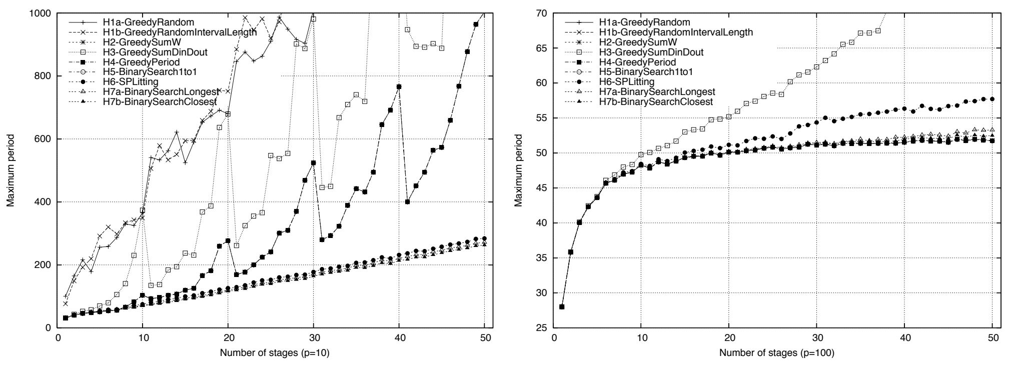


Figure 8: Experiment 3: large computations.

In this case, both plots are showing the same behavior as in Experiment 1. In fact, even though communications are not the same for each application, their relative importance is negligible in front of the computations, and thus the system behaves as if communications were homogeneous. Thus, the One-to-one Mapping heuristics are clearly the best with $p = 100$, while H7b returns the smallest maximum period for $p = 10$.

5.4 Experiment 4: small computations

The last experiment is the opposite to Experiment 3 since the computations are now negligible compared to the communications. The communication time is still chosen between 1 and 20, but the computation time is now chosen between 0.01 and 10.

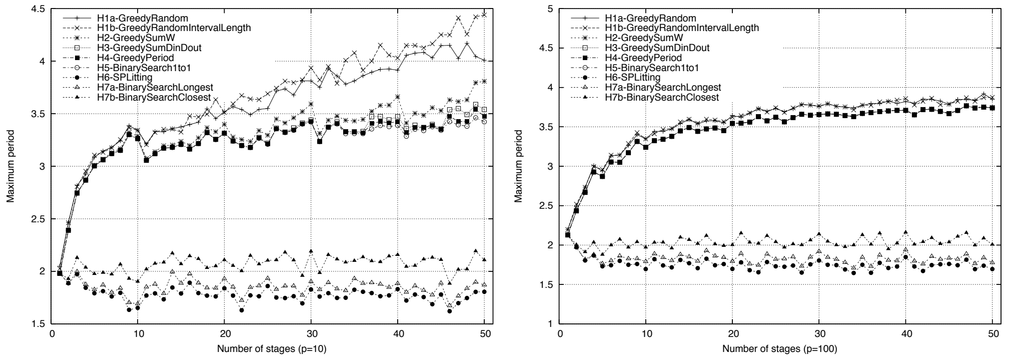


Figure 9: Experiment 4: small computations.

In this case, since communications are important, the sophisticated heuristics perform better. This is because these heuristics often choose different length intervals in order to reduce the communication cost, while all the greedy and random heuristics are always making the same choice of intervals.

Similarly to the heterogeneous case (Experiment 2), H6 is the best heuristic, H7a and H7b are just a little bit ahead. Also, for One-to-one Mapping situations, the greedy heuristics returns the same result as H5.

However, in this situation, H3 is better than H2. This never happened before because computations were always the most relevant parameter to optimize. Also, we notice that the random heuristics are quite close to the greedy ones, because they choose the same intervals and so they pay the same communication cost. The clever choices performed by the greedy heuristics allow to save a little on the computation side.

5.5 Summary

To summarize our experimental results, we first point out that our heuristics are always much more efficient than random mappings. Moreover, we identify three heuristics which may each turn out to be the most efficient, depending upon the application and platform characteristics.

When there are more processors than pipeline stages, we can expect that a One-to-one Mapping might be a good choice. Actually, if the computations are the costly part of the application on a given platform, and communications are of lesser importance, then the greedy One-to-one Mapping heuristics return the best result, which we expect to be close to the optimal. This is also the case when communications are fully homogeneous between stages, because splitting between more stages does not add a large overhead. The optimal binary search H5 for One-to-one Mapping should be used in such cases.

The same balance between communications and computations leads to a different result when there are fewer processors, because it is then necessary to share the computation load between processors, and the decision where to split intervals can be really relevant. All the greedy heuristics are using intervals of fixed length and cannot make any clever choices. In such cases, H7b (Binary Search Closest) is the most efficient heuristic.

Moreover, as soon as communications are costly or with a high degree of heterogeneity, the greedy heuristics do not return satisfying results, because they may cut intervals on costly links between stages. Thus the more sophisticated Interval Mapping heuristics performs better. In such cases, the splitting heuristic H6 is the best choice.

6 Assessing the absolute performance of the heuristics

We introduce an integer linear program which allows to compute the optimal mapping for a given platform and application. We compare the results of the heuristics to the solution of this linear program solution, when available. The large number of integer variables in the linear program makes it impossible to solve large problem instances.

6.1 Linear program formulation

We present here an integer linear program to compute the optimal interval-based mapping on *Fully Heterogeneous* platforms. We assume n stages and p processors, plus two fictitious extra stages $\mathcal{S}_0$ and $\mathcal{S}_{\mathsf{n}+1}$ respectively assigned to P_{in} and P_{out}. First we need to define a few variables:

- For $k \in [0..\mathsf{n}+1]$ and $u \in [1..\mathsf{p}] \cup \{\mathsf{in}, \mathsf{out}\}$, $x_{k,u}$ is a boolean variable equal to 1 if stage $\mathcal{S}_k$ is assigned to processor P_u; we let $x_{0,\mathsf{in}} = x_{\mathsf{n}+1,\mathsf{out}} = 1$, and $x_{k,\mathsf{in}} = x_{k,\mathsf{out}} = 0$ for $1 \le k \le \mathsf{n}$.

- For $k \in [0..\mathsf{n}]$, $u, v \in [1..\mathsf{p}] \cup \{\mathsf{in}, \mathsf{out}\}$ with $u \neq v$, $z_{k,u,v}$ is a boolean variable equal to 1 if stage $\mathcal{S}_k$ is assigned to P_u and stage S_{k+1} is assigned to P_v: hence $\mathsf{link}_{u,v} : P_u \to P_v$ is used for the communication between these two stages. If $k \neq 0$ then $z_{k,\mathsf{in},v} = 0$ for all $v \neq \mathsf{in}$ and if $k \neq \mathsf{n}$ then $z_{k,u,\mathsf{out}} = 0$ for all $u \neq \mathsf{out}$.

- For $k \in [0..\mathsf{n}]$ and $u \in [1..\mathsf{p}] \cup \{\mathsf{in}, \mathsf{out}\}$, $y_{k,u}$ is a boolean variable equal to 1 if stages $\mathcal{S}_k$ and S_{k+1} are both assigned to P_u; we let $y_{k,\mathsf{in}} = y_{k,\mathsf{out}} = 0$ for all k, and $y_{0,u} = y_{\mathsf{n},u} = 0$ for all u.
- For $u \in [1..\mathsf{p}]$, $\mathsf{first}(u)$ is an integer variable which denotes the first stage assigned to P_u; similarly, $\mathsf{last}(u)$ denotes the last stage assigned to P_u. Thus P_u is assigned the interval $[\mathsf{first}(u), \mathsf{last}(u)]$. Of course $1 \leq \mathsf{first}(u) \leq \mathsf{last}(u) \leq \mathsf{n}$.
- T_{period} is the period of the pipeline.

We list below the constraints that need to be enforced. For simplicity, we write $\sum_u$ instead of $\sum_{u \in [1..\mathsf{p}] \cup \{\mathsf{in},\mathsf{out}\}}$ when summing over all processors. First there are constraints for processor and link usage:

- Every stage is assigned a processor: $\forall k \in [0..\mathsf{n}+1], \qquad \sum_u x_{k,u} = 1$.
- Every communication either is assigned a link or collapses because both stages are assigned to the same processor:

$$\forall k \in [0..\mathsf{n}], \qquad \sum_{u \neq v} z_{k,u,v} + \sum_u y_{k,u} = 1$$

- If stage $\mathcal{S}_k$ is assigned to P_u and stage $\mathcal{S}_{k+1}$ to P_v, then $\mathsf{link}_{u,v} : P_u \to P_v$ is used for this communication:

$$\forall k \in [0..\mathsf{n}], \forall u, v \in [1..\mathsf{p}] \cup \{\mathsf{in}, \mathsf{out}\}, u \neq v, \qquad x_{k,u} + x_{k+1,v} \leq 1 + z_{k,u,v}$$

- If both stages $\mathcal{S}_k$ and $\mathcal{S}_{k+1}$ are assigned to P_u, then $y_{k,u} = 1$:

$$\forall k \in [0..\mathsf{n}], \forall u \in [1..\mathsf{p}] \cup \{\mathsf{in}, \mathsf{out}\}, \qquad x_{k,u} + x_{k+1,u} \leq 1 + y_{k,u}$$

- If stage $\mathcal{S}_k$ is assigned to P_u, then necessarily $\mathsf{first}_u \leq k \leq \mathsf{last}_u$. We write this constraint as:

$$\forall k \in [1..\mathsf{n}], \forall u \in [1..\mathsf{p}], \qquad \mathsf{first}_u \leq k.x_{k,u} + \mathsf{n}.(1 - x_{k,u})$$

$$\forall k \in [1..\mathsf{n}], \forall u \in [1..\mathsf{p}], \qquad \mathsf{last}_u \geq k.x_{k,u}$$

- If stage $\mathcal{S}_k$ is assigned to P_u and stage $\mathcal{S}_{k+1}$ is assigned to $P_v \neq P_u$ (*i.e.* $z_{k,u,v} = 1$) then necessarily $\mathsf{last}_u \leq k$ and $\mathsf{first}_v \geq k+1$ since we consider intervals. We write this constraint as:

$$\forall k \in [1..\mathsf{n}-1], \forall u, v \in [1..\mathsf{p}], u \neq v, \qquad \mathsf{last}_u \leq k.z_{k,u,v} + \mathsf{n}.(1 - z_{k,u,v})$$

$$\forall k \in [1..\mathsf{n}-1], \forall u, v \in [1..\mathsf{p}], u \neq v, \qquad \mathsf{first}_v \geq (k+1).z_{k,u,v}$$

- There remains to express the period of each processor and to constrain it by T_{period}:

$$\forall u \in [1..\mathsf{p}], \qquad \sum_{k=1}^{n} \left\{ \left(\sum_{t \neq u} \frac{\delta_{k-1}}{\mathsf{b}_{t,u}} z_{k-1,t,u} \right) + \frac{\mathsf{w}_k}{\mathsf{s}_u} x_{k,u} + \left(\sum_{v \neq u} \frac{\delta_k}{\mathsf{b}_{u,v}} z_{k,u,v} \right) \right\} \leq T_{\mathsf{period}}$$

Finally, the objective function is to minimize the period T_{period}.

We have $O(\mathsf{np}^2)$ variables, and as many constraints. All variables are boolean or integer, except the period, which is rational. We present some experiments comparing the heuristics to the linear program solution, when the number of variables is small enough to allow for a resolution in reasonable time.

6.2 LP Experiments

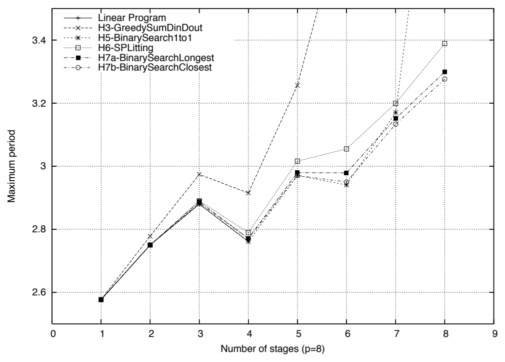
We compared the performance of the heuristics with the optimal solution returned by the linear program. Because of the large number of integer variables, experiments have been constrained to small platforms and applications. Also, in the following, we use only 10 instances of each application; we used 100 instances in the previous experiments but the LP programs can be quite long to solve. The results now represent an average over these 10 instances of the problem.

6.2.1 LP limitation

The largest experiment has been conducted with $p = 8$ processors. We tried to solve the linear program with up to $n = 8$ stages. The parameters are chosen randomly as in Experiment 1, with homogeneous communications and balanced communication/computation ratios (Section 5.1).

However, from $n = 4$ stages, the LP program requires quite a long time to be solved: it took up to 14 hours of computation time to solve a single instance of the problem. Because it was already very long for $n = 4$, we did not experiment with higher values of n. In practice, the use of the linear program in such cases is very limited because of the extremely long time required for the resolution.

We plotted (Figure 10) the result obtained with the LP program and some of the relevant heuristics. We see that the best heuristics, in this case the optimal One-to-one Mapping obtained with H5 and the binary search H7b, are very close to the result of the LP. The table displays the exact results, and we can see that there is only a tiny difference between the optimal result of the LP and the result returned by our heuristics. This can be explained by the precision of the binary search which was set to 0.0001. A more accurate result could have been obtained with the heuristics by increasing the precision.



n	LP	H5-BS121	H7b-BSC
1	2.576857	2.576882	2.576882
2	2.749913	2.749934	2.749934
3	2.879871	2.879900	2.883072
4	2.760960	2.760981	2.770690

Figure 10: LP limitation.

6.2.2 LP on small platforms

Since we could not perform large experiments with the LP on platforms with many processors, we restricted ourselves to smaller platforms in order to study the absolute performance of the heuristics. The results seemed quite encouraging on the previous experiment with up to 4 stages and 8 processors, since we almost always found the optimal mapping.

This experiment has been conducted on a platform with $p = 4$ processors, and applications with up to $n = 10$ stages. We plot the results for the homogeneous case (similar to Experiment 1, Section 5.1) and for the heterogeneous case (similar to Experiment 2, Section 5.2), which are the two most relevant cases. We restrict the plots (Figure 11) to some of the best heuristics identified in the previous experiments.

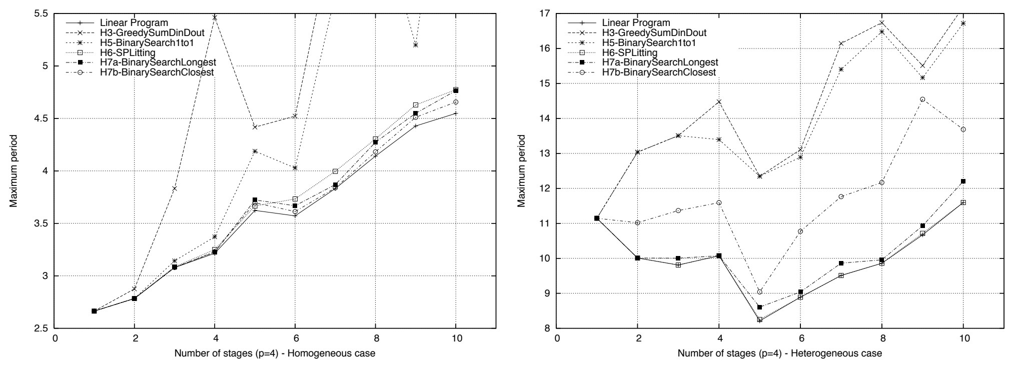


Figure 11: LP on small platforms.

In the homogeneous case, we point out that the best heuristic H7b is very close to the optimal result returned by the LP program. The maximum difference is for $\mathsf{n} = 10$, with a difference of 0.11, which represents an error of less than 3%. This result is quite promising, even though we cannot conduct experiments for larger platforms because of the LP limitation.

The results are even better in the heterogeneous case, where the splitting heuristic H6 is almost always returning the optimal mapping, with a largest error less than 0.05%.

Altogether, despite its intrinsic limitation to small platforms, the LP formulation enables us to assess the absolute performance of our heuristics. The results are very satisfying since the heuristics are always very close to the optimal result. In addition, their execution time remains small for large application/platform pairs, while the LP is not usable for platforms with more than 8 processors, due to the large number of integer variables.

7 Related work

We classify several related papers along the following four main lines:

Scheduling task graphs on heterogeneous platforms– Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors, see [19, 26] among others. Unfortunately, all these heuristics assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications. Recent papers [15, 16, 23] suggest to take communication contention into account. Among these extensions, scheduling heuristics under the one-port model [17, 18] are considered in [3]: just as in this paper, each processor can communicate with at most one other processor at a given time-step.

Mapping pipelined computations onto special-purpose architectures– There are two lines of work related to mapping pipeline computations onto special architectures: the first deal with special-purpose architectures and FPGA arrays. A representative example is the work by Fabiani and Lavenier [14]. They study the placement of linear computations onto reconfigurable arrays. The other line of work is related to the design of fault-tolerant or power-aware mapppings for embedded systems. Representative examples are [27, 2].

Mapping pipelined computations onto clusters and grids– The papers quoted in this paragraph are the most closely related to our work. They consider the problem of mapping communicating tasks onto heterogeneous platforms, but the applicative framework is different. In [25], Taura and Chien consider applications composed of several copies of the same task graph, expressed as a DAG (directed acyclic graph). These copies are to be executed in pipeline fashion. Taura and Chien also restrict to mapping all instances of a given task type (which corresponds to a stage in our framework) onto the same processor. In other words, they consider the same problem as ours, except that the linear pipeline is replaced by a general DAG. Their problem is shown NP-complete, and they provide an iterative heuristic to determine a good mapping. At each step, the heuristic refines the current clustering of the DAG. Beaumont et al [4] consider the same problem as Taura and Chien, i.e. with a general DAG, but they allow a given task type to be mapped onto several processors, each executing a fraction of the total number of tasks. The problem remains NP-complete, but becomes polynomial for special classes of DAGs, such as series-parallel graphs. For such graphs, it is possible to determine the optimal mapping owing to an approach based upon a linear programming formulation. The drawback with the approach of [4] is that the optimal throughput can only be achieved through very long periods, so that the simplicity and regularity of the schedule are lost, while the latency is severely increased.

Another important series of papers comes from the DataCutter project [13]. One goal of this project is to schedule multiple data analysis operations onto clusters and grids, decide where to place and/or replicate various components [7, 8, 24]. A typical application is a chain of consecutive filtering operations, to be executed on a very large data set. So the task graphs targeted by DataCutter are more general than our linear pipeline framework, but still much more regular than arbitrary DAGs, which allows them to design several heuristics to efficiently solve the previous placement and replication optimization problems.

Mapping skeletons onto clusters and grids– Benoit et al [5, 6] have explored the use of stochastic process algebra to decide for the best mapping of pipeline and deal skeletal applications. However, in this work, no formal method has been developed in order to find a mapping; instead, the authors performed a relative comparison between several (given) mappings. They provided a performance model for each mapping, based on process algebra, and they determined which one was the best according to the period, estimated through performance results of the model.

8 Conclusion

In this paper, we have thoroughly studied a difficult mapping problem onto heterogeneous platforms. We restricted ourselves to the class of applications which have a pipeline structure, and studied the complexity of the problem for different variants of mapping strategies and different types of platforms. To the best of our knowledge, it is the first time that pipeline mapping is studied from a theoretical perspective, while it is quite a standard and widely used pattern in many real-life applications.

For a One-to-one Mapping, we provided a polynomial algorithm which finds the optimal solution. However, restricting each processor to execute a single stage may not be the best choice, mainly when communications are very costly, hence take very long time between two consecutive stages. In this case, we would rather place the two consecutive stages onto the same processor, which is allowed in the Interval Mapping variant of the problem. For this latter case, we provided several efficient polynomial heuristics for *Communication Homogeneous* platforms. Finally, we point out that the absolute performance of the heuristics is quite good, since their result is close to the optimal solution returned by an integer linear program.

There remains much work to extend the results of this paper, in several directions. On the theoretical side, the complexity for the Interval Mapping variant on *Communication Homogeneous* platforms is still an open problem, even though we guess it might be NP-complete. On the practical side, we still need to design heuristics for *Fully Heterogeneous* platforms and assess their performance, which is a challenging problem. Finally, in the longer term, we plan to perform real experiments on heterogeneous platforms, using an already-implemented skeleton library, in order to compare the effective performance of the application for a given mapping (obtained with our heuristics) against the theoretical performance of this mapping.

A natural extension of this work would be to consider other widely used skeletons. For example, when there is a bottleneck in the pipeline operation due to a stage which is both computationally-demanding and not constrained by internal dependencies, we can nest another skeleton in place of the stage. For instance a farm or deal skeleton would allow to split the workload of the initial stage among several processors. Using such deal skeletons may be either the programmer's decision (explicit nesting in the application code) or the result of the mapping procedure. Extending our mapping strategies to automatically identify opportunities for deal skeletons, and implement these, is a difficult but very interesting perspective.

References

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

[2] 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.

[3] O. Beaumont, V. Boudet, and Y. Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *HCW'2002, the 11th Heterogeneous Computing Workshop.* IEEE Computer Society Press, 2002.

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In *HeteroPar'2004: International Conference on Heterogeneous Computing, jointly published with ISPDC'2004: International Symposium on Parallel and Distributed Computing*, pages 296–302. IEEE Computer Society Press, 2004.

[5] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of pipeline-structured parallel programs with skeletons and process algebra. *Scalable Computing: Practice and Experience*, 6(4):1–16, December 2005.

[6] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Scheduling skeleton-based grid applications using PEPA and NWS. *The Computer Journal, Special issue on Grid Performability Modelling and Measurement*, 48(3):369–378, 2005.

[7] M. Beynon, A. Sussman, U. Catalyurek, T. Kurc, and J. Saltz. Performance optimization for data intensive grid applications. In *PProceedings of the Third Annual International Workshop on Active Middleware Services (AMS'01).* IEEE Computer Society Press, 2001.

[8] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002.

[9] 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.

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

[11] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[12] P. Crescenzi and V. Kann. A compendium of NP optimization problems. World Wide Web document, URL: http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html.

[13] DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm.

[14] E. Fabiani and D. Lavenier. Placement of linear arrays. In *FPL 2000, 10th International Conference on Field Programmable Logic and Applications*. IEEE Computer Society Press, 2000.

[15] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[16] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[17] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, 1989.

[18] D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM J. Computing*, 21:111–139, 1992.

[19] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[20] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 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] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[23] O. Sinnen and L. Sousa. Comparison of contention-aware list scheduling heuristics for cluster computing. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 382–387. IEEE Computer Society Press, 2001.

[24] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the grid. In *2002 ACM/IEEE Supercomputing Conference*. ACM Press, 2002.

[25] K. Taura and A. A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 2000.

[26] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1999.

[27] D. Zhu, R. Melhem, and B. Childers. Power-aware scheduling for multi-processor real-time systems. *IEEE Trans. Parallel Distributed Systems*, 14(7):686–700, 2003.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399