Complexity results for throughput and latency optimization of replicated and data-parallel workflows.

Anne Benoit, Yves Robert

Complexity results for throughput and latency optimization of replicated and data-parallel workflows

Anne Benoit and Yves Robert

LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
UMR 5668 - CNRS - ENS Lyon - UCB Lyon - INRIA
{Anne.Benoit|Yves.Robert}@ens-lyon.fr

March 2007

LIP Research Report RR-2007-12

Abstract

Mapping applications onto parallel platforms is a challenging problem, even for simple application patterns such as pipeline or fork graphs. Several antagonist criteria should be optimized for workflow applications, such as throughput and latency (or a combination). In this paper, we consider a simplified model with no communication cost, and we provide an exhaustive list of complexity results for different problem instances. Pipeline or fork stages can be replicated in order to increase the throughput of the workflow, by sending consecutive data sets onto different processors. In some cases, stages can also be data-parallelized, i.e. the computation of one single data set is shared between several processors. This leads to a decrease of the latency and an increase of the throughput. Some instances of this simple model are shown to be NP-hard, thereby exposing the inherent complexity of the mapping problem. We provide polynomial algorithms for other problem instances. Altogether, we provide solid theoretical foundations for the study of mono-criterion or bi-criteria mapping optimization problems.

Key words: pipeline graphs, fork graphs, scheduling algorithms, throughput maximization, latency minimization, bi-criteria optimization, heterogeneous platforms, complexity results.

1 Introduction

In this paper we deal with scheduling and mapping strategies for simple application workflows. Such workflows operate on a collection of data sets that are executed in a pipeline fashion [26, 25, 30]. Each data set is input to the application graph and traverses it until its processing is complete. The application graph itself is composed of several stages, each corresponding to a given task. The mapping assigns these stages to distinct processor sets, so as to minimize one or several objectives.

Key metrics for a given workflow are the throughput and the latency. The throughput measures the aggregate rate of processing of data, and it is the rate at which data sets can enter the system. Equivalently, the inverse of the throughput, defined as the period, is the time interval required between the beginning of the execution of two consecutive data sets. 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. Minimizing the latency is antagonistic to minimizing the period, and tradeoffs should be found between these criteria. Efficient mappings aim at the minimization of a single criterion, either the period or the latency, but they can also use a bi-criteria approach, such as minimizing the latency under period constraints (or the converse).

Searching for an optimal mapping encompasses various levels of difficulty, some related to the application and others linked to the target platform. An application stage, or even an interval of consecutive stages, may be replicated onto several processors, which will execute successive data sets in a round-robin fashion, thereby reducing the period. Another possibility for data-parallel stages is to share the execution of the same data set among several processors, thereby reducing the latency. Achieving a balanced utilization of all resources over the entire application graph becomes a complicated goal. This is already true on simple, homogeneous platforms composed of identical processors and interconnection links. The situation becomes even worse when dealing with heterogeneous platforms, with different-speed processors and different-capacity links.

In this paper we make some important restrictions on the application graphs under study, as well as on the execution model to deploy the application workflow on the platform. Our goal is to provide a solid theoretical foundation for the study of single criterion or bi-criteria mappings. We aim at assessing the additional complexity induced by replicated and/or data-parallel stages on the application side, and by different-speed processors on the platform side. This implies to deal with simple (but important) application graphs for which efficient mappings can be found in polynomial time using identical processors, and without replicating or data-parallelizing any stage.

To this purpose we restrict to two important application graphs, namely linear pipelines and fork graphs, as illustrated on Figures 1 and 2. Both graphs are ubiquitous in parallel processing and represent archetype application workflows. Pipeline graphs occur in many applications in the domains of image processing, computer vision, query processing, etc, while fork graphs are mandatory to distribute files or databases in master-slave environments. While important, both graphs are simple enough so that the design of optimal mappings is well understood in simple frameworks.

Pipeline graphs are easier to deal with, because there is a single dependence path. For example Subhlok and Vondran [26, 27] have been able to design dynamic programming algorithms for bi-criteria mappings on homogeneous platforms. Also, if we neglect all communication costs, min-

imizing the period amounts to solve the well-known chains-to-chains problem. Given an array of n elements $a_1, a_2, \ldots, a_{\mathsf{n}}$, this problem is to partition the array into p intervals whose element sums are well balanced (technically, the aim is to minimize the largest sum of the elements of any interval). This problem has been extensively studied in the literature (see the pioneering papers [9, 13, 21] and the survey [22]). It amounts to load-balance n computations whose ordering must be preserved (hence the restriction to intervals) onto p identical processors. Does this problem remain polynomial with different-speed processors and the possibility of replicating or data-parallelizing the intervals? The complexity of these important extensions of the chains-to-chains problem is established in this paper.

Fork graphs are more difficult to tackle, because there are more opportunities for parallelism, hence a wider combinatorial space to explore when searching for good mappings. Still, we provide several complexity results for this class of graphs.

The rest of the paper is organized as follows. We start by illustrating the problem on a simple example in Section 2. This section provides an insight of the complexity of the problem, even in simple cases. Then in Section 3, we detail the framework: we present both the general model, and a simplified model without communication costs. The exhaustive complexity results for this simplified model are summarized in Section 4, then we detail the results for pipeline graphs (Section 5), and for fork graphs (Section 6). Related work is surveyed in Section 7. Finally, we conclude in Section 8.

2 Working out an example

Consider an application workflow whose application graph is a n-stage pipeline. The k-th stage requires w_k operations. The workflow is mapped onto a platform with p processors P_1 to P_{p}. The speed of processor P_q is s_q, which means that the time for P_q to process stage $\mathcal{S}_i$ is $\frac{\mathsf{w}_i}{\mathsf{s}_q}$. For the sake of simplicity, assume that no communication cost is paid during the execution. The rule of the game for the mapping is to partition the set of stages into intervals of consecutive stages and to map these intervals onto the processors. The period will be the longest time to process an interval, while the latency will be the sum of the execution times over all intervals.

To illustrate this, consider the following little example with four stages $\mathcal{S}_1$ to $\mathcal{S}_4$. Below each stage $\mathcal{S}_i$ we indicate the number of computations w_i (expressed in flops) that it requires.

$$\underset{14}{\mathcal{S}_1} \rightarrow \underset{4}{\mathcal{S}_2} \rightarrow \underset{2}{\mathcal{S}_3} \rightarrow \underset{4}{\mathcal{S}_4}$$

Assume that we have an homogeneous platform made up of three identical processors, all of unit speed: $\mathsf{p} = 3$ and $\mathsf{s}_1 = \mathsf{s}_2 = \mathsf{s}_3 = 1$. What is the minimum period? Obviously, mapping $\mathcal{S}_1$ to P_1, the other three stages to P_2, and discarding P_3, leads to the best period $T_{\mathsf{period}} = 14$. The solution is not unique, the only constraint is that the processor assigned to $\mathcal{S}_1$ is not assigned any other stage. Note that the latency is always $T_{\mathsf{latency}} = 24$, whatever the mapping, as it is the total computation time needed for a data set to traverse the four stages. This simple observation always holds true with identical processors.

How can we decrease the period? If the computations of a given stage are independent from one data set to another, two consecutive computations (different data sets) for the same stage can be mapped onto distinct processors, thus reducing the period for the processing of this stage. Such a stage can be *replicated*, using the terminology of Subhlok and Vondran [26, 27] and of the DataCutter team [5, 6, 25]. This corresponds to the *dealable* stages of Cole [10]. Note that the

computations of a replicated stage can be fully sequential for a given data set, what matters is that they do not depend from previous results for other data sets, hence the possibility to process different data sets in different locations. If all stages can be replicated, a solution would be to assign the whole processing of a data set to a given processor, and distribute the different data sets among all processors of the platform.

For instance, if the four stages of our example can all be replicated, we can derive the following mapping: processor P_1 would process data sets numbered $1, 4, 7, \ldots$, P_2 those numbered $2, 5, 8, \ldots$ and P_3 those numbered $3, 6, 9, \ldots$. Each data set is processed within 24 time steps, hence a new data set can be input to the platform every $24/3 = 8$ time steps, and $T_{\mathsf{period}} = 8$. Such a mapping extends the previous rule of the game. Instead of mapping an interval of stages onto a single processor, we map it onto a set of processors, and data sets are processed in a round robin fashion by these processors as they enter the interval. With identical processors, we see that the time to process the interval by each processor is simply the weight of the interval (the sum of the computation times of the stages composing the interval) divided by the processor speed, and since each processor only executes a fraction of the data sets, the period is the previous time, further divided by the number of processors assigned to the replication.

With different-speed processors and such a round robin distribution, we would need to retain the longest time needed to process an instance, i.e. the time of the slowest assigned processor, and to divide it by the number of processors. Rather than a round robin distribution of the data sets to processors, we could let each processor execute a number of instances proportional to its speed, thus leading to an optimal throughput. However, as further discussed in Section 3.3, we do not allow such a distribution scheme since it is quite likely to lead to an out-of-order execution of data sets which is not acceptable in the general case.

Formally, replicating the interval $\mathcal{S}_i$ to $\mathcal{S}_j$ onto processors P_{q_1} to P_{q_k} thus requires a time $\frac{\sum_{u=i}^{j} \mathsf{w}_u}{k \times \min_u(\mathsf{s}_{q_u})}$. With k identical processors of speed s, the formula reduces to $\frac{\sum_{u=i}^{j} \mathsf{w}_u}{k \times \mathsf{s}}$.

The previous mapping has replicated a single interval of length four, but it would have been possible to replicate only a subset of stages. For instance we could replicate only S_1 onto P_1 and P_2, and assign the other three stages to P_3, leading to $T_{\mathsf{period}} = \max(\frac{14}{2}, 4+2+4) = 10$. Using a fourth processor P_4 we could further replicate the interval $\mathcal{S}_2$ to $\mathcal{S}_4$, achieving $T_{\mathsf{period}} = \max(7, 5) = 7$. Note that the latency is not modified when replicating stages, because the processing of a given data set remains unchanged. Both of these mappings with replication still achieve $T_{\mathsf{latency}} = 24$.

So, how can we decrease the latency? We need to speed up the processing of each stage separately, which is possible only if the computations within this stage can be parallelized, at least up to some fraction. The processing of such *data-parallel* stages can then be shared by several processors. Contrarily to replicated stages where different instances (for different data sets) are assigned to different resources, each instance of a data-parallel stage is assigned to several processors, which speeds-up the production of each result. There is another major difference: while we can replicate intervals of consecutive stages, we can only data-parallelize single stages (but maybe several of them). To see why, consider two consecutive stages, the first one executing some low-level filtering on its input file (an image), and the second stage implementing various high-level component extraction algorithms. Both stages can be made data-parallel, but the entire image produced by the first stage is needed as input to the second stage. In fact, if both stages could have been data-parallelized simultaneously, the application designer may have chosen to gather them into a single stage, thereby given more opportunities for an efficient parallelization.

As for now, assume fully data-parallel stages (in general, there might be an inherently sequential

part in the stage, which can be modeled using Amdahl's law [1]), see Section 3). Then the time needed to process data-parallel stage $\mathcal{S}_i$ using processors P_{q_1} to P_{q_k} is $\frac{\mathsf{w}_i}{\sum_{u=1}^{k} \mathsf{s}_{q_u}}$. With k identical processors of speed s, the formula reduces to $\frac{\mathsf{w}_i}{k \times \mathsf{s}}$.

Going back to the example, assuming four data-parallel stages, we can reduce the latency down to $T_{\mathsf{latency}} = 17$ by data-parallelizing $\mathcal{S}_1$ onto P_1 and P_2, and assigning the other three stages to P_3. Note that it is not the same mapping as above, because $\mathcal{S}_1$ is data-parallelized instead of being replicated. The period turns out to be the same, namely $T_{\mathsf{period}} = 10$, but the latency is different.

To illustrate the additional complexity induced by heterogeneous platforms, we revisit the example with four different-speed processors: $\mathsf{s}_1 = \mathsf{s}_2 = 2$ and $\mathsf{s}_3 = \mathsf{s}_4 = 1$. Hence P_1 and P_2 are twice faster than P_3 and P_4. Assume that it is possible to replicate or data-parallelize each stage. It becomes tricky to compute the optimal period and latency. If we replicate all stages (i.e. replicate an interval of length 4), we obtain the period $T_{\mathsf{period}} = \frac{24}{4 \times 1} = 6$, which is not optimal because P_1 and P_2 achieve their work in 12 rather than 24 time-steps and then remain idle, because of the round robin data set distribution. A better solution is to data-parallelize $\mathcal{S}_1$ on P_1 and P_2, and to replicate the interval of the remaining three stages onto P_3 and P_4, leading to the period $T_{\mathsf{period}} = \max(\frac{14}{2+2}, \frac{10}{2 \times 1}) = 5$. This is indeed the optimal value for the period, as can be checked by an exhaustive exploration. The first mapping achieves a latency $T_{\mathsf{latency}} = 24$ while the second obtains $T_{\mathsf{latency}} = \frac{14}{4} + 10 = 13.5$. The minimum latency is $T_{\mathsf{latency}} = \frac{14}{5} + 10 = 12.8$, achieved by data-parallelizing $\mathcal{S}_1$ on P_1, P_2 and P_3 and assigning $\mathcal{S}_4$ to P_4. Again, it is not obvious to see that this is the optimal value.

The goal of this little example was to show the combinatorial nature of the target optimization problems. While mono-criterion problems (period or latency) with identical processors seem to remain tractable, it seems harder to solve either bi-criteria problems with identical processors, or mono-criterion problems with different-speed processors. The major contribution of this paper is to derive several new complexity results for these important problems.

3 Framework

We outline in this section the characteristics of the applicative framework, together with possible execution models on the target platform. In particular we introduce the *pipeline* and *fork* application graphs, we define *replicated* tasks and *data-parallel* tasks, and we discuss several scenarios to account for communication costs and computation/communication overlap. We also detail the objective function, chosen either as to minimizing the period or the response time, or as a trade-off between these two antagonistic criteria.

Finally we describe the simplified problems whose complexity will be explored in this paper. Although we use a crude model with neither overhead nor communication cost for the tasks, some problems are already of combinatorial nature on homogeneous platforms (with identical processors), while some others remain of polynomial complexity. We will assess the impact of heterogeneity (using different speed processors) for all these problems.

3.1 Applicative framework

We consider simple application workflows whose graphs are either a pipeline or a fork. Such graphs are representative of a wide class of applications, and constitute the typical building blocks upon which to build and execute more complex workflows.

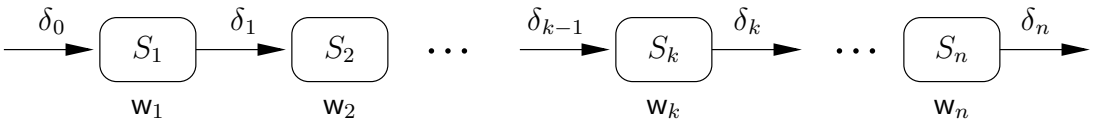

Figure 1: The application pipeline.

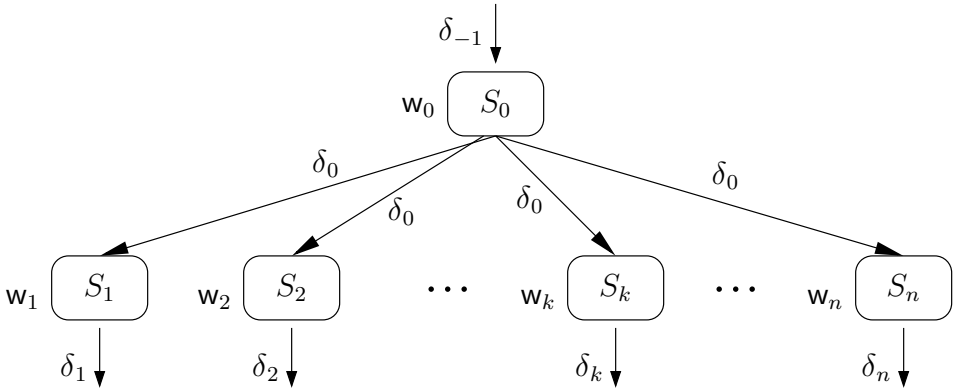

Figure 2: The application fork.

Pipeline graph – A pipeline graph of n stages $\mathcal{S}_k$, $1 \leq k \leq \mathsf{n}$ is illustrated on Figure 1. Consecutive data sets are fed into the pipeline and processed from stage to stage, until they exit the pipeline after the last stage.

Each stage executes a task. More precisely, 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. This operation corresponds to the k-th task and is repeated periodically on each data set. 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.

Fork graph – A fork graph of $\mathsf{n}+1$ stages $\mathcal{S}_k$, $0 \leq k \leq \mathsf{n}$ is illustrated on Figure 2. $\mathcal{S}_0$ is the root stage while $\mathcal{S}_1$ to $\mathcal{S}_\mathsf{n}$ are independent stages that can be executed simultaneously for a given data set. Stage $\mathcal{S}_k$ ($0 \leq k \leq \mathsf{n}$) performs a number of w_k computations on each data set. As for the pipeline graph, consecutive data sets are fed into the fork. Each data set first proceeds through stage $\mathcal{S}_0$, which outputs its results, of size δ_0, to all the other stages. The first stage $\mathcal{S}_0$ receives an input of size δ_{-1} from the outside world, while the other stages $\mathcal{S}_k$, $1 \leq j \leq \mathsf{n}$, may return their results, of size δ_k, to the outside world.

Replicated stage/task – If the computations of the k-th stage are independent from one data set to another, the k-th stage $\mathcal{S}_k$ can be replicated [26, 27, 25]: several consecutive computations are mapped onto distinct processors. Data sets are processed in a round robin fashion by these processors. As already pointed out, the computations of a replicated stage can be fully sequential for a given data set, as long as they do not depend from previous results for other data sets. Replicating a stage or an interval of stages does not change the latency, as each data set follows

the same execution as without replication, but it can decrease the period, as shown in the example of Section 2.

Data-parallel stage/task – If the computations of the k-th stage are data-parallel, their execution can be split among several processors. Each instance of a data-parallel stage is assigned to several processors, which speeds-up the production of each result. Data-parallelizing a task reduces both the latency and the period, at the price of consuming several resources for a given stage.

3.2 Execution models

Target platform – We target a heterogeneous platform 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 $\mathsf{b}_{u,v}$. 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. 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. $P_{\mathsf{in}} = P_{\mathsf{out}}$.

Communication contention – The standard model for DAG scheduling heuristics [31, 18, 29] does a poor job to model physical limits of interconnection networks. The model assumes an unlimited number of simultaneous sends and receives, i.e. a network card of infinite capacity, on each processor. A more realistic model is the *one-port* model [7, 8]. 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 [24]. The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes simpler models [2, 19, 17] where communication time only depends on the sender, not on the receiver. In these models, the communication speed from a processor to all its neighbors is the same.

Another realistic model is the *bounded* multi-port model [14]. In this model, the total communication volume outgoing from a given node is bounded (by the capacity of its network card), but several communications along different links can take place simultaneously (provided that the link bandwidths are not exceeded either). This model would require several communication threads to be deployed. On homogeneous platforms it would be implemented with one-port communications, because if the links have same bandwidths it is better to send messages serially than simultaneously. However, the bounded multi-port is more flexible for heterogeneous platforms.

3.3 Mapping strategies

The general mapping problem consists in assigning application stages to platform processors. When stages are neither replicated nor data-parallel, it is easier to come with a cost model, which we detail before discussing extensions to handle replication and data-parallelism.

Pipeline graphs – For pipeline graphs, it is natural to map intervals of consecutive stages onto processors [26, 27]. Intuitively, assigning several consecutive tasks to the same processor will increase their computational load, but may well dramatically decrease communication requirements. The cost model associated to interval mappings is the following. We search for a partition of $[1..\mathsf{n}]$ into $m \leq \mathsf{p}$ intervals $I_j = [d_j, e_j]$ such that $d_j \leq e_j$ for $1 \leq j \leq m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m-1$ and $e_m = \mathsf{n}$. Interval I_j is mapped onto processor $P_{\mathsf{alloc}(j)}$, and the period is expressed as

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

Here, we assume that $\mathsf{alloc}(0) = \mathsf{in}$ and $\mathsf{alloc}(m+1) = \mathsf{out}$. The latency is obtained by the following expression (data sets traverse all stages, and only interprocessor communications need be paid for):

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

The optimization problem Interval Mapping is to determine the best mapping, over all possible partitions into intervals, and over all processor assignments. The objective can be to minimize either the period, or the latency, or a combination: given a threshold period, what is the minimum latency that can be achieved? and the counterpart: given a threshold latency, what is the minimum period that can be achieved?

Fork graphs – For fork graphs, it is natural to map any partition of the graph onto the processors. Assume such a partition with q sets, where $q \leq \mathsf{p}$. The first set of the partition will contain the root stage $\mathcal{S}_0$ and possibly other independent stages (say $\mathcal{S}_1$ to $\mathcal{S}_k$ without loss of generality), while the other sets will only contain independent stages chosen from $\mathcal{S}_{k+1}$ to $\mathcal{S}_{\mathsf{n}}$. Assuming that the first set (with the root stage) is assigned to P_1, and that the $q-1$ remaining sets are assigned to $P_2, \ldots, P_q$. Defining the period requires to make several hypotheses on the communication model:

- A flexible model would allow P_1 to initiate the communications to the other processors immediately upon completion of the execution of $\mathcal{S}_0$, while a stricter model (say, with a single execution thread) would allow the communications to start only after P_1 has completed all its computations, including those for stages $\mathcal{S}_1$ to $\mathcal{S}_k$.
- Either way, we need to specify the ordering of the communications. This is mandatory for the one-port model, obviously, but this is also true for the bounded multi-port model: a priori, there is no reason for the $q-1$ communications to take place in parallel; the scheduler might decide to send some messages first and others later.

Rather than proceeding with complex formulas, let us define the period and the latency informally:

- The period is the maximum time needed by a processor to receive its data, perform all its computations, and output the result. The period does not depend upon the ordering of the communications but only upon their duration (which is a constant for the one-port model but changes in the bounded multi-port model, depending upon the number of simultaneous messages sent together with the input).

- The latency is the time elapsed between the moment where a data set is input to P_0 and the moment where the last computation concerning this data set is completed. The latency depends whether the model is flexible or strict, and also depends upon the ordering of the communications.

We will use the word *interval* instead of *subset* when partitioning the stages of a fork graph and assigning them to processors. Each processor executes some subset that may include the root stage and/or may include several other (independent) stages, so the assignment is not properly speaking an interval. But for the convenience of the reader, we keep the same terminology as for pipeline graphs.

Replicated stages – Defining the cost model for replicated stages is difficult, in particular when two or more consecutive intervals are replicated onto several (distinct) processor sets.

We start with the replication of a single interval of a pipeline workflow. Assume that the interval $\mathcal{S}_i$ to $\mathcal{S}_j$ is replicated onto processors P_{q_1} to P_{q_k}. What is the time needed to process the interval? Because P_{q_1} to P_{q_k} execute the stages in round-robin fashion, the processing time will not be the same for each data set, and we need to retain the longest time $t_{\max}$ taken by any processor, including communication and computation costs. The period will then be equal to $\frac{t_{\max}}{k}$, because each processor computes every k-th data set: the slowest processor has indeed $t_{\max}$ time-steps available between the arrival of two consecutive inputs.

It is difficult to write formulas for $t_{\max}$ because of communication times. If the stages before $\mathcal{S}_i$ and after $\mathcal{S}_j$ are not replicated, the source of the input and the destination of the output remain the same for each assigned processor P_{q_u} $(1 \leq u \leq k)$, which does simplify the estimation: we would define $t_{\max}$ as the longest time needed for a processor to receive a message from the source, perform its computations and output the message to the destination. But if, for instance, the stage before $\mathcal{S}_i$ is replicated, or belongs to a replicated interval, the source of the input will vary from each processor assigned to the latter stage, and it becomes tricky to analyze the time needed to send and receive messages between any processor pair. We can always take the longest path over all possible pairs, but there may appear synchronization issues that complicate the estimation. Finally, the latency will be the sum of the longest paths throughout the mapping, and again it may be perturbed by hot spots and synchronization issues.

The situation gets even more complicated for fork graphs. Again, we need to consider the longest path to define $t_{\max}$ when replicating a stage interval. While the period does not depend upon whether the model is flexible or strict for $\mathcal{S}_0$, the latency does. Also, the latency dramatically depends upon the communication ordering, and the same problems appear when we want to estimate it precisely.

We conclude this paragraph with a digression on the round-robin rule enforced for mapping replicated stages. With different speed processors, a more efficient strategy to replicate a stage interval would be to let each processor execute a number of instances proportional to its speed. For instance, coming back the example of Section 2 with two fast processors of speed 2 and two slow

ones of speed 1, we could assign twice as many data sets to the fast processors than to the slow ones. Each resource would then be fully utilized, leading to an optimal throughput. However, such a demand-driven assignment is quite likely to lead to an out-of-order execution of data sets in the general case: because of the different pace at which processors are executing the computations, the k-th data set may well exit the replicated stage interval later than the $k+1$-st data set. This would violate the semantics of the application if, say, the next stage is sequential. Because in real-life applications, some stages are sequential and some can be replicated, the round robin rule is always enforced [10, 23].

Data-parallel stages – When introducing data-parallel stages, even the computational model requires some attention. Consider a stage $\mathcal{S}_i$ to be data-parallelized on processors P_{q_1} to P_{q_k}. We may assume that a fraction of the computations is inherently sequential, hence cannot be parallelized, and thus introduce a fixed overhead f_i that would depend only on the stage and not on the assigned processors. Introducing a fixed overhead f_i may be less accurate than precisely estimating the overhead introduced for each assigned processor, but it does account for the startup time induced by system calls. Hence for computations, assuming that each processor executes a share of the work proportional to its speed, we obtain the expression

$$f_i + \frac{\mathsf{w}_i}{\sum_{u=1}^{k} \mathsf{s}_{q_u}}.$$

Next there remains to model communications. First, we have to model intra-stage communications. For example we can envision that a given processor, say P_{q_1}, acts as the master and delivers some internal data to the remaining processors P_{q_2} to P_{q_k}, which in turn will return their partial results to P_{q_1}. This scenario would call for a more sophisticated distribution of the work than a simple proportional sharing, because some fast computing processor P_{q_j} may well have a low bandwidth link with P_{q_1}. In addition, inter-stage communications, i.e. input and output data, induce the same difficulties as for replicated stages, as they originate from and exit to various sources. Finally, as explained in Section 2, we do not allow to data-parallelize stage intervals for pipeline, i.e. we restrict to data-parallelizing single stages. The situation is slightly different for the fork, i.e. we can data-parallelize a set of independent stages (any stages except $\mathcal{S}_0$) on the same set of processors, since they have no dependency relation. However, $\mathcal{S}_0$ cannot be data-parallelized together with other independent stages, since the dependence relation would lead to the same problems as the ones encountered for the pipeline. In both cases, the next difficulty is to chain two dependent data-parallel stages on two distinct processor sets, which calls for a precise model of redistribution costs.

Altogether, we see that it is very difficult to come with a satisfactory model for communications, and that replicated and data-parallel stages dramatically complicate the story. Still, we point out that given a particular application and a target platform, it would be worthwhile to instantiate all the formulas given in this section, as they are very likely to lead to a precise estimation of computation and communication costs. We are not convinced that fully general models involving arbitrary computation and communication cost functions, as suggested in [26, 27], can be instantiated for homogeneous platforms, and we are pretty sure that such models would fail for heterogeneous clusters.

In contrast, we sketch below a very simplified model, where all communication costs and overheads are neglected. We agree that such a model may be realistic only for large-grain applications.

In fact, our objective is to assess the inherent difficulty of the period and/or latency optimization problems, and we believe that the complexity results established in this paper will provide a sound theoretical basis for more experimental approaches.

3.4 Simplified model

In this section, we advocate a simplified model as the base model to lay solid theoretical foundations. We deal either with n-stage pipeline graphs (stages numbered from S_1 to $\mathcal{S}_\mathsf{n}$) or with $(\mathsf{n}+1)$-stage fork graphs (same numbering, plus $\mathcal{S}_0$ the root stage). There are p different speed processors P_1 to P_p. We neglect all communication costs and overheads. The cost to execute stage $\mathcal{S}_i$ on processor P_u alone is $\frac{\mathsf{w}_i}{\mathsf{s}_u}$.

We assume that all stages are data-parallel and can be replicated. Computation costs are as follows:

- The cost to data-parallelize the stage interval $\mathcal{S}_i$ to $\mathcal{S}_j$ ($i = j$ for a pipeline graph, and $0 < i \leq j$ or $i = j = 0$ for a fork graph) on the set of k processors $P_{q_1}, \ldots, P_{q_k}$ is

$$\frac{\sum_{\ell=i}^{j} \mathsf{w}_\ell}{\sum_{u=1}^{k} \mathsf{s}_{q_u}}.$$

  This cost is both equal to the period of the assigned processors and to the delay to traverse the interval.

- The cost to replicate the stage interval $\mathcal{S}_i$ to $\mathcal{S}_j$ on the set of k processors $P_{q_1}, \ldots, P_{q_k}$ is

$$\frac{\sum_{\ell=i}^{j} \mathsf{w}_\ell}{k \times \min_{1 \leq u \leq k} \mathsf{s}_{q_u}}.$$

  This cost is equal to the period of the assigned processors but the delay to traverse the interval is the time needed by the slowest processor, i.e. $t_{\max} = \frac{\sum_{\ell=i}^{j} \mathsf{w}_\ell}{\min_{1 \leq u \leq k} \mathsf{s}_{q_u}}$.

Note that we do not allow stage intervals of length at least 2 to be data-parallelized in a pipeline: such intervals can only be replicated (or executed on a single processor, which is a particular case of replication). However, we do allow several consecutive (or non consecutive) stages to be data-parallelized using distinct processor sets.

For pipeline and fork graphs, the period is defined as the maximum period of a given processor, and can be readily computed using the previous formulas. Computing the latency is also easy for pipelines, because it is the sum of the delays incurred when traversing all stages; depending upon whether the current interval is replicated or data-parallel, we use the formula for the delay given above.

Computing the latency for fork graphs requires some additional notations. Assume a partition of the $\mathsf{n}+1$ stages into q sets $\mathcal{I}_r$, where $1 \leq r \leq q \leq \mathsf{p}$. Without loss of generality, the first set $\mathcal{I}_1$ contains the root stage and is assigned to the set of k processors $P_{q_1}, \ldots, P_{q_k}$. Let $t_{\max}(r)$ be the delay of the r-th set, $1 \leq r \leq q$, computed as if it was a stage interval of a pipeline graph, i.e. using the previous formulas to account for data-parallelism or replication. We use a flexible model where the computations of set $\mathcal{I}_r$, $r \geq 2$, can start as soon as the computation of stage $\mathcal{S}_0$, from which it has an input dependence, is completed. In other words, there is no need to wait for the completion

of all tasks in $\mathcal{I}_1$ to initiate the other sets $\mathcal{I}_r$, we only wait for $\mathcal{S}_0$ to terminate. We then derive the latency of the mapping as

$$T_{\mathsf{latency}} = \max\left(t_{\max}(1), \frac{\mathsf{w}_0}{\mathsf{s}_0} + \max_{2 \leq r \leq q} t_{\max}(r)\right).$$

Here, s_0 is the speed at which $\mathcal{S}_0$ is processed, hence $\mathsf{s}_0 = \sum_{u=1}^{k} \mathsf{s}_{q_u}$ if $\mathcal{I}_1$ is data-parallelized, and $\mathsf{s}_0 = \min_{1 \leq u \leq k} \mathsf{s}_{q_u}$ if $\mathcal{I}_1$ is replicated.

We are ready to define the optimization problems formally. Given:

- an application graph (n-stage pipeline or $(\mathsf{n}+1)$-stage fork),
- a target platform (*Homogeneous* with p identical processor or *Heterogeneous* with p different-speed processors),
- a mapping strategy with replication, and either with data-parallelization or without.
- an objective (the period T_{period} or the latency T_{latency}),

determine an interval-based mapping that minimizes the objective. In the case with data-parallel stages, only intervals of length one can be data-parallelized for the pipeline, and we cannot data-parallelize $\mathcal{S}_0$ together with other independent stages for the fork, as explained previously. We see that there are sixteen possible combinations, hence sixteen optimization problems to solve. In fact there are more, because we also aim at exploring bi-criteria problems. For such problems, the objective becomes one of the following:

- given a threshold period $\mathcal{P}_{\mathsf{threshold}}$, determine a mapping whose period does not exceed $\mathcal{P}_{\mathsf{threshold}}$ and that minimizes the latency T_{latency};
- given a threshold latency $\mathcal{L}_{\mathsf{threshold}}$, determine a mapping whose latency does not exceed $\mathcal{L}_{\mathsf{threshold}}$ and that minimizes the period T_{period}.

Obviously, the bi-criteria problems are expected to be more difficult to solve than mono-criterion problems. Still, in some particular cases, such as pipelines or forks whose stages are all identical, we will be able to derive optimal algorithms of polynomial complexity.

4 Summary of complexity results

In this section, we provide a brief overview of all the complexity results established in this paper. As already mentioned, we restrict to the simplified model described in Section 3.4, and we focus on pipeline and fork graphs. Both *Homogeneous* and *Heterogeneous* platforms are considered, and we study several optimization criteria (latency, period, and both simultaneously).

We distinguish results for a model allowing data-parallelization, and a model without. Replication is allowed in all cases, since complexity results for a model with no replication and no data-parallelization are already known, at least for pipeline graphs [26, 25, 4].

Hom. platforms	without data-par			with data-par		
Objective	P	L	both	P	L	Both
Hom. pipeline	-			-		
Het. pipeline	Poly (str)			Poly (DP)		
Hom. fork	-	Poly (DP)		-	Poly (DP)	
Het. fork	Poly (str)	NP-hard		Poly (str)	NP-hard	
Het. platforms	**without data-par**			**with data-par**		
Objective	P	L	both	P	L	Both
Hom. pipeline	Poly (*)	-	Poly (*)	NP-hard		
Het. pipeline	NP-hard (**)	Poly (str)	NP-hard	-		
Hom. fork	Poly (*)			NP-hard		
Het. fork	NP-hard	-		-		

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

4.1 Summary

We summarize in Table 1 all the new complexity results.

In Table 1, the upper part refers to *Homogeneous* platforms while the lower part refers to *Heterogeneous* ones. The second column corresponds to the model without data-parallel stages, while the third column corresponds to the model with data-parallel stages (replication is allowed in both models). Finally, heterogeneous pipelines and forks are the application graphs described in Section 3.1 while the homogeneous versions refer to graph with identical stages: in a homogeneous pipeline, each stage has the same weight w, and in a homogeneous fork, the root stage has weight w_0 and each other stage has weight w. The polynomial entries with (str) mean that the optimal algorithm is straightforward. The entries with (DP) are filled using a dynamic programming algorithm. For a bi-criteria optimization, we compute in parallel the latency and the period. Entries (*) denote the more interesting contributions, obtained with a complex polynomial algorithm mixing binary search and dynamic programming. Most of the NP-completeness reductions are based on 2-PARTITION [12] and they are rather intuitive, except the entry (**), which is quite an involved reduction. Entries (-) mean that the result is directly obtained from another entry: a polynomial complexity for a more general instance of the application implies a polynomial complexity for the simpler case. Similarly, a NP-hard complexity for simpler cases implies the NP-completeness of the problem for harder instances.

4.2 Preliminary results

Lemma 1. *On Homogeneous platforms, there exists an optimal mapping which minimizes the period without using data-parallelism.*

Proof. Let us denote by s the speed of the processors of the *Homogeneous* platform. The minimum period obtained for a data-parallelized stage with w computations, mapped on the set of processors J, is $\frac{\mathsf{w}}{\sum_{j \in J} \mathsf{s}_j}$. In the case of the data-parallelization of an interval of independent stages in a fork graph, w is the sum of the corresponding computations. Since the platform is *Homogeneous*,

this period is equal to $\frac{\mathsf{w}}{|J|.\mathsf{s}}$, which is the minimum period obtained by replicating the same stage (or interval of stages) onto the same set of processors.

Any optimal mapping containing data-parallel stages can thus be transformed into a mapping which only contains replication. □

Lemma 2. *There exists an optimal mapping which minimizes the latency without replicating stages.*

Proof. Replicating a stage does not reduce the latency but only the period, since the latency is still the time required for an input to be processed by the slowest processor enrolled in the replication.

We can thus transform any optimal mapping which minimizes the latency by a new one realizing the same latency just by removing the extra processors assigned to any replicated stage. □

5 Complexity results for pipeline graphs

This section deals with pipeline graphs. For homogeneous platforms (Section 5.1), we revisit results from Subhlok and Vondran [26, 27] and provides new or simplified algorithms. For heterogeneous platforms (Section 5.2), to the best of our knowledge, all algorithms and complexity results are new.

5.1 Pipeline – *Homogeneous* platforms

First, we consider a pipeline application on *Homogeneous* platforms, and the three objective functions: (i) minimize the period, (ii) minimize the latency, (iii) minimize both the period and the latency. This section revisits results from Subhlok and Vondran [26, 27] and provides either new or simplified algorithms.

For both models, the solution is polynomial for each of the objective function for a heterogeneous pipeline (different computations for each stages). A fortiori, these results stand for homogeneous pipelines. In the following, s is the speed of the processors.

Theorem 1. *For Homogeneous platforms, the optimal pipeline mapping which minimizes the period can be determined in polynomial time, with or without data-parallelism.*

Proof. The minimum period that can be achieved by the platform is clearly bounded by $\frac{\sum_{i=1}^{n} \mathsf{w}_i}{\sum_{j=1}^{p} \mathsf{s}_j}$. Indeed, there are $\sum_{i=1}^{n} \mathsf{w}_i$ computations to do with a total resource of $\sum_{j=1}^{p} \mathsf{s}_j$.

On *Homogeneous* platforms, this minimum period can be achieved by replicating a single interval of all stages onto all processors. In this case, $\min_j \mathsf{s}_j = \mathsf{s}$ and $\sum_{j=1}^{p} \mathsf{s}_j = p.\mathsf{s}$, and thus the minimum period is obtained.

Another proof is the following: from Lemma 1, any optimal mapping can be transformed into a mapping composed of q intervals, each being replicated on b_k processors, $1 \leq k \leq q$. Interval k has a workload a_k, and $\sum_{k=1}^{q} a_k = \sum_{i=1}^{n} \mathsf{w}_i = \mathsf{w}$, $\sum_{k=1}^{q} b_k \leq \mathsf{p}$.

This mapping by interval realizes a maximum period $T = \max_{k=1}^{q}(\frac{a_k}{\mathsf{s}.b_k})$. Or,

$$\mathsf{w} = \sum_{k=1}^{q} \left(\frac{a_k}{b_k}.b_k \right) \leq \max_{k=1}^{q}(\frac{a_k}{b_k}). \sum_{k=1}^{q} b_k \leq T.\mathsf{p}.\mathsf{s}$$

It follows that the maximum period of this optimal interval mapping is greater than $\frac{\mathsf{w}}{\mathsf{p}.\mathsf{s}}$ which is the period obtained by mapping all stages as a single interval replicated onto all processors.

□

Theorem 2. *For Homogeneous platforms without data-parallelism, the optimal pipeline mapping which minimizes the latency can be determined in polynomial time.*

Proof. Following Lemma 2, since there is no data-parallelism, all mappings have the same latency $\sum_{i=1}^{n} \mathsf{w}_i/\mathsf{s}$, and thus any mapping minimizes the latency. □

Corollary 1. *For Homogeneous platforms without data-parallelism, the optimal pipeline mapping which minimizes both the period and the latency can be determined in polynomial time.*

Proof. Replicating the whole interval of stages onto all processors minimizes both criteria (Theorems 1 and 2). □

Theorem 3. *For Homogeneous platforms with data-parallelism, the optimal pipeline mapping which minimizes the latency can be determined in polynomial time.*

Proof. We exhibit here a dynamic programming algorithm which computes the optimal mapping.

We compute recursively the value of $L(i, j, q)$, which is the optimal latency that can be achieved by any interval-based mapping of stages S_i to S_j using exactly q processors. The goal is to determine $L(1, \mathsf{n}, \mathsf{p})$, since it is never harmful to use all processors on *Homogeneous* platforms (we can replicate or data-parallelize stages with the extra processors without increasing the latency).

The recurrence relation can be expressed as

$$L(i,j,q) = \min \begin{cases} \frac{\mathsf{w}_i}{q'.\mathsf{s}} + L(i+1,j,q-q') & \text{for } 1 \leq q' \leq q-1 \\ L(i,j-1,q-q') + \frac{\mathsf{w}_j}{q'.\mathsf{s}} & \text{for } 1 \leq q' \leq q-1 \\ L(i,k-1,q-q'-1) + \frac{\mathsf{w}_k}{q'.\mathsf{s}} + L(k+1,j,q-q'-1) & \text{for } \begin{cases} 1 \leq q' \leq q-2 \\ i < k < j \end{cases} \end{cases}$$

for $q > 2$, with the initialization

$$L(i,i,q) = \frac{\mathsf{w}_i}{q.\mathsf{s}} \quad \text{for } q \geq 1$$

$$L(i,j,1) = L(i,j,2) = \frac{\sum_{k=i}^{j} \mathsf{w}_k}{\mathsf{s}} \quad \text{for } i < j$$

$$L(i,j,0) = +\infty$$

The recurrence is easy to justify: to compute $L(i, j, q)$, we search over all possible data-parallelized stages. We cannot data-parallelize an interval, and it follows from Lemma 2 that the only way to reduce latency is to data-parallelize stages. The three cases make the difference between choosing the first stage, the last stage, or a stage in the middle of the interval. We have $L(i, j, 2) = L(i, j, 1)$ for $j > i$ since in this case there are not enough processors in order to data-parallelize stages between $\mathcal{S}_i$ and $\mathcal{S}_j$. The complexity of this dynamic programming algorithm is bounded by $O(\mathsf{n}^3\mathsf{p})$. □

It is not possible to extend the previous dynamic programming algorithm to deal with different-speed processors, since the algorithm intrinsically relies on identical processors in the recurrence computation. Different-speed processors would execute sub-intervals with different latencies. Because of this additional difficulty, the problem for *Heterogeneous* platforms seems to be very combinatorial: we prove that it is NP-hard below (Section 5.2).

Theorem 4. *For Homogeneous platforms with data-parallelism, the optimal pipeline mapping which minimizes (i) the period for a bounded latency, or (ii) the latency for a bounded period, can be determined in polynomial time.*

Proof. We exhibit here a dynamic programming algorithm which computes the optimal mapping.

We compute recursively the values of $(L, P)(i, j, q)$, which are the optimal latency and period that can be achieved by any interval-based mapping of stages S_i to S_j using exactly q processors. This computation can be done by minimizing L for a fixed P (the value of L takes $+\infty$ when the period exceeds P), or by minimizing P for a fixed L. The goal is to compute $(L, P)(1, \mathsf{n}, \mathsf{p})$.

The recurrence relation can be expressed as

$$(L,P)(i,j,q) = \min \begin{cases} (\frac{\sum_{k=i}^{j} \mathsf{w}_k}{\mathsf{s}}, \quad \frac{\sum_{k=i}^{j} \mathsf{w}_k}{q.\mathsf{s}}) \quad (1) \\ (L(i,k,q') + L(k+1,j,q-q'), \quad \max(P(i,k,q'), P(k+1,j,q-q'))) \quad (2) \\ \qquad \text{for} \qquad 1 \leq q' \leq q-1,\ i \leq k < j \end{cases}$$

We assume here that $i < j$, thus the whole interval is composed of more than one stage and it cannot be data-parallelized. In case (1), we replicate the whole interval, while in case (2) we split the interval in k.

The initialization relations are:

- For a single stage, the best choice is to data-parallelize it, and thus $(L, P)(i, i, q) = (\frac{\mathsf{w}_i}{q.\mathsf{s}}, \frac{\mathsf{w}_i}{q.\mathsf{s}})$.
- If there is only one processor, the only solution is to map the whole interval onto this processor: $(L, P)(i, j, 1) = (\frac{\sum_{k=i}^{j} \mathsf{w}_k}{\mathsf{s}}, \frac{\sum_{k=i}^{j} \mathsf{w}_k}{\mathsf{s}})$.

The recurrence is easy to justify: to compute $(L, P)(i, j, q)$, since we cannot data-parallelize an interval, either we replicate the whole interval, either we split it into two sub-intervals, possibly reduced to one stage and then data-parallelized. If one of the two criteria is fixed, we can then minimize the other one. The complexity of this dynamic programming algorithm is bounded by $O(\mathsf{n}^3\mathsf{p})$. □

As already stated for the mono-criterion optimization problem, this cannot be extended for *Heterogeneous* platforms, and the problem then becomes NP-complete (Section 5.2).

5.2 Pipeline – *Heterogeneous* platforms

Theorem 5. *For Heterogeneous platforms with data-parallelism, finding the optimal mapping for a homogeneous pipeline, for any objective (minimizing latency or period), is NP-complete.*

Proof. We consider the associated decision problems: (i) given a period P, is there a mapping of period less than P? (ii) given a latency L, is there a mapping of latency less than L? The problems are obviously in NP: given a period or a latency, and a mapping, it is easy to check in polynomial time that it is valid by computing its period and latency.

To establish the completeness, we use a reduction from 2-PARTITION [12]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset $I \subset \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{m} a_i$. Without loss of generality, we can assume that all a_j are different and strictly smaller than $S/2$ (hence $S/a_k > 2$ for all k).

We build the following instance $\mathcal{I}_2$ of our problem: the pipeline is composed of two stages with $\mathsf{w} = S/2$, and $\mathsf{p} = m$ processors with speeds $\mathsf{s}_j = a_j$ for $1 \leq j \leq m$.

For the latency decision problem, we ask whether it is possible to realize a latency of 2. 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. The solution to $\mathcal{I}_2$ which data-parallelizes both stages, one on the set of processors I, and one on the remaining processors, has clearly a latency of 2 since each stage completes in time 1.

On the other hand, if $\mathcal{I}_2$ has a solution, this solution has to use data-parallelism. Otherwise, the best latency that can be achieved is obtained by mapping both stages on the fastest processor k. But because $S/a_k > 2$, the achieved latency for a given stage is at least $\frac{S/2}{a_k} > 2$. The only way to obtain a latency smaller than 2 is thus to data-parallelize both stages. In the solution, let I be the set of processors assigned to the first stage. We have

$$\frac{S/2}{\sum_{j \in I} a_j} + \frac{S/2}{\sum_{j \notin I} a_j} \leq 2.$$

Let $a = \sum_{j \in I} a_j$. It follows that $S^2 \leq 4aS - 4a^2$, and thus the only solution is $a = S/2$. Therefore, the set of processors I is a solution to the instance of 2-PARTITION $\mathcal{I}_1$.

The proof for the period problem is quite similar. We use the same instance $\mathcal{I}_2$, and we ask whether we can realize a period of 1. When we have a solution of $\mathcal{I}_1$, the same mapping as above realizes the period 1. Given a solution of $\mathcal{I}_2$, we argue that the mapping is data-parallelized as above, because replicating one of the stages cannot be better than a data-parallelization of the stage, and replicating both stages on any subset of processors J would achieve a period

$$\frac{S}{|J| \times \min_{j \in J} a_j}$$

Since the a_j are all distinct, we have $|J| \times \min_{j \in J} a_j < S$ for any possible subset J.

□

The results are more interesting when data-parallelism is not possible. In this case, minimizing the latency can be done in polynomial time, but the complexity of the problem of minimizing the period depends on the pipeline type: it is polynomial for a homogeneous pipeline, while it becomes NP-complete for a heterogeneous pipeline.

Theorem 6. *For Heterogeneous platforms without data-parallelism, the optimal pipeline mapping which minimizes the latency can be determined in polynomial time.*

Proof. Following Lemma 2, since there is no data-parallelism, the minimum latency can be achieved by mapping the whole interval onto the fastest processor j, resulting in the latency $\sum_{i=1}^{n} \mathsf{w}_i/\mathsf{s}_j$.

This result holds for both heterogeneous and homogeneous pipeline. □

Theorem 7. *For Heterogeneous platforms without data-parallelism, the optimal homogeneous pipeline mapping which minimizes the period can be determined in polynomial time.*

First, we need a preliminary lemma which provides a regular form to the optimal solution. The idea is that replication should be done with a set of processors of similar speed, so that little computational resource is wasted. If we replicate some stages on a slow processor and a fast processor, the latter could only compute at the same rate as the former, and it would be idle for the remaining time. In other words, if we sort the processors according to their speeds, there exists an optimal solution which replicates stage intervals onto processor intervals:

Lemma 3. *Consider the model without data-parallelism. If an optimal solution which minimizes the period of a pipeline uses q processors, then consider the q fastest processors of the platform, denoted as $P_1, ..., P_q$, ordered by non-decreasing speeds: $\mathsf{s}_1 \leq ... \leq \mathsf{s}_q$. There exists an optimal solution which replicates intervals of stages onto k intervals of processors $I_r = [P_{s_r}, P_{e_r}]$, with $1 \leq r \leq k \leq q$, $s_1 = 1$, $e_k = q$, and $e_r + 1 = s_{r+1}$ for $1 \leq r < k$.*

Proof. We prove this lemma with an exchange argument. If the optimal solution is not using the q fastest processors $P_1, ..., P_q$, we can replace one of the slower processor by a fastest one without increasing the period. It could only decrease it, but the solution being optimal, the period remains the same.

We thus have an optimal solution using processors $P_1, ..., P_q$. This solution realizes a partition of the stages into intervals, and a set of processors is assigned to each interval. Consider the interval in which P_1 is enrolled. If the set of processors of this interval is not of the form $P_1, P_2, ..., P_{e_1}$, then we can exchange the fastest processors included in the set with the slower processors. This does not modify the period for this interval of stages since the only parameters of the period are the speed of P_1 and the number of processors assigned to this interval. Moreover, since we give fastest processors for the remaining stages of the pipeline, the period cannot be decreased either.

We iterate this transformation until the solution has the expected form, and the period can not have been increased during this transformation. Thus, the new solution is optimal. □

We build an optimal solution of the form stated in Lemma 3 in order to prove Theorem 7.

Proof. The algorithm performs a binary search on the period K in order to minimize it. We also perform a loop on the number of processors q implied in the optimal solution, with $1 \leq q \leq \mathsf{p}$. The q fastest processors are selected and ordered by increasing speed. We renumber them $P_1, ..., P_q$, with $\mathsf{s}_1 \leq \mathsf{s}_2 ... \leq \mathsf{s}_q$.

Finally, we solve a dynamic programming algorithm. The stages are all identical (homogeneous pipeline), with a workload w, and since there is an optimal solution composed of intervals of processors replicating intervals of stages, we need to form the intervals and decide how many stages they compute. $W(1, q)$ denotes the number of stages assigned to processors $P_1, ..., P_q$.

The recurrence writes:

$$W(i,j) = \max \begin{cases} \left\lfloor \frac{K.\mathsf{s}_i(j-i)}{\mathsf{w}} \right\rfloor & (1) \\ \max_{i \leq k < j}(W(i,k) + W(k+1,j)) & (2) \end{cases}$$

Case (1) corresponds to assigning an interval to processors $P_i, ..., P_j$. We compute the maximum number of stages that can be processed by these processors in order to fit into period K. Case (2) recursively tries to split the interval of processors at P_k. We maximize the number of stages that can be handled by both intervals of processors.

The initialization is the following:

$$W(i,i) = \left\lfloor \frac{K.\mathsf{s}_i}{\mathsf{w}} \right\rfloor$$

The recurrence is easy to justify since we search over all possible partitioning of the processors into consecutive intervals, in order to maximize the number of stages handled by these processors. At any time, the period is bounded by K. If $W(1,q) \geq \mathsf{n}$, then we have succeeded and we can try a smaller period in the binary search, otherwise we increase the period if we do not succeed for any values of q. The solution is the best one between all solutions using intervals of consecutive processors, and this solution is optimal (Lemma 3).

The loop over the number of processors is necessary since there is a trade-off to make between the number of processors used (possibly a large number of slow processors) and the speed of these processors (enrolling a slow processor in a replication scheme is decreasing the period of the whole interval).

The complexity of the dynamic programming algorithm is $O(\mathsf{p}^4)$ for each target value of the period. We need to bound the number of iterations in the binary search to establish the complexity. Intuitively, the proof goes as follows: we encode all application and platform parameters as rational numbers of the form $\frac{\alpha_r}{\beta_r}$, and we bound the number of possible values for the period as a multiple of the least commun multiple of all the integers α_r and β_r. The logarithm of this latter number is polynomial in the problem size, hence the number of iterations of the binary search is polynomial too[1]. Finally, we point out that in practice we expect only a very small number of iterations to be necessary to reach a reasonable precision. □

Theorem 8. *For Heterogeneous platforms without data-parallelism, the optimal homogeneous pipeline mapping for a bi-criteria optimization problem can be determined in polynomial time.*

Proof. The bi-criteria optimization problem is slightly more complex because of the latency which needs to be summed over all intervals. For a given latency L and a given period K, we perform a loop on the number of processors q and then a dynamic programming algorithm which aims at minimizing the latency. We succeed if L is obtained, and we can either perform a binary search on L to minimize the latency for a fixed period, or a binary search on K to minimize the period for a fixed latency.

The dynamic programming algorithm computes $L(\mathsf{n}, 1, q)$, where $L(m,i,j)$ is the minimum latency that can be obtained to map m pipeline stages on processors P_i to P_j, while fitting in period K. The recurrence writes:

$$L(m,i,j) = \min_{\substack{1 \leq m' < m \\ i \leq k < j}} \begin{cases} \frac{m.\mathsf{w}}{\mathsf{s}_i} & \text{if } \frac{m.\mathsf{w}}{(j-i).\mathsf{s}_i} \leq K \quad (1) \\ L(m',i,k) + L(m-m',k+1,j) & (2) \end{cases}$$

Case (1) corresponds to replicating the m stages onto processors $P_i, ..., P_j$, while case (2) splits the interval. The initialization writes:

$$L(1,i,j) = \begin{cases} \frac{\mathsf{w}}{\mathsf{s}_i} & \text{if } \frac{\mathsf{w}}{(j-i).\mathsf{s}_i} \leq K \\ +\infty & \text{otherwise} \end{cases}$$

[1]The interested reader will find a fully detailed proof for a very similar mapping problem in [20].

$$L(m,i,i) = \begin{cases} \frac{m.\mathsf{w}}{\mathsf{s}_i} & \text{if } \frac{m.\mathsf{w}}{\mathsf{s}_i} \leq K \\ +\infty & \text{otherwise} \end{cases}$$

The recurrence is easy to justify since we search over all possible partitioning of the processors into consecutive intervals, and over all possible number of stages assigned to these intervals. At any time, the period is bounded by K. If $L(\mathsf{n}, 1, q) \leq L$, then we have succeeded and we can try a smaller period or latency in the binary search. If we do not succeed for any values of q, we increase the period or latency. The solution is the best one between all solutions using intervals of consecutive processors, and it is easy to see that this solution is optimal, following Lemma 3, since all the exchanges performed in the proof of the lemma are not increasing the latency.

The complexity of the dynamic programming algorithm is $O(\mathsf{n}^2.\mathsf{p}^4)$ for each target value of the period and/or latency. The cost of the binary search can be bounded as stated in the mono-criterion proof. □

Theorem 9. *For Heterogeneous platforms without data-parallelism, the decision problem* Pipeline-Period-Dec *associated to the period minimization problem for heterogeneous pipeline applications is NP-complete.*

Definition 1 (Pipeline-Period-Dec)**.** *Given* n *elements* $\mathsf{w}_1, \mathsf{w}_2, \ldots, \mathsf{w}_n$*,* p *values* $\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_p$ *and a bound* K*, can we find a partition of* $[1..n]$ *into* q *intervals* $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_q$*, with* $\mathcal{I}_k = [d_k, e_k]$ *and* $d_k \leq e_k$ *for* $1 \leq k \leq q$*,* $d_1 = 1$*,* $d_{k+1} = e_k + 1$ *for* $1 \leq k \leq q-1$ *and* $e_q = \mathsf{n}$*, and an assignment function* $\sigma : \{1, 2, \ldots, \mathsf{p}\} \to \{1, 2, \ldots, q\}$*, such that*

$$\max_{1 \leq k \leq q} \frac{\sum_{i \in \mathcal{I}_k} \mathsf{w}_i}{|\mathcal{I}_k|.(\min_{j=1..\mathsf{p} | \sigma(j)=k} \mathsf{s}_j)} \leq K \quad ?$$

Proof. The Pipeline-Period-Dec decision problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that the partition into q intervals is valid and that the maximum period for each interval $k \in [1..q]$ does not exceed the bound K.

To establish the completeness, we use a reduction from NUMERICAL 3-DIMENSIONAL MATCHING (N3DM), which is NP-complete in the strong sense [12]. We consider an instance $\mathfrak{I}_1$ of N3DM: given $3m$ numbers $x_1, x_2, \ldots, x_m$, $y_1, y_2, \ldots, y_m$ and $z_1, z_2, \ldots, z_m$ and a bound M, does there exist two permutations σ_1 and σ_2 of $\{1, 2, \ldots, m\}$, such that $x_i + y_{\sigma_1(i)} + z_{\sigma_2(i)} = M$ for $1 \leq i \leq m$? Because N3DM is NP-complete in the strong sense, we can encode M in unary and assume that the size of $\mathfrak{I}_1$ is $O(m + M)$. We assume that $\forall i = 1..m$, $x_i < M$, $y_i < M$, $z_i < M$, and $\sum_{i=1}^{m} x_i + \sum_{i=1}^{m} y_i + \sum_{i=1}^{m} z_i = m.M$, otherwise $\mathfrak{I}_1$ cannot have a solution.

We build the following instance $\mathfrak{I}_2$ of Pipeline-Period-Dec:

- We define $\mathsf{n} = (M + 3)m$ stages, whose weights are outlined below:

$$A_1 \quad \underbrace{111...1}_{M} \quad C \quad D \quad | \quad A_2 \quad \underbrace{111...1}_{M} \quad C \quad D \quad | \quad \ldots \quad | \quad A_m \quad \underbrace{111...1}_{M} \quad C \quad D$$

Here, $R = \max(20, m + 1)$, $B = 2M$, $C = 5RM$, $D = 10R^2M^2$, and $A_i = B + x_i$ for

$1 \leq i \leq m$. To define the w_i formally for $1 \leq i \leq n$, let $N = M + 3$. We have for $1 \leq i \leq m$:

$$\begin{cases} \mathsf{w}_{(i-1)N+1} = A_i = B + x_i \\ \mathsf{w}_{(i-1)N+j} = 1 \text{ for } 2 \leq j \leq M+1 \\ \mathsf{w}_{iN-1} = C \\ \mathsf{w}_{iN} = D \end{cases}$$

- For the number of processors (and intervals), we choose $\mathsf{p} = 3m$. For $k = 1..\mathsf{p}$, we let s_k be the speed of processor P_k where, for $1 \leq j \leq m$:

$$\begin{cases} \mathsf{s}_j = B + M - y_j \\ \mathsf{s}_{m+j} = C + M - z_j \\ \mathsf{s}_{2m+j} = D \end{cases}$$

Finally, we ask whether there exists a solution matching the bound $K = 1$. Clearly, the size of $\mathfrak{I}_2$ is polynomial in the size of $\mathfrak{I}_1$. We now show that instance $\mathfrak{I}_1$ has a solution if and only if instance $\mathfrak{I}_2$ does.

Suppose first that $\mathfrak{I}_1$ has a solution, with permutations σ_1 and σ_2 such that $x_i + y_{\sigma_1(i)} + z_{\sigma_2(i)} = M$. For $1 \leq i \leq m$:

- We map each stage A_i and the following $z_{\sigma_2(i)}$ stages of weight 1 onto processor $P_{\sigma_1(i)}$.
- We map the following $M - z_{\sigma_2(i)}$ stages of weight 1 and the next stage, of weight C, onto processor $P_{m+\sigma_2(i)}$.
- We map the next stage, of weight D, onto the processor P_{2m+i}.

We do have a valid partition of all the stages into $\mathsf{p} = 3m$ intervals. For $1 \leq i \leq m$, the load and speed of the processors are indeed equal:

- The load of $P_{\sigma_1(i)}$ is $A_i + z_{\sigma_2(i)} = B + x_i + z_{\sigma_2(i)}$ and its speed is $B + M - y_{\sigma_1(i)} = B + x_i + z_{\sigma_2(i)}$.
- The load of $P_{m+\sigma_2(i)}$ is $M - z_{\sigma_2(i)} + C$, which is equal to its speed.
- The load and speed of P_{2m+i} are both D.

The mapping does achieve the bound $K = 1$, hence a solution to $\mathfrak{I}_1$.

Suppose now that $\mathfrak{I}_2$ has a solution, i.e. a mapping matching the bound $K = 1$. We first observe that $\mathsf{s}_i < \mathsf{s}_{m+j} < \mathsf{s}_{2m+k} = D$ for $1 \leq i, j, k \leq m$. The processors are thus categorized into slow processors, medium-speed processors and fast processors. Indeed $\mathsf{s}_j = B + M - y_j \leq B + M = 3M$, $5RM \leq \mathsf{s}_{m+j} = C + M - z_j \leq (5R+1)M$ and $D = 10R^2M^2$.

Let us first show that each of the m stages of weight D must be assigned to a processor of speed D, and it is the only stage assigned to this processor. If we add more stages to the interval, we need to replicate the interval which does not fit onto the processor of speed D. When replicated, the interval can have a load being the number of processors multiplied by the speed of the slowest of the processors used in the replication. Even adding all the medium-speed and/or slow processors into a replication is not enough because their speed reduce drastically the value of the minimum speed. For instance, m replicated processors of medium speed cannot process more than

$m.\min_i \mathsf{s}_{m+i} \leq mCM < D$. Similarly, any subset of processors including one non fast processor cannot process a stage D, not to speak of additional stages. The only possibility is thus to assign exactly one stage D to one single processor of speed D, since grouping some of these stages into intervals would increase the load for processors D and force replication.

These m singleton assignments divide the set of stages into m intervals, namely the set of stages before the first stage of weight D, and the $m-1$ sets of stages lying between two consecutive stages of weight D. The total weight of each of these m intervals is $A_i+M+C > B+M+C = (3+5R)M$. Thus, assigning the m slow processors to a single interval is not enough since the computation load of an interval processed by these processors would then be bounded by $m \times 3M < C$. Therefore, there is exactly one medium-speed processor assigned to each interval.

Moreover, this processor is not fast enough to handle the whole interval since its speed is less than $C+M$ and the load is greater than $C+M+B$. Since there remains only m available processors (the slow ones), each interval is assigned exactly one of these slow processors.

Consider such an interval A_i 111...1 C with M stages of weight 1, and let P_{i_1} and P_{m+i_2} be the two processors assigned to this interval (one slow and one medium-speed). If the whole interval is replicated onto both processors, the computing capacity is bounded by $2 \times 3M$ since one of the processor is a slow one. This is less than the load of the whole interval, and so it is not possible. Stages A_i and C are not assigned to the same processor (otherwise the whole interval would). So P_{i_1} receives stage A_i and h_i stages of weight 1 while P_{m+i_2} receives $M-h_i$ stages of weight 1 and stage C. It cannot be the other way round since P_{i_1} is too slow to handle stage C.

This defines two permutations $\sigma_1(i)$ and $\sigma_2(i)$ such that $i_1 = \sigma_1(i)$ and $i_2 = \sigma_2(i)$. Because the bound $K = 1$ must be achieved, we must have:

- $A_i + h_i = B + x_i + h_i \leq B + M - y_{\sigma_1(i)}$
- $M - h_i + C \leq C + M - z_{\sigma_2(i)}$

Therefore $z_{\sigma_2(i)} \leq h_i$ and $x_i + h_i \leq M - y_{\sigma_1(i)}$, and

$$\sum_{i=1}^{m} x_i + \sum_{i=1}^{m} z_i \leq \sum_{i=1}^{m} x_i + \sum_{i=1}^{m} h_i \leq mM - \sum_{i=1}^{m} y_i$$

By hypothesis, $\sum_{i=1}^{m} x_i + \sum_{i=1}^{m} z_i = mM - \sum_{i=1}^{m} y_i$, hence all inequalities are tight, and in particular $\sum_{i=1}^{m} x_i + \sum_{i=1}^{m} h_i = mM - \sum_{i=1}^{m} y_i = \sum_{i=1}^{m} x_i + \sum_{i=1}^{m} z_i$.

We can deduce that $\sum_{i=1}^{m} z_i = \sum_{i=1}^{m} h_i$, and since $z_{\sigma_2(i)} \leq h_i$ for all i, we have $z_{\sigma_2(i)} = h_i$ for all i.

Similarly, we deduce that $x_i + h_i = M - y_{\sigma_1(i)}$ for all i, and therefore $x_i + y_{\sigma_1(i)} + z_{\sigma_2(i)} = M$.

Altogether, we have found a solution for $\mathfrak{I}_1$, which concludes the proof.

□

6 Complexity results for fork graphs

This section deals with fork graphs. To the best of our knowledge, all results are new, including those for homogeneous platforms. We will use the word *interval* instead of *subset* when partitioning the stages of a fork graph and assigning them to processors, in order to keep the same terminology as in the previous section (and as discussed in Section 3).

6.1 Fork – *Homogeneous* platforms

Theorem 10. *For Homogeneous platforms, the optimal fork mapping which minimizes the period can be determined in polynomial time, with or without data-parallelism.*

Proof. The minimum period that can be achieved by the platform is clearly bounded by $\frac{\mathsf{w}_0+\sum_{i=1}^{\mathsf{n}} \mathsf{w}_i}{\sum_{j=1}^{\mathsf{p}} \mathsf{s}_j}$. Indeed, there are $\mathsf{w}_0 + \sum_{i=1}^{\mathsf{n}} \mathsf{w}_i$ computations to do with a total resource of $\sum_{j=1}^{\mathsf{p}} \mathsf{s}_j$.

On *Homogeneous* platforms, this minimum period can be achieved by replicating all tasks onto all processors. In this case, $\min_j \mathsf{s}_j = \mathsf{s}$ and $\sum_{j=1}^{\mathsf{p}} \mathsf{s}_j = \mathsf{p}.\mathsf{s}$, and thus the minimum period is obtained.

□

Theorem 11. *For Homogeneous platforms, the optimal homogeneous fork mapping which minimizes the latency can be determined in polynomial time, with or without data-parallelism.*

Note that the problem becomes NP-complete for a heterogeneous fork (Theorem 12).

Proof. We provide a dynamic programming algorithm for the bi-criteria optimization. The program is slightly different for the two cases, with or without data-parallelism.

In both cases, the idea is to loop over the number $0 \leq n_0 \leq \mathsf{n}$ of stages which belong to the same interval as the root stage $\mathcal{S}_0$, and which are mapped onto $1 \leq q_0 \leq \mathsf{p}$ processors. Then we can compute the minimum period and latency that can be obtained to map the $\mathsf{n} - n_0$ remaining stages on the $\mathsf{p} - q_0$ remaining processors (possibly as a bi-criteria optimization).

Case 1: with data-parallelism

In this case we can always data-parallelize the $\mathsf{n} - n_0$ remaining stages on the processors, which leads to both the minimum period and the minimum latency. We differentiate two cases: (1) $n_0 = 0$, then w_0 is alone and can be data-parallelized over q_0 processors, (2) the root stage is not alone and we can only replicate it on q_0 processors to decrease the period.

The minimum latency is obtained as:

$$\min_{\substack{1 \leq q_0 \leq \mathsf{p} \\ 1 \leq n_0 \leq \mathsf{n}}} \left(\frac{\mathsf{w}_0}{q_0.s} + \frac{\mathsf{n}.\mathsf{w}}{(\mathsf{p}-q_0).s} \ , \ \max(\frac{\mathsf{w}_0 + n_0.\mathsf{w}}{s}, \frac{\mathsf{w}_0}{s} + \frac{(\mathsf{n}-n_0)\mathsf{w}}{(\mathsf{p}-1).s}) \right)$$

Note that in this case, if w_0 is not alone, we do not replicate it since it does not decrease the latency, but instead we keep the $\mathsf{p} - 1$ remaining processors to data-parallelize the other stages.

For the bi-criteria solution, we define L_0 as the time required to compute w_0. Since we are in a flexible model, the other stages start computation at time L_0. P_0 is the maximum period of the processors involved in the computation of w_0.

If $n_0 = 0$, then w_0 is data-parallelized and

$$L_0 = P_0 = \frac{\mathsf{w}_0}{q_0.s}$$

Otherwise these stages are replicated, and

$$L_0 = \frac{\mathsf{w}_0}{s}, \quad P_0 = \frac{\mathsf{w}_0 + \mathsf{n}_0.\mathsf{w}}{q_0.s}$$

For the remaining stages, they are all data-parallelized, leading to the best latency and the best period. We thus have

$$L = \max\left(L_0 + \frac{\mathsf{n}_0.\mathsf{w}}{s}, L_0 + \frac{(\mathsf{n}-n_0).\mathsf{w}}{(\mathsf{p}-q_0).s}\right)$$

$$P = \max\left(P_0, \frac{(\mathsf{n}-n_0).\mathsf{w}}{(\mathsf{p}-q_0).s}\right)$$

Case 2: without data-parallelism

This second case is slightly more complex since we cannot data-parallelize the remaining $\mathsf{n}-n_0$ stages. We need to select the best way to split them into intervals and to replicate these intervals in order to minimize the latency for a given period, or minimize the period for a given latency (bi-criteria). P_0 and L_0 are defined as above in the case with w_0 replicated, but this time n_0 can be null. We compute $(P, L)(\mathsf{n}-n_0, \mathsf{p}-q_0)$ which represents the minimum period and latency that can be obtained to map the $\mathsf{n}-n_0$ remaining stages on the $\mathsf{p}-q_0$ remaining processors, and the recurrence writes:

$$(P,L)(i,q) = \begin{cases} (1)\ \left(\max(P_0, \frac{i.\mathsf{w}}{q.s}\ ), L_0 + \max(\frac{n_0.\mathsf{w}}{s}, \frac{i.\mathsf{w}}{s})\right) \\ (2)\ \min\limits_{\substack{1 \le k < i \\ 1 \le q' < q}} \left(\begin{array}{l} \max(P_0, P(k,q'), P(i-k, q-q')), \\ \quad L_0 + \max(\frac{n_0\mathsf{w}}{s}, L(k,q'), L(i-k, q-q')) \end{array} \right) \end{cases}$$

Case (1) corresponds to replicating the i stages onto q processors, while case (2) splits the interval. The initialization writes:

$$(P,L)(1,q) = \left(\max(P_0, \frac{\mathsf{w}}{q.s}),\ L_0 + \frac{\max(n_0,1)\mathsf{w}}{s}\right)$$

$$(P,L)(i,1) = \left(\max(P_0, \frac{i.\mathsf{w}}{s}),\ L_0 + \frac{\max(n_0,i)\mathsf{w}}{s}\right)$$

This recurrence is easy to justify since we try all possible splittings, and thus we explore all the cases. We can solve the recurrence either for a fixed latency or for a fixed period, minimizing the other criterion.

Summary. All the previous algorithms provide optimal solutions for the latency minimization and bi-criteria problems, in polynomial time. The complexity is always bounded by $O(\mathsf{n}^3\mathsf{p}^3)$.

□

Theorem 12. *For Homogeneous platforms, the optimal heterogeneous fork mapping which minimizes the latency is a NP-complete problem, with or without data-parallelism.*

Proof. We consider the associated decision problem: given a latency L, is there a mapping of latency less than L? The problem is obviously in NP: given a latency and a mapping, it is easy to check in polynomial time that it is valid by computing its latency.

To establish the completeness, we use a reduction from 2-PARTITION [12]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset $I \subset \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{m} a_i$.

We build the following instance $\mathcal{I}_2$ of our problem: the fork is composed of $m+1$ stages, with $\mathsf{w}_0 = 1$ and $\mathsf{w}_i = a_i$ for $i = 1..m$. The platform is composed of $\mathsf{p} = 2$ processors, both of speed 1. Is it possible to achieve a latency of $1 + S/2$? 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. The same reduction works for both models (with or without data-parallelism).

Suppose first that $\mathcal{I}_1$ has a solution, the subset I. The solution to $\mathcal{I}_2$ which gives the root stage plus the stages $\mathcal{S}_i, i \in I$ to one processor and the remaining stages to the other processor clearly achieves a latency of $1 + S/2$, since it is the computational time required by both processors (for the second one, we need to add the time 1 required to process w_0).

On the other hand, if $\mathcal{I}_2$ has a solution, let us show that it cannot use data-parallelism, and thus that the result holds true for both model. If $\mathcal{S}_0$ is associated with some other stages, it cannot be data-parallelized, so it must be alone on its processor if we want to data-parallelize a part of the fork. However, in this case, we need at least one processor for $\mathcal{S}_0$ and one for the remaining stages, and thus we do not have enough processors to data-parallelize anything.

Replication cannot be used to reduce latency, so the only way to obtain a latency smaller than $1 + S$ (everything on one processor), is to share the set of stages between both processors. The latency of $1+S/2$ is reached only if each processor is in charge of a computational load of exactly $S/2$ (without counting w_0). The subset of stages that a processor handles is thus a solution to $\mathcal{I}_1$, since it realizes a 2-partition of the set of stages. □

6.2 Fork – *Heterogeneous* platforms

Theorem 13. *For Heterogeneous platforms with data-parallelism, finding the optimal mapping for a homogeneous fork, for any objective (minimizing latency or period), is NP-complete.*

Proof. We consider the associated decision problems: (i) given a period P, is there a mapping of period less than P? (ii) given a latency L, is there a mapping of latency less than L? The problems are obviously in NP: given a period or a latency, and a mapping, it is easy to check in polynomial time that it is valid by computing its period and latency.

To establish the completeness, we use a reduction from 2-PARTITION [12]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset $I \subset \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{m} a_i$.

We build the following instance $\mathcal{I}_2$ of our problem: the fork is composed of two stages $\mathcal{S}_0$ and S_1 with $\mathsf{w} = S/2$, and $\mathsf{p} = m$ processors with speeds $\mathsf{s}_j = a_j$ for $j = 1..m$.

This instance is indeed a pipeline, thus the reduction is exactly similar to the one of Theorem 5 (same problem for the pipeline application), which ends the proof. □

The problem is already NP-hard for a homogeneous fork, so it remains NP-hard for the more general case of a heterogeneous fork.

Theorem 14. *For Heterogeneous platforms without data-parallelism, the optimal homogeneous fork mapping for any objectives can be determined in polynomial time.*

Note that the problem becomes NP-hard if we consider a heterogeneous fork (Theorem 15).

The algorithm that we provide for this problem is quite similar to the one for pipeline graphs (Theorem 7). We need a variant of Lemma 3 for the fork, to express the solution with intervals of similar speed processors, but care should be taken of $\mathcal{S}_0$. The idea is that $\mathcal{S}_0$ is handled by one of the intervals, and we need to identify this interval.

Lemma 4. *Consider the model without data-parallelism. If an optimal solution which minimizes the period or the latency of a fork uses q processors, and the slowest processor involved in the computation of $\mathcal{S}_0$ is P_0, then let us consider the q fastest processors of the platform, denoted $P_1, ..., P_q$, ordered by non-decreasing speeds: $\mathsf{s}_1 \leq ... \leq \mathsf{s}_q$. Let q_0 be the number of P_0 in the new ordering, and if P_0 is not in $P_1, ..., P_q$, this means that P_0 is slower than P_1, and we set $q_0 = 1$.*

There exists an optimal solution which replicates intervals of stages onto k intervals of processors $I_r = [P_{s_r}, P_{e_r}]$, with $1 \leq r \leq k \leq q$, $s_1 = 1$, $e_k = q$, and $s_r + 1 = e_{r+1}$ for $1 \leq r < k$, and one of the intervals starts with P_{q_0} and is in charge of $\mathcal{S}_0$.

Proof. The proof uses an exchange argument, similarly to the proof of Lemma 3. If P_0 is slower than P_1, we can replace it by P_1 because this exchange can only decrease period and latency.

The solution realizes a partition of the stages into intervals, each interval being handled by a set of processors. One of the interval is in charge of $\mathcal{S}_0$ (possibly an empty interval with no other stage).

At this point, it is easy to exchange fast processors for slower ones when they are implied in a replication with an even slower processor, as we were doing for the pipeline case. □

We are now ready to prove Theorem 14, expressing the solution in the form exhibited by Lemma 4. Note that $\mathcal{S}_0$ can be handled by any interval in the optimal solution, thus we need to perform a loop on the slowest processor of the processor interval that executes the root stage.

Proof. We write the recurrence aiming at the general bi-criteria optimization problem. A binary search is performed either on the period or on the latency, and the other parameter is fixed, thus we build a mapping fitting a period K and a latency L.

We perform a loop on $q = 1..\mathsf{p}$, the number of enrolled processors, then a loop on $q_0 = 1..q$, the first processor of the interval which handles $\mathcal{S}_0$, and we compute recursively the number of stages that we can give to an interval of processors $[P_i, P_j]$, and the corresponding period and latency that are achieved. As in Theorem 7, the processors are ordered by non-decreasing speeds, and we maximize $W(1, q)$ for the given period and latency. For each interval except the one handling $\mathcal{S}_0$, we need to add $\frac{\mathsf{w}_0}{\mathsf{s}_{q_0}}$ to the latency, which corresponds to the time at which the interval of processors starts working. Thus we define $L_0 = L - \frac{\mathsf{w}_0}{\mathsf{s}_{q_0}}$, which is the latency that these intervals must achieve in order to fit into L. If $L_0 \leq 0$ then there is no solution.

Since we need to split the processors at q_0, for $q_0 > 1$ we compute $W(1, q) = W(1, q_0 - 1) + W(q_0, q)$. We fail if $W(1, q) < \mathsf{n}$, since the total number of stages that need to be processed is n.

The recurrence then writes, for the interval of processors $[P_i, P_j]$:

$$W(i,j) = \max \begin{cases} \text{if } i = q_0, \begin{cases} \min(\left\lfloor \frac{K.\mathsf{s}_i(j-i)-\mathsf{w}_0}{\mathsf{w}} \right\rfloor, \left\lfloor \frac{L.\mathsf{s}_i-\mathsf{w}_0}{\mathsf{w}} \right\rfloor) & \text{if } W(i,j) \geq 0 \\ -\infty \quad \text{otherwise} \end{cases} & (1a) \\ \text{if } i \neq q_0, \min(\left\lfloor \frac{K.\mathsf{s}_i(j-i)}{\mathsf{w}} \right\rfloor, \left\lfloor \frac{L_0.\mathsf{s}_i}{\mathsf{w}} \right\rfloor) & (1b) \\ \max_{i \leq k < j}(W(i,k) + W(k+1,j)) & (2) \end{cases}$$

Case (1) corresponds to assigning an interval to processors $P_i, ..., P_j$. We distinguish whether this interval is in charge of $\mathcal{S}_0$ (1a) or not (1b). Depending on w_0, it may happen that the whole interval cannot fit in the given period or latency, even with $W(i, j) = 0$. In this case, we set $W(i, j) = -\infty$ to ensure that this solution will not be chosen. We compute the maximum number

of stages that can be processed by these processors in order to fit into period K and latency L. Case (2) recursively tries to split the interval of processors at P_k. Initially, the period and latency are always fitting into K or L, and the property always remains true. We maximize the number of stages that can be handled by both intervals of processors.

The initialization is the following:

$$W(i,i) = \left\lfloor \frac{\min(K, L_0).\mathsf{s}_i}{\mathsf{w}} \right\rfloor \quad \text{for } i \neq q_0$$

$$W(q_0,q_0) = \begin{cases} \left\lfloor \frac{\min(K,L).\mathsf{s}_0 - \mathsf{w}_0}{\mathsf{w}} \right\rfloor & \text{if } \frac{\mathsf{w}_0}{\mathsf{s}_0} \geq \min(K,L) \\ -\infty \quad \text{otherwise} \end{cases}$$

The recurrence is easy to justify since we search over all possible partitionings of the processors into consecutive intervals, in order to maximize the number of stages handled by these processors. At any time, the period and latency are bounded by K and L. If $W(1,q) \geq \mathsf{n}$, then we have succeeded and we can try a smaller period or latency in the binary search, otherwise we increase the value if we do not succeed for any values of q and q_0. The solution is the best one between all solutions in the form of Lemma 4, and following the lemma, it is optimal.

The complexity of the dynamic programming algorithm is $O(\mathsf{p}^5)$ for each target value of the period and/or latency. The number of iterations in the binary search can be bounded exactly as in the proof of Theorem 7. □

Theorem 15. *For Heterogeneous platforms without data-parallelism, finding the optimal mapping for a heterogeneous fork, for any objective (minimizing latency or period), is NP-complete.*

Proof. For the latency objective the problem is already NP-complete on *Heterogeneous* platforms, implying this result.

For the period, we consider the associated decision problem: given a period P, is there a mapping of period less than P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time that it is valid by computing its period.

To establish the completeness, we use a reduction from 2-PARTITION [12]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset $I \subset \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{m} a_i$.

We build the following instance $\mathcal{I}_2$ of our problem: the fork is composed of $m+2$ stages such that $\mathsf{w}_0 = S, \mathsf{w}_{m+1} = S$, and $\mathsf{w}_i = a_i$ for $i = 1..m$. The total load is thus $3S$. The platform is composed of 2 processors whose speeds are $\mathsf{s}_1 = 5 \times \frac{S}{2}$ and $\mathsf{s}_2 = \frac{S}{2}$. We ask the following question: is it possible to achieve a period $P = 1$? 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, the subset I. The solution to $\mathcal{I}_2$ which gives $\mathcal{S}_0$, $\mathcal{S}_{m+1}$ and the stages $\mathcal{S}_i, i \in I$ to P_1 and the remaining stages to P_2 clearly achieves a period of 1, since the load assigned to each processor is equal to its speed.

On the other hand, if $\mathcal{I}_2$ has a solution, let us show that this solution does not use replication. Since there are only two processors in the platform, the only way to replicate consists in replicating the whole fork onto both processors, thus achieving a period $\frac{3S}{2 \times \frac{S}{2}} = 3 > 1$. Also, if the solution was using only the fastest processor P_1, then the period would be $\frac{3S}{\frac{5S}{2}} = \frac{6}{5} > 1$. Therefore, $\mathcal{I}_2$ is sharing the load between both processors, and P_1 must handle a total load of $5 \times \frac{S}{2}$, while P_2 must

handle a total load of $\frac{S}{2}$. Stages $\mathcal{S}_0$ and $\mathcal{S}_{m+1}$ cannot be handled by P_2, thus P_1 is in charge of them, and the other stages are shared between both processors following a 2-partition. Therefore, $\mathcal{I}_1$ has a solution. □

6.3 Extension to fork-join graphs

We have concentrated in this section on the complexity of mapping algorithms for fork graphs, but it is also very common to have fork-join graphs, in which a final stage, $\mathcal{S}_{\mathsf{n}+1}$, is gathering all the results and performing some final computations.

In this section we briefly explain that all the complexity results obtained for fork graphs can be extended to fork-join graphs. In other words, the complexity is not modified by the addition of the final stage.

Clearly, all the problem instances which are NP-complete for a simple fork are still NP-complete for a fork-join graph. The question is to check whether we can extend the polynomial algorithms to handle fork-join or not. The answer is positive in all cases. We do not formally present these new algorithms, but rather give an insight on how to design the extensions.

First, consider the polynomial entries on *Homogeneous* platforms. The straightforward algorithm to minimize the period for a fork is still working for a fork-join, since the replication of the whole graph on all the processors still provides the optimal period. Minimizing the latency or a bi-criteria algorithm was requiring a dynamic programming algorithm for a homogeneous fork (the problem being NP-hard for a heterogeneous fork). The dynamic programming algorithms used in the proof of Theorem 11 extend to fork-join graphs by adding two external loops, the first over the number of stages which belong to the same interval as the final stage $\mathcal{S}_{\mathsf{n}+1}$, and the second over the number of processors onto which these latter stages are mapped. We should also consider the case in which $\mathcal{S}_0$ and $\mathcal{S}_{\mathsf{n}+1}$ are in the same interval. The rest of the algorithms is unchanged. Taking the new loops into account, we add a factor $O(\mathsf{np})$ to the complexity, which finally becomes $O(\mathsf{n}^4\mathsf{p}^4)$.

The only polynomial algorithm on *Heterogeneous* platforms is for a homogeneous fork without data-parallelism. This corresponds to the algorithm of Theorem 14, which executes a binary search, and a dynamic programming computation at each iteration of the binary search. For a homogeneous fork-join graph, Lemma 4 can be extended to describe the form of an optimal solution, still using intervals of processors with consecutive speeds. One of the processor intervals must be in charge of $\mathcal{S}_{n+1}$, it can either be the one in charge of $\mathcal{S}_0$ or another one. We need to distinguish both cases, and to add a loop on the first processor of the interval which handles $\mathcal{S}_{\mathsf{n}+1}$ whenever it is different from the one which handles $\mathcal{S}_0$. The formula are then slightly modified to take into account the time of the final computations, but the algorithm remains similar. We have added $O(\mathsf{p})$ to the complexity, leading to a total complexity of $O(\mathsf{p}^6)$ for each iteration of the binary search.

On the theoretical side, we see that extending all the complexity results to fork-join graphs was not very difficult. But we believe that this extension was worth mentioning, because of the importance of fork-join graphs in many practical applications. In fact, numerous parallel applications can be expressed with the master-slave paradigm: the master initiates some computations, and then distributes (scatters) data to the slaves (in our case, stages $\mathcal{S}_1, ..., \mathcal{S}_{\mathsf{n}}$ of the fork-join). Results are then collected and combined (join operation).

7 Related work

As already mentioned, this work is an extension of the work of Subhlok and Vondran [26, 27] for pipeline applications on homogeneous platforms. We extend the complexity results to heterogeneous platforms and fork applications, for a simpler model with no communications.

We have also discussed the relationship with the chains-to-chains problem [9, 15, 13, 16, 21, 22] in Section 1. In this paper we extend the problem by adding the possibility to replicate or to data-parallelize intervals of stages, which modifies the complexity.

Several papers consider the problem of mapping communicating tasks onto heterogeneous platforms, but for a different applicative framework. In [28], 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. 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 [3] 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 [3] 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 [11]. 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 [5, 6, 25]. A typical application is a chain of consecutive filtering operations, to be executed on a very large data set. The task graphs targeted by DataCutter are more general than linear pipelines or forks, but still more regular than arbitrary DAGs, which makes it possible to design efficient heuristics to solve the previous placement and replication optimization problems. However, we point out that a recent paper [30] targets workflows structured as arbitrary DAGs and considers bi-criteria optimization problems on homogeneous platforms. The paper provides many interesting ideas and several heuristics to solve the general mapping problem. It would be very interesting to experiment these heuristics on all the combinatorial instances of pipeline and fork optimization problems identified in this paper.

8 Conclusion

In this paper, we have considered the important problem of mapping structured workflow applications onto computational platforms. Our main focus was to study the complexity of the most tractable instances, in order to give an insight of the combinatorial nature of the problem. We have concentrated on simple application schemes, namely pipeline and fork computations, and studied the mapping of such computation patterns on *Homogeneous* and *Heterogeneous* platforms with no communication costs. We considered the two major objective functions, minimizing the latency and minimizing the period, and also studied bi-criteria optimization problems. Already, several instances of the problem are shown to be NP-complete, while others can be solved with complex polynomial algorithms, mixing binary search and dynamic programming techniques.

It is interesting to see that most problems are already combinatorial, because it shows that there is no chance to find an optimal mapping when adding the complexity of communications. We have succeeded to establish the complexity of all the problems exposed in Table 1, sometimes distinguishing whether the application is fully regular or not. Some results are surprising: for instance consider the bi-criteria optimization problem on a pipeline application mapped onto a *Heterogeneous* platform. If there is no data-parallelism, the problem is polynomial for a regular application while it becomes NP-complete when pipeline stages have different computation costs. The same problem is NP-hard in both cases if we add the possibility to data-parallelize stages of the pipeline. The results for the fork pattern have been extended to fork-join computations: the complexity remains the same in all cases.

We believe that this exhaustive study of complexity provides a solid theoretical foundation for the study of single criterion or bi-criteria mappings, with the possibility to replicate and possibly data-parallelize application stages.

As future work, we could select some of the polynomial instances of the problem and try to assess the complexity when adding some communication parameters to the application and to the platform. Also, heuristics should be designed to solve the combinatorial instances of the problem. We have restricted to simple communication schemes, since the problem on general DAGs is already too difficult, but we could build heuristics based on some of our polynomial algorithms to solve more complex instances of the problem, with general application graphs structured as combinations of pipeline and fork kernels.

Acknowledgement

We would like to thank Umit V. Catalyurek, Tahsin M. Kurc, Joel H. Saltz, and Naga Vydyanathan for insightful discussions on workflows and multi-criteria optimization. Special thanks go to Naga for her numerous comments on an earlier version of the paper.

References

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[2] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*. IEEE Computer Society Press, 1998.

[3] 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.

[4] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. Research Report 2007-05, LIP, ENS Lyon, France, Jan. 2007. Available at `graal.ens-lyon.fr/~yrobert/`. Updated version of RR-2006-40. Short version to appear in ICCS'2007.

[5] 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.

[6] 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.

[7] 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.

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

[9] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Computers*, 37(1):48–57, 1988.

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

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

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] P. Hansen and K.-W. Lih. Improved algorithms for partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Computers*, 41(6):769–771, 1992.

[14] B. Hong and V. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'2003)*. IEEE Computer Society Press, 2003.

[15] M. A. Iqbal. Approximate algorithms for partitioning problems. *Int. J. Parallel Programming*, 20(5):341–361, 1991.

[16] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Trans. Parallel and Distrbuted Systems*, 6(2):170–175, 1995.

[17] S. Khuller and Y. Kim. On broadcasting in heterogenous networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.

[18] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[19] P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.

[20] L. Marchal, V. Rehn, Y. Robert, and F. Vivien. Scheduling and data redistribution strategies on star platforms. Research Report 2006-23, LIP, ENS Lyon, France, June 2006.

[21] B. Olstad and F. Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[22] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *J. Parallel Distributed Computing*, 64(8):974–996, 2004.

[23] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[24] 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.

[25] 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.

[26] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 134–143. ACM Press, 1995.

[27] 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.

[28] 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.

[29] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[30] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Saddayappan, and J. Saltz. An approach for optimizing latency under throughput constraints for application workflows on clusters. Research Report OSU-CISRC-1/07-TR03, Ohio State University, Columbus, OH, Jan. 2007. Available at `ftp://ftp.cse.ohio-state.edu/pub/tech-report/2007`.

[31] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, 1994.