hal-00521712, version 1 - 28 Sep 2010

A Survey of Pipelined Workflow Scheduling: Models and Algorithms

ANNE BENOIT
Institut Universitaire de France and École Normale Supérieure de Lyon, France
and
UMIT V. CATALYUREK
Department of Biomedical Informatics and Department of Electrical & Computer Engineering, The Ohio State University
and
YVES ROBERT
Institut Universitaire de France and École Normale Supérieure de Lyon, France
and
ERIK SAULE
Department of Biomedical Informatics, The Ohio State University

A large class of applications need to execute the same workflow on different data sets. Efficient execution of such applications necessitates intelligent distribution of the application components and tasks on a parallel machine, and orchestrating the execution by utilizing task-, data-, pipelined-, and replicated-parallelism. The scheduling problem that encompasses all of these techniques is called *pipelined workflow scheduling*, and has been widely studied in the last decade. Multiple models and algorithms flourished to tackle various programming paradigms, constraints, machine behaviors or goals. This paper surveys the field by summing up and structuring known results and approaches.

General Terms: Algorithms, Performance, Theory.
Additional Key Words and Phrases: workflow programming, filter-stream programming, scheduling, pipeline, throughput, latency, models, algorithms, distributed systems, parallel systems.

LIP Research Report RR-LIP-2010-28

Authors' Address: Anne Benoit and Yves Robert, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 Allée d'Italie 69364 LYON Cedex 07, FRANCE, {Anne.Benoit|Yves.Robert}@ens-lyon.fr.
Umit V. Catalyurek and Erik Saule, The Ohio State University, 3190 Graves Hall — 333 W. Tenth Ave., Columbus, OH 43210, USA, {umit|esaule}@bmi.osu.edu.

1. INTRODUCTION

For large-scale applications targeted to parallel and distributed computers, finding an efficient task and communication mapping and schedule is critical to reach the best possible application performance. At the heart of the scheduling process is the *workflow* of an application: an abstract representation that expresses the atomic computation units and their data dependencies. A large class of applications needs to execute the same workflow on different independent data items. Examples of such applications are video processing [GRRL05], image analysis [SKS+09], motion detection [KRC+99], signal processing [CkLW+00; HFB+09], databases [CHM95], molecular biology [RKO+03], and various scientific data analyses, including particle physics [DBGK03], earthquake [KGS04], weather and environmental data analyses [RKO+03].

To execute data items in parallel, two simple approaches are commonly used. The first one consists of finding an efficient parallel execution schedule for one single data item, and then executing all the data items using the same schedule, one after the other. This approach is well known as *latency* or *makespan* optimization. Although some good algorithms are known for this problem [KA99b; KA99a], the resulting performance of the system may be far from the peak performance of the target parallel platform. The workflow may have a limited degree of parallelism for efficient execution of a single data item, and hence the parallel machine may not be fully utilized.

The second approach consists in executing multiple data items in parallel. This approach complicates the scheduling problem, because the execution of one data item imposes some constraints on the execution of the next one. The scheduling problems which use both intra data item and inter data item parallelisms are called *pipelined workflow scheduling*, or in short *pipelined scheduling*. (Notice that some applications may not allow this kind of parallelism ; for instance, the applications with a feedback loop like iterative solvers.)

Pipelined workflow scheduling has been widely studied in the last decade. The pipelined execution model is the core of many software and programming middleware. It is used on different types of parallel machine such as SMP (Intel TBB [Rei07]), clusters (DataCutter [BKC+01], Anthill [TFG+08], Dryad [IBY+07]), grid computing environment (Microsoft AXUM [Mic09], DAGMan [CT02], Taverna [OGA+06], LONI [MGPD+08], Kepler [BML+06]), and more recently on cluster with accelerators (DataCutter [HCR+08] and DataCutter-Lite [HC09]). Multiple models and algorithms have emerged to deal with various programming paradigms, hardware constraints, and scheduling objectives.

To evaluate the performance of a schedule, various optimization criteria are used in the literature. The most common ones are (i) the *latency* (denoted by $\mathcal{L}$), or *makespan*, which is the maximum time a data item spends in the system, and (ii) the *throughput* (denoted by $\mathcal{T}$), which is the number of data items processed per time unit. The period of the schedule (denoted by $\mathcal{P}$) is the time elapsed between two consecutive data items entering the system. Note that the period is commonly used instead of the throughput, since it is the inverse of the achieved throughput. Depending on the application, a combination of multiple performance objectives may be desired. For instance, an interactive video processing application

hal-00521712, version 1 - 28 Sep 2010

(such as SmartKiosk [KRC+99], a computerized system that interacts with multiple people using cameras) needs to be reactive while ensuring a good frame rate; these constraints call for an efficient latency/throughput trade-off. Other criteria may include reliability, resource cost, and energy consumption.

Several types of parallelism can be used to achieve good performance. *Task-parallelism* is the most well known. It consists in concurrently executing independent tasks of the workflow for the same data item; it can help minimize the workflow latency. *Pipelined-parallelism* is used when two dependent tasks in the workflow are being executed simultaneously on different data items. The goal is to improve the throughput of the application, possibly at the price of more communication, hence potentially a larger latency. *Replicated-parallelism* can improve the throughput of the application, because several copies of a single task operate on different data items concurrently; this is especially useful in situations where more computational resources than workflow tasks exist. Finally, *data-parallelism* may be used when some tasks contain inherent parallelism. It corresponds to using several processors to execute a single task for a single data item.

The rest of this paper is organized as follows. Before going into technical detail, Section 2 illustrates the various parallelism techniques, task properties, and their impact on objective functions using a motivating example.

The first issue when dealing with a pipelined application is to select the right model among the tremendous number of variants that exist. To solve this issue, Section 3 organizes the different characteristics that the target application can exhibit into three components: the workflow model, the system model, and the performance model. This organization helps position a given problem with respect to related work.

The second issue is building the relevant scheduling problem from the model of the target application. There is no direct formulation going from the model to the scheduling problem, so we cannot provide a general method to derive the scheduling problem. However, in Section 4, we illustrate the main techniques on basic problems and show how the application model impacts the scheduling problem. The scheduling problems become either more or less complicated depending upon the application requirements. As usual in optimization theory, the most basic (and sometimes unrealistic) problems can usually be solved in polynomial time, whereas the most refined and accurate models usually lead to NP-hard problems. Although the complexity of some problems is still open, Section 4 concludes by highlighting the known frontier between polynomial and NP-complete problems.

Finally, in Section 5, we survey various techniques that can be used to solve the scheduling problem, i.e., finding the best parallel execution of the application according to the performance criteria. We provide optimal algorithms to solve the simplest problem instances in polynomial time. For the most difficult instances, we present some general heuristic methods which aim at giving good approximate solutions.

2. MOTIVATING EXAMPLE

In this section, we focus on a simple pipelined application and emphasize the need for scheduling algorithms.

hal-00521712, version 1 - 28 Sep 2010

Consider an application composed of four tasks, which dependencies form a linear chain: a data item must first be processed by task S_1 before it can be processed by S_2, then S_3, and finally S_4. The processing of a data item takes respectively 5, 2, 3, and 20 time units for tasks S_1, S_2, S_3 and S_4, as illustrated in Fig. 1(a). If two consecutive tasks are executed on two distinct processors, then a communication cost must be paid, in order to transfer the intermediate result. These costs are set respectively to 20, 15 and 1 for communications $S_1 \to S_2$, $S_2 \to S_3$, and $S_3 \to S_4$. The target platform consists of three processors, with various speeds and interconnection bandwidths, as illustrated in Fig. 1(a). If task S_1 is scheduled to be executed on processor P_2, a data item is processed within $\frac{5}{1} = 5$ time units, while the execution on the faster processor P_1 requires only $\frac{5}{10} = 0.5$ time units. Similarly, the communication of a data of cost c from processor P_1 to processor P_2 takes $\frac{c}{1}$ time units, while it is ten times faster to communicate from P_1 to P_3.

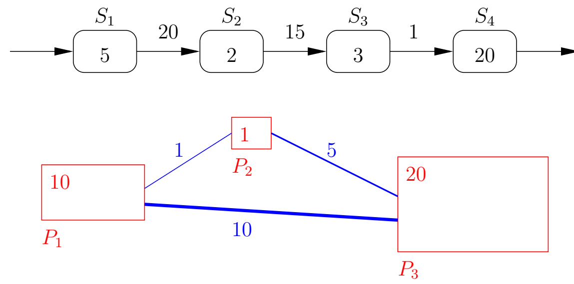
First examine the execution of the application sequentially on the fastest processor, P_3 (see Fig. 1(b)). For such an execution, there is no communication cost to pay, and because of the dependencies between tasks, this is actually the best way to execute a single data item. The latency is computed as $\mathcal{L} = \frac{5+2+3+20}{20} = 1.5$. A new data item can be processed once the previous one is finished, hence the period $\mathcal{P} = \mathcal{L} = 1.5$.

Of course, this sequential execution does not exploit any parallelism. Since there are no independent tasks in this application, we cannot use *task-parallelism* here. However, we now illustrate *pipelined-parallelism*: different tasks are scheduled on distinct processors, and thus they can be executed simultaneously on different data items. In the execution of Fig. 1(c), all processors are used, and we balanced the computation requirement of tasks according to processor speeds. The performance of such a parallel execution turns out to be quite bad, because lots of large communications occur. The latency is now obtained by summing up all computation and communication times: $\mathcal{L} = \frac{5}{10} + 20 + 2 + 15 + \frac{3}{10} + \frac{1}{10} + \frac{20}{20} = 38.9$. Moreover, the period is not better than the one obtained with the sequential execution presented previously because communications become the bottleneck of the execution. Indeed, the transfer from S_1 to S_2 takes 20 time units, and therefore the period cannot be better than 20: $\mathcal{P} \geq 20$. This example of execution illustrates that parallelism should be used with caution.

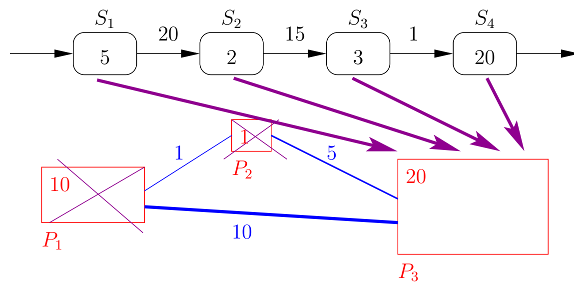
However, one can obtain a throughput better than that of the sequential execution as shown in Fig. 1(d). In this case, we enforce some resource selection: the slowest processor is discarded since it only slows down the whole execution. We process different data items in parallel: the k-th data item is executed at time step $k+1$ for S_4 on P_3, while the $(k-1)$-th data item is processed on P_2 sequentially for S_1, S_2, S_3, also in time step $k+1$. This partially sequential execution avoids all large communication costs. The communication time is $\frac{1}{10}$. In a model in which communication and computation can overlap, after the first item, this communication occurs while processors are computing. Finally, the period is $\mathcal{P} = 1$. Note that this improved period is obtained at the price of a higher latency: the latency has gone from 1.5 in the fully sequential execution to $\mathcal{L} = 1 + \frac{1}{10} + 1 = 2.1$ here.

This example illustrates the necessity of finding efficient trade-offs between antagonistic criteria.

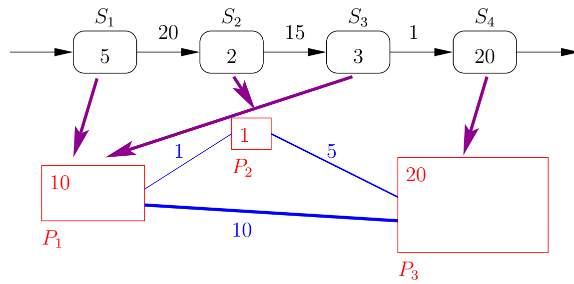
hal-00521712, version 1 - 28 Sep 2010



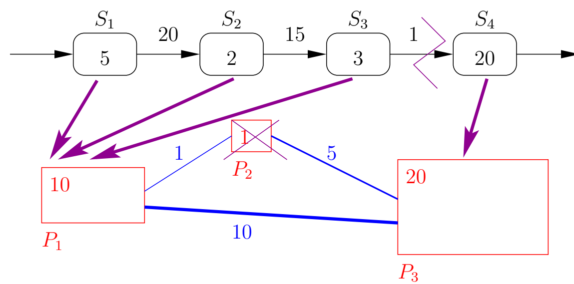
(a) Application and platform.



(b) Sequential execution on the fastest processor.



(c) Greedy execution using all processors.



(d) Resource selection to optimize period.

Fig. 1. Motivating example.

hal-00521712, version 1 - 28 Sep 2010

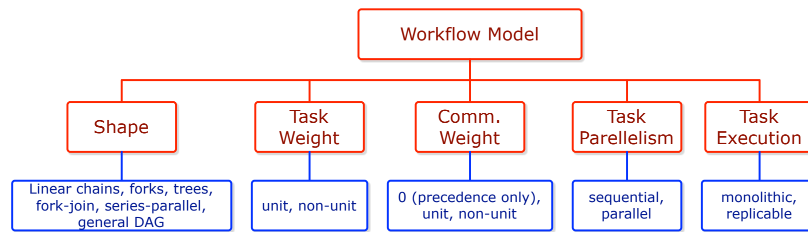


Fig. 2. The components of the Workflow Model.

3. MODELING TOOLS

This section gives general information on the scheduling problems. It should help the reader understand the key properties of pipeline applications.

All applications of pipelined scheduling are characterized by properties from three components that we call the *Workflow Model*, the *System Model* and the *Performance Model*. These components correspond to "which kind of program we are scheduling", "which parallel machine will host the program" and "what are we trying to optimize". This three-component view is similar to the three-field notation which is used to define classical scheduling problems [Bru07].

In the example of Section 2, the workflow model is the four-stage application with linear dependencies, computation costs, and communication costs; the system model is the three-processor platform with speeds and bandwidths; and the performance model corresponds to the two optimization criteria: latency and period.

We present in Sections 3.1, 3.2 and 3.3 the three models; then Section 3.4 classifies work in the taxonomy that has been detailed.

3.1 Workflow Model

The workflow model defines the program that is going to execute; its components are presented in Fig. 2.

Programs are usually represented as Directed Acyclic Graphs (or DAGs) in which nodes represent computation tasks, and edges represent communications between them. The shape of the graph is a parameter. Most program DAGs are not arbitrary but instead have some predefined form. For instance, it is common to find DAGs which are a single linear chain, as in the example of Section 2. Some other frequently encountered structures are fork graphs, trees, fork-join, and series-parallel graphs. Most of the time, the DAG is assumed to have a single node without parent node called the source and a single node without child node called the sink.

The weight of the tasks are important because they represent computational requirements. For some applications, all the tasks have the same computation requirement (they are said to be unit tasks). The weight of communications is defined similarly. Notice that a zero weight may be used to express a precedence between tasks with no communication.

The tasks of the program may contain parallelism. Although the standard model only uses sequential tasks, some applications feature parallel tasks. Three models of parallel tasks are commonly used (this naming was proposed by [FRS$^+$97] and is now commonly used in job scheduling for production systems): a *rigid* task requires

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

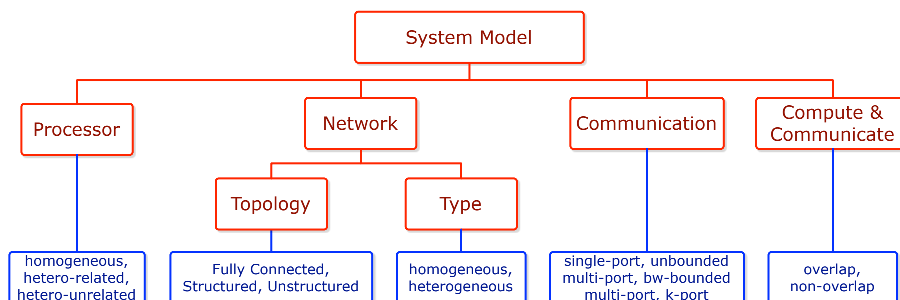


Fig. 3. The components of the System Model.

a given number of processors to execute; a *moldable* task can run on any number of processors, and its computation time is given by a speed up function (which can be arbitrary or can match a classical model such as the Amdahl's law [Amd67]); and a *malleable* task can change the number of processors it is executing on during its execution.

The task execution model indicates whether it is possible to execute concurrent replicas of a task at the same time or not. Replicating a task may not be possible due to an internal state of the task; the processing of the next data item depends upon the result of the computation of the current one. Such tasks are said to be *monolithic*; otherwise they are *replicable*. When a task is replicated, it is common to impose some constraints on the allocation of the data items to the replicas. For instance, the dealable stage rule [Col04] forces data items to be allocated in a round-robin fashion among the replicas. This constraint is enforced to avoid out-of-order completion and is quite useful when, say, a replicated task is followed by a monolithic one.

3.2 System Model

The system model describes the parallel machine used to run the program; its components are presented in Fig. 3 and are now described in more detail.

First, processors may be identical (*homogeneous*), or instead they can have different processing capabilities (*heterogeneous*). There are two common models of *heterogeneous* processors. Either their processing capabilities are linked by a constant factor, i.e., the processors have different speeds (known as the *related* model in scheduling theory), or they are not speed-related, which means that a processor may be fast on a task but slow on another one (known as the *unrelated* model in scheduling theory). Homogeneous and related processors are common in clusters. Unrelated processors arise when dealing with dedicated hardware or from preventing certain tasks to execute on some machines (to handle licensing issues or applications that do not fit in some machine's memory).

The network defines how the processors are interconnected. The topology of the network describes the presence and capacity of the interconnection links. It is common to find *fully connected networks* in the literature, which can model buses as well as Internet connectivity. *Arbitrary networks* which topologies are specified explicitly through an interconnection graph are also common. In between,

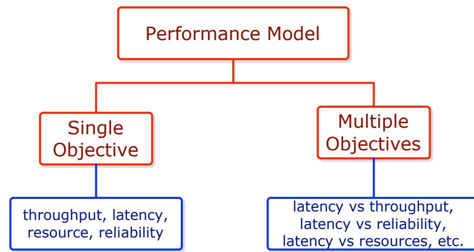


Fig. 4. The components of the Performance Model.

some systems may exhibit structured networks such as *chains*, *2D-meshes*, *3D-torus*, etc. Regardless of the connectivity of the network, links may be of different types. They can be homogeneous – transport the information in the same way – or they can have different speeds. The most common heterogeneous link model is the *bandwidth* model, in which a link is characterized by it sole bandwidth. There exist other communication models such as the *delay* model [RS87] which only consider latency with infinite bandwidth, or the *LogP*(**L**atency, **o**verhead, **g**ap and **P**rocessor) model [CKP+93], which is a realistic communication model. The latter two models are rarely used in pipelined scheduling.

Some assumptions must be made in order to define how communications take place. The *one-port* model [BRP99] forbids a processor to be involved in more than one communication at a time. This simple, but somewhat pessimistic, model is useful for representing single-threaded systems; it has been reported to accurately model certain MPI implementations that serialize the communication when the messages are larger than a few megabytes [SP04]. The opposite model is the *multi-port* model that allows a processor to be involved in an arbitrary number of communications simultaneously. This model is often considered to be unrealistic since some algorithms will use a large number of simultaneous communications which induce large overheads in practice. An in-between model is the *k-port* model where the number of simultaneous communications must be bounded by a parameter of the problem [HP03]. In any case, the model can also limit the total bandwidth that a node can use at a given time (that corresponds to the capacity of its network card).

Finally, some machines have hardware dedicated to communication or use multi-threading to handle communication; thus they can compute while using the network. This leads to an *overlap* of communication and computation. However, some machines or software libraries are still mono-threaded, so no such overlapping is possible.

3.3 Performance Model

The performance model describes the goal of the scheduler and tells from two valid schedules which one is better. Its components are presented in Fig. 4.

The most common objective in pipelined scheduling is to maximize the throughput of the system, which is the number of data processed by unit of time. In permanent applications such as interactive real time systems, it indicates the load that the system can handle.

hal-00521712, version 1 - 28 Sep 2010

Another common objective is to minimize the latency of the application, which is basically defined as the time taken by a single data item to be entirely processed. It measures the *response time* of the system to handle each data item. The maximum latency is most of the time the chosen objective since this value may depend on the chosen data item. Latency is mainly relevant in interactive systems. Note that latency minimization corresponds to *makespan* minimization in DAG scheduling, when there is a single data item to process.

Other objectives have also been studied. When the size of the computing system increase, hardware and software become more likely to be affected by malfunctions. Lots of formulation can model the problem (See [BBG+09] for details) but most of the time it reduces to optimizing the probability of correct execution of the application called reliability [GST09]. An other concern is the energy consumption of a computing system. It such models, processors' speed can be adjusted and the slower they are, the less energy they consume. Different model exist depending on the shape the energy cost function (usually quadratic or cubic in the speed) and on which speed values a processor can run at [BRGR10].

The advent of more complex systems and modern user requirements increased the interest in the optimization of several objectives at the same time. There are various ways to optimize multiple objectives [DRST09], but the most classical one is to optimize one of the objective while ensuring a given threshold value on the other ones. Deciding which objectives are constrained, and which one remains to optimize, makes no theoretical difference [TB07]. However, there is often an objective which is a more natural candidate for optimization when designing heuristics.

3.4 Placing Related Work in the Taxonomy

The problem of scheduling pipelined linear chains, with both monolithic and replicable tasks, on homogeneous or heterogeneous platforms, has been addressed in scheduling literature [LLP98; SV96; BR08; BR10]. [LLP98] proposes a three-step mapping methodology for maximizing the throughput of applications comprising a sequence of computation stages, each one consisting of a set of identical sequential tasks. [SV96] proposes a dynamic programming solution for optimizing latency under throughput constraints for applications composed of a linear chain of data-parallel tasks. [BR08] addresses the problem of mapping pipeline skeletons of linear chains of tasks on heterogeneous systems. [BR10] explores the theoretical complexity of the bi-criteria optimization of latency and throughput for chains and fork graphs of replicable and data-parallel tasks under the assumptions of linear clustering and round-robin processing of input data items.

Other works that address specific task graph topologies include [CNNS94], which proposes a scheme for the optimal processor assignment for pipelined computations of monolithic parallel tasks with series-parallel dependencies, and focuses on minimizing latency under throughput constraints. Also, [HM94] (extended in [CHM95]) discusses the throughput optimization for pipelined operator trees of query graphs that comprise sequential tasks.

Pipelined scheduling of arbitrary precedence task graphs of sequential monolithic tasks has been explored by a few researchers. In particular, [JV96] and [HO99] discuss heuristics for maximizing the throughput of directed acyclic task graphs on multiprocessor systems using point-to-point networks. [YKS03] presents an ap-

hal-00521712, version 1 - 28 Sep 2010

proach for resource optimization under throughput constraints. [SRM06] proposes an integrated approach to optimize throughput for task scheduling and scratch-pad memory allocation based on integer linear programming for multiprocessor system-on-chip architectures. [GRRL05] proposes a task mapping heuristic called EXPERT (EXploiting Pipeline Execution undeR Time constraints) that minimizes latency of streaming applications, while satisfying a given throughput constraint. EXPERT identifies maximal clusters of tasks that can form synchronous stages that meet the throughput constraint and maps tasks in each cluster to the same processor so as to reduce communication overhead and minimize latency.

Pipelined scheduling algorithms for arbitrary DAGs that target heterogeneous systems include the work of [Bey01], which presents the Filter Copy Pipeline (FCP) scheduling algorithm for optimizing latency and throughput of arbitrary application DAGs on heterogeneous resources. FCP computes the number of copies of each task that is necessary to meet the aggregate production rate of its predecessors and maps these copies to processors that yield their least completion time. Later on, [SFB+02] proposed Balanced Filter Copies which refine Filter Copy Pipeline. [BHCF95] and [RA01] address the problem of pipelined scheduling on heterogeneous systems. [RA01] uses clustering and task duplication to reduce the latency of the pipeline while ensuring a good throughput. However, these works target monolithic tasks, while [SFB+02] targets replicable tasks. Finally, [VCK+07] addresses the latency optimization under throughput constraints for arbitrary precedence task graphs of replicable tasks on homogeneous platforms.

An extensive set of papers dealing with pipelined scheduling is summed up in Table I. Each paper is listed with its characteristic. Since there are too many characteristics to present, we focus on the main ones: structure of the precedence constraints, type of computation, replication, performance metric, and communication model. The table is sorted according to the characteristic so that searching for papers close to a given problem is made easier. Different papers with the same characteristics are merged into a single line.

The structure of the precedence constraints (the *Str.* column) can be a single chain (C), a structured graph such as a tree or series parallel graph (S) or an arbitrary DAG (D). The processing units have computation capabilities (the *Comp.* column) which can be homogeneous (H), heterogeneous related (R) or heterogeneous unrelated (U). Replication of task (the *Rep.* column) can be authorized (Y) or not (N). The performance metric to compare the schedule (the *Metric* column) can be the throughput (T), the latency (L), the reliability (R), the energy consumption (E) or the number of processor used (N). The multi-objective problems are denoted with an & so that T&L denotes the bi-objective problem of optimizing both the throughput and the latency. Finally, the communication model (the *Comm.* column) can follow the precedence only model (P), the one-port model (1), the multi-port model (M), the k-port model (k), the delay model (D) or can be abstracted in the scheduling problem (abstr). When a paper deals with several scheduling model, the variation are denoted with a slash (/). For instance, the paper [BRSR08] deals with scheduling a chain (C) on either homogeneous or heterogeneous related processor (H/R) without using replication (N) to optimize the latency, the reliability or both of them (L/R/L&R) under the one-port model (1).

hal-00521712, version 1 - 28 Sep 2010

Table I. Papers about pipelined scheduling and the characteristics of the scheduling problems.

Reference	Str.	Comp.	Rep.	Metric	Comm.
[Bok88][HNC92] [Iqb92][MO95] [Nic94][PA04] [LLP98]	C	H	N	T	P
[Dev09]	C	H	N	T&L	P
[MCG+08]	C	H	Y	T	P
[BR08]	C	H/R	N	T	1
[ABDR09]	C	H/R	N	T/L/T&L	1/M
[ABR08]	C	H/R	N	T/L	M
[BRGR10]	C	H/R	N	T/L/E	1
[BRSR08]	C	H/R	N	L/R/L&R	1
[BR09]	C	H/R	Y/N	T/L/T&L	1
[BKRSR08][BKRSR09]	C	R	N	T&L	1
[BRT09]	C	R	N	L	1
[BRSR07]	C	R	N	T/L/T&L	1
[ABMR10]	C	R	N	T/L/T&L	1/M
[dNFJG05]	C	R	Y	T&N	M
[BGGR09]	C	R	Y	T	1/M
[KN10]	C	R	Y/N	T	M
[BR10]	C/S	H/R	Y/N	T/L&T	P
[HM94] [CHM95]	S	H	N	T	M
[CNNS94]	S	H	N	T&L	P
[JV96]	D	H	N	T	M
[HO99]	D	H	N	T&L	M
[GRRL05]	D	H	N	T&L	D
[KRC+99]	D	H	N	T&L	P
[VCK+07] [VCK+08] [VCK+10]	D	H	Y	T&L	M
[SV95]	D	H	Y/N	T	abstr
[SV96]	D	H	Y/N	T&L	abstr
[RA01]	D	H/U	N	T&L	M
[TC99]	D	R	N	T	M
[YKS03]	D	R	N	T&N	D
[Bey01][SFB+02]	D	R	Y	T	M
[BHCF95]	D	U	N	T	D
[SRM06]	D	U	N	T	M

4. FORMULATING THE SCHEDULING PROBLEM

The goal of this section is to build a mathematical formulation of the scheduling problem from a given application. It is a common practice to consider a more restrictive formulation than strictly necessary, in order to focus on more structured schedules which are likely to perform well.

We outline some principles in Section 4.1, and then we detail a few examples to illustrate the main techniques in Section 4.2. Finally we conclude in Section 4.3 by highlighting the known frontier between polynomial and NP-complete problems.

4.1 Compacting the Problem

One way to schedule a pipelined application is to explicitly schedule all the tasks of all the data items, amounts to completely unroll the execution graph and to assign a start-up time and a processor to each task. In order to ensure that all dependencies and resource constraints are fulfilled, it must be checked that all predecessor relations are satisfied by the schedule and that every processor does not execute more than one task at a given time. To do so, it may be necessary to associate a

hal-00521712, version 1 - 28 Sep 2010

start-up time to each communication, and a fraction of the bandwidth used (multi-port model). However, the number of tasks to schedule could be extremely large, making this approach intractable in practice.

To avoid this problem, a solution is to construct a more compact schedule, which hopefully has some interesting properties. The overall schedule should be easily deduced from the compact schedule in an incremental way. Checking whether the overall schedule is valid or not and computing the performance index (e.g., throughput, latency) should be easy operations. To make an analogy with compilation, this amounts to move from DAG scheduling to loop nest scheduling. Compiler techniques such as Lamport hyperplane vectors, or space-time unimodular transformations [Wol89; DRV00; KA02] allow to efficiently expose the parallelism while providing a linear or affine closed-form expression for scheduling each statement instance within each loop iteration.

The most common compact schedules are *cyclic* schedules. If a schedule has a period $\mathcal{P}$, then all computations and communications are repeated every $\mathcal{P}$ time units: two consecutive data items are processed exactly in the same way, with a shift of $\mathcal{P}$ time units. The cyclic schedule is constructed from the *elementary* schedule which gives the schedule of one data-item. If task t_i is executed on processor j at time s_i in the elementary schedule, then the execution of this task t_i for data item x will be executed at time $s_i + (x-1)\mathcal{P}$ on the same processor j in the cyclic schedule. This simple expression makes it easy to unravel the scheduling at run time.

With cyclic schedules, one data item starts its execution every $\mathcal{P}$ time units. Thus, the system has a throughput $\mathcal{T} = 1/\mathcal{P}$. However, the latency $\mathcal{L}$ of the application is harder to compute; in the general case, one must follow the entire processing of a given data item (but all data items have the same latency, which helps simplify the computation). The latency $\mathcal{L}$ is the length of the elementary schedule.

Checking the validity of a cyclic schedule is easier than that of an arbitrary schedule. Intuitively, it is sufficient to check the data items released in the last $\mathcal{L}$ units of time to make sure that a processor does not execute two tasks at the same time, or that a communication link is not used twice. Technically, we can build an operation list [ABDR09; ABMR10] which size is proportional to the original application precedence task graph and does not depend upon the number of data items that are processed.

A natural extension of cyclic schedules are *periodic* schedules, which repeat their operation every K data items. When $K = 1$, we retrieve cyclic schedules, but larger values of K are useful to gain performance (throughput increase) or to allow for replicated parallelism. For throughput increase, it is easy to give an example. Suppose that we have three different-speed processors P_1, P_2 and P_3 with speeds 1/3, 1/5 and 1/8, respectively. Within 120 time units, P_1 can process 40 data items, P_2 can process 24 and P_3 can process 15, resulting a periodic schedule with $K = 40+24+15 = 79$, and a throughput $\mathcal{T} = 79/120$. If the application has large inter-task communication costs, we cannot distribute the execution of a given data item among two or more processors. Hence the best choice for a cyclic schedule is to use only the fastest processor, for a throughput $\mathcal{T} = 1/3$, about half that

hal-00521712, version 1 - 28 Sep 2010

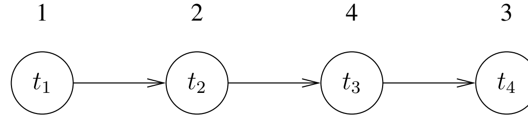


Fig. 5. An instance of the chain scheduling problem.

of the periodic schedule. Of course it is easy to generalize the example to derive an arbitrarily bad throughput ratio between cyclic and periodic schedules. As for replicated parallelism, the gain in throughput comes with a price: in some case, it becomes very difficult to compute the throughput, because there is no longer a critical resource that dictates the pace of operation for the entire system. Please refer to [BGGR09] for details.

Other common compact schedules consist in giving only the fraction of the time each processor spends executing each task [BLMR04; VCK$^+$08]. Such representations are more convenient when using linear programming tools. Reconstructing the actual schedule involves several concepts from graph theory and may be difficult to use in practice but can be done in polynomial time [BLMR04].

4.2 Examples

The goal of this section is to provide examples to help the reader understand how to build a schedule from the application and platform, as well as how the problem varies when basic assumptions are modified.

4.2.1 *Chain on Identical Processors with Interval Mapping.* We consider the problem of scheduling a linear chain of n monolithic tasks onto p identical processors, linked by an infinitely fast network. For $1 \leq i \leq n$, task t_i has a processing time p_i. Fig. 5 presents an instance of this scheduling problem with four tasks of processing times $p_1 = 1$, $p_2 = 2$, $p_3 = 4$, and $p_4 = 3$.

When scheduling chains of tasks, several mapping rules can be enforced:

—The *one-to-one mapping* rule ensures that each task is mapped to a different processor. This rule may be useful to deal with tasks having a high memory requirement, but all inter-task communications must then be paid.

—Another classical rule is the *interval mapping* rule, which ensures that if a processor executes tasks $t_{i_{begin}}$ and $t_{i_{end}}$, then all tasks t_i, with $i_{begin} < i < i_{end}$, are executed on the same processor. This rule, which provides an extension of one-to-one mappings, is often used to reduce the communication overhead of the schedule.

—Finally, the *general mapping* rule does not enforce any constraint, and thus any schedule is allowed. Note that for a homogeneous platform with communication costs, [ABR08] showed for the throughput objective that the optimal interval mapping is a 2-approximation of the optimal general mapping.

In this section, we are considering interval mappings. Therefore, a solution to the scheduling problem is a partition of the task set $\{t_1, \ldots, t_n\}$ into m sets or intervals $\{I_1, \ldots, I_m\}$, where I_j $(1 \leq j \leq m)$ is a set of consecutive tasks $(\forall i \in I_j, i' \in I_{j'}, \ j < j' \Rightarrow i < i')$, and $m \leq p$ (recall that there are p processors). The length of

hal-00521712, version 1 - 28 Sep 2010

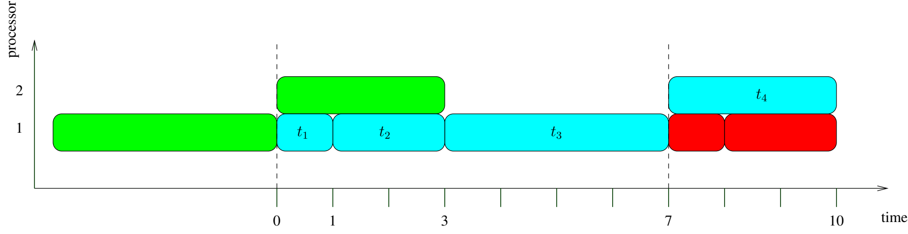


Fig. 6. The optimal solution to the instance of Fig. 5 using interval mapping on two processors.

an interval is defined as the sum of the processing time of its tasks: $L_j = \sum_{i \in I_j} p_i$, for $1 \leq j \leq m$. Note that all processors are identical (with unit speed), so that all mappings of intervals onto processors are identical.

In this case, the intervals are compact representations of the schedule. The *elementary* schedule represents the execution of a single data item: task t_i starts its execution at time $s_i = \sum_{i' < i} p_{i'}$ on the processor in charge of its interval. An overall schedule of period $\mathcal{P} = \max_{1 \leq j \leq m} L_j$ can now be constructed: task t_i is executed at time $s_i + (x-1)\mathcal{P}$ on the x-th data item. A solution of the instance of Fig. 5 on two processors which uses the intervals $\{t_1, t_2, t_3\}$ and $\{t_4\}$ is depicted in Fig. 6, where the boxes represent tasks and data items are identified by colors. The schedule is focused on the cyan data item (the labeled tasks) which follows the green one (partially depicted) and precedes the red one (partially depicted). Each task is periodically scheduled every 7 time units (a period is depicted with dotted lines). Processor 2 is idle during 4 time units for each period.

One can check that such a schedule is valid: the precedence constraints are respected, two tasks are never scheduled on the same processor at the same time (the processor in charge of interval I_j executes tasks for one single data item during L_j time units, and the next data item arrives after $\max_{j'} L_{j'}$ time units), and the monolithic constraint is also fulfilled, since all the instances of a task are scheduled on a unique processor.

To conclude, the throughput of the schedule is $\mathcal{T} = \frac{1}{\mathcal{P}} = \frac{1}{max_{1 \leq j \leq m} L_j}$, and its latency is $\mathcal{L} = \sum_{1 \leq i \leq n} p_i$. Note that given an interval mapping, it is not possible to achieve a better throughput since the machine for which $L_j = \max_{j'} L_{j'}$ will never be idle, and it is the one that defines the period. Note also that the latency is optimal over all schedules, since $\sum_i p_i$ is a lower bound on the latency.

For such a problem (no communication, identical processors, linear dependency graph, no replication, interval mapping), the problem of optimizing the throughput is reduced to the classical chain-on-chain problem [PA04], and it can be solved in polynomial time, using for instance a dynamic programming algorithm.

4.2.2 *Chain on Identical Processors with General Mapping.* This problem is a slight variation of the previous one: solutions are no longer restricted to interval mapping schedules, but any mapping may be used. By suppressing the interval mapping constraint, we can usually obtain a better throughput, but the scheduling problem and schedule reconstruction become harder, as we illustrate in the following example.

hal-00521712, version 1 - 28 Sep 2010

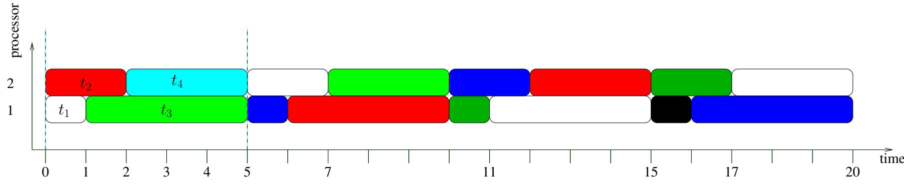


Fig. 7. The solution of optimal throughput to the instance of Fig. 5 using a general mapping on two processors.

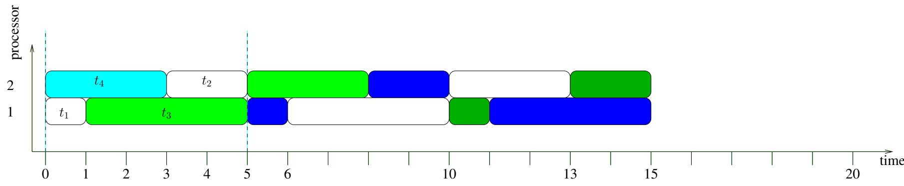


Fig. 8. A solution of same throughput than Fig. 7, but with better latency.

The solution of a general mapping can be expressed as a partition of the task set $\{t_1, \ldots, t_n\}$ into m sets $\{A_1, \ldots, A_m\}$, but these sets are not enforced to be intervals anymore. The optimal period is then $\mathcal{P} = \max_{1 \leq j \leq m} \sum_{i \in A_j} p_i$.

We present a generic way to reconstruct from the mapping a periodic schedule that preserves the throughput. A core schedule is constructed by scheduling all the tasks according to the allocation without leaving any idle time and, therefore, reaches the optimal period. Task t_i in set A_j is scheduled in the core schedule at time $s_i = \sum_{i' < i, i' \in A_j} p_{i'}$. A solution of the instance presented in Fig. 5 is depicted in Fig. 7 between the dotted lines (time units 0 to 5); it schedules tasks t_1 and t_3 on processor 1, and tasks t_2 and t_4 on processor 2.

The periodic schedule is built so that each task takes its predecessor from the previous period: inside a period, each task is processing a different data item. We can now follow the execution of the x-th data item: it starts being executed for task t_i at time $s_i + (i + x - 1)\mathcal{P}$, as illustrated for the white data item in Fig. 7. This technique produces schedules with large latency, between $(n-1)\mathcal{P}$ and $n\mathcal{P}$. In the example, the latency is 20, exactly 4 times the period.

The strict rule of splitting the execution in n periods ensures that no precedence constraint is violated. However, if the precedence constraint between task t_i and task t_{i+1} is respected in the core schedule, then it is possible to schedule both of them in a single time period. Consider the schedule depicted in Fig. 8. It uses the same allocation as the one in Fig. 7, but tasks t_2 and t_4 have been swapped in the core schedule. Thus, tasks t_1 and t_2 can be scheduled in the same period, leading to a latency of 13 instead of 20.

Note that the problem of finding the best general mapping for the throughput maximization problem is NP-complete: it is equivalent to the 2-PARTITION problem [GJ79] (consider an instance with two processors).

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

4.2.3 *Chain with a Fixed Processor Allocation.* In the previous examples, we have given hints of techniques to build the best core schedule, given a mapping and a processor allocation, in simple cases with no communication costs. In those examples, we were able to schedule tasks in order to reach the optimal throughput and/or latency.

Given a mapping and a processor allocation, obtaining a schedule that reaches the optimal latency can be done by greedily scheduling the tasks in the order of the chain. However, this may come at the price of a degradation of the throughput, since idle times may appear in the schedule. We can ensure that there will be no conflicts if the period equals the latency (only one data item in the pipeline at any time step).

If we are interested in minimizing the period, the presence of communications makes the problem much more difficult. In the model without overlap, it is actually NP-hard to decide the order of communications (i.e., decide the start time of each communication in the core schedule) in order to obtain the minimum period (see [ABMR10] for details). If computation and communication can be overlapped, the processor works simultaneously on various data sets, and we are able to build a conflict free schedule. When a bi-criteria objective function is considered, more difficulties arise, as the ordering of communications also becomes vital.

Even though the reconstruction technique has been illustrated only on a simple chain example, it can be applied in a similar way on a more complex DAG scheduling problem. We believe that the interest of interval mapping schedules of chains can be transposed to convex schedules on DAGs. Although such mapping rules were not considered in pipeline application, [BHCF95] build processor ordered schedules which is a property implied by convex clusters. In classical DAG scheduling problems convex clustering techniques have been developed before [PST05].

4.2.4 *Scheduling Moldable Tasks with Series-Parallel Precedence.* Chains are not the only kind of precedence constraints which are structured enough to help deriving interesting results. For instance, [CNNS94] considers the scheduling of series-parallel pipelined precedence task graphs, composed of moldable tasks. Series-parallel graphs are constructed recursively by expanding an edge of a series-parallel graph into several independent series-parallel graphs. Fig. 9 gives an example of such a graph. A given processor executes a single task and communications are assumed to be included in the parallel processing times.

Since a processor is only involved in the computation of a single task (note that in [CNNS94]'s model, there is no communication), each task can begin its execution as soon as all its predecessors have completed their executions or just after it finishes its execution on the previous data item. Therefore, one can build a periodic schedule which period is the length of the longest task by scheduling the ancestors of this task as late as possible and the successors as soon as possible. The obtained elementary schedule has no idle time on the longest path of the application task graph and the other paths only idle to wait for the longest path. Hence, the latency of the solution is the length of the longest path from the source to the sink, while the throughput is the inverse of the processing time of the longest task.

Since the application task graph is a series-parallel graph, the latency and throughput of a solution can be expressed according to its Binary Decomposition Tree

hal-00521712, version 1 - 28 Sep 2010

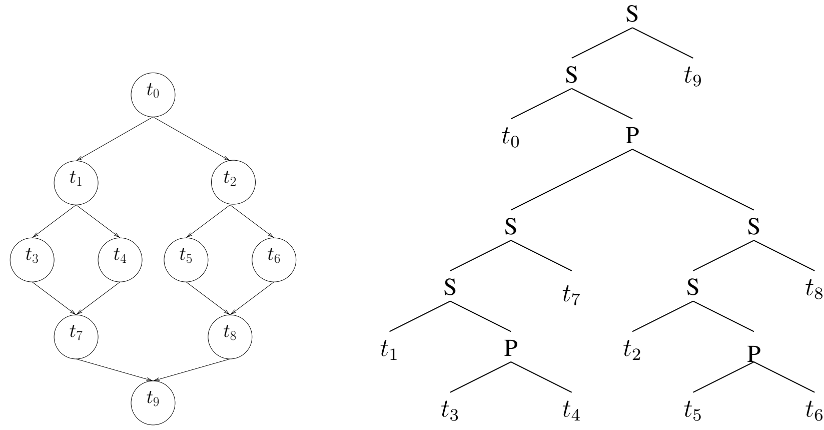


Fig. 9. A series-parallel graph, and its binary decomposition tree.

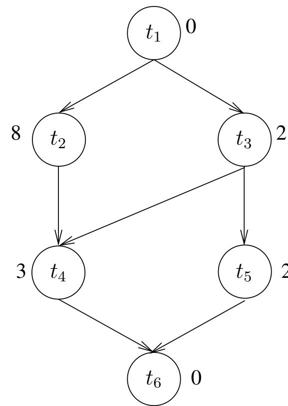


Fig. 10. An arbitrary DAG. (The processing requirement of each task is the label next to the task.)

(BDT) [VTL82] into series nodes S and parallel nodes P (see example in Fig. 9). In the BDT form, the throughput of a node is the minimum of the throughputs of the children of the node: $\mathcal{T}(S(l,r)) = \mathcal{T}(P(l,r)) = \min(\mathcal{T}(l), \mathcal{T}(r))$. The expression of the latency depends on the type of the considered node. If the node is a parallel node, then the latency is the maximum of the latency of its children: $\mathcal{L}(P(l,r)) = \max(\mathcal{L}(l), \mathcal{L}(r))$. If it is a serial node, the latency is the sum of the latency of its children: $\mathcal{L}(S(l,r)) = \mathcal{L}(l) + \mathcal{L}(r)$.

4.2.5 *Arbitrary DAGs on Homogeneous Processors.* Many applications cannot be represented by a structured graph such as a chain or a series-parallel graph. Arbitrary DAGs are more general but at the same time they are more difficult to schedule efficiently. Fig. 10 presents a sample arbitrary DAG.

Scheduling arbitrary DAGs poses problems which are similar to scheduling chains. Consider first the case of one-to-one mappings, in which each task is allocated to

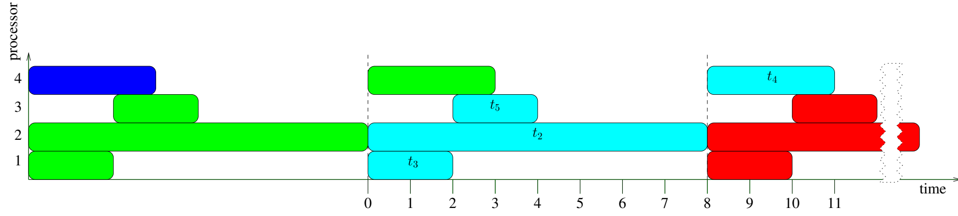


Fig. 11. One to one mapping of the instance of Fig. 10 with $\mathcal{L} = 11$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

a different processor. A periodic schedule is easily built by scheduling all tasks as soon as possible. Task i is scheduled in the periodic schedule on processor i at time $s_i = \max_{i' \in pred(i)} s_{i'} + p_{i'}$. This schedule can be executed periodically every $\mathcal{P} = \max_i p_i$ with throughput $\mathcal{T} = \frac{1}{\max_i p_i}$. The latency is the longest path in the graph $\mathcal{L} = \max s_i + p_i$. A schedule built in such a way does not schedule two tasks on the same processor at the same time since single task is executed in each period per processor and its processing time is smaller or equal to period. Under the one-to-one mapping constraint, this schedule is optimal for both objective functions. The solution of the graph presented in Fig. 10 is presented in Fig. 11, with a latency $\mathcal{L} = 11$ and a throughput $\mathcal{T} = \frac{1}{8}$.

When there is no constraint enforced on the mapping rule, problems similar to those of general mappings for linear chains appear (see Section 4.2.2): we cannot easily derive an efficient periodic schedule from the processor allocation. Establishing a periodic schedule that reaches the optimal throughput given a processor allocation is easy without communication cost, but it can lead to a large latency. Similarly to the case of chains, a core schedule is obtained by scheduling all the tasks consecutively without taking care of the dependencies. This way, we obtain the optimal period (for this allocation) equal to the load of the most loaded processor. The periodic schedule is built so that each task takes its data from the execution of its predecessors in the last period. Therefore, executing a data item takes as many periods as the depth of the precedence task graph. On the instance of Fig. 10, the optimal throughput on two processors is obtained by scheduling t_2 alone on a processor. Fig. 12 presents a periodic schedule of this processor allocation according to this generic technique, leading to a latency $\mathcal{L} = 15$. Note that t_5 could be scheduled in the same period as t_3 and in general this optimization can be done by a greedy algorithm. However, it does not guarantee to obtain the schedule with the optimal latency, which is presented in Fig. 13 and has a latency $\mathcal{L} = 11$. Indeed, contrarily to linear pipelines, given a processor allocation, obtaining the periodic schedule that minimizes the latency is NP-hard [RSBJ95].

4.2.6 *Scheduling Arbitrary DAGs on Homogeneous Processors with Replication.* A task is replicable if it does not contain an internal state. It means that the same task can be executed at the same time on different data items. On the instance presented in Fig. 10, only two processors can be useful: the dependencies prevent from executing any three tasks simultaneously, so a third processor would improve neither the throughput nor the latency for monolithic tasks. However, if task t_2 is

hal-00521712, version 1 - 28 Sep 2010

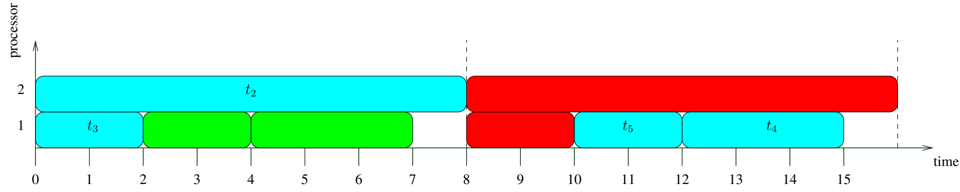


Fig. 12. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 15$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

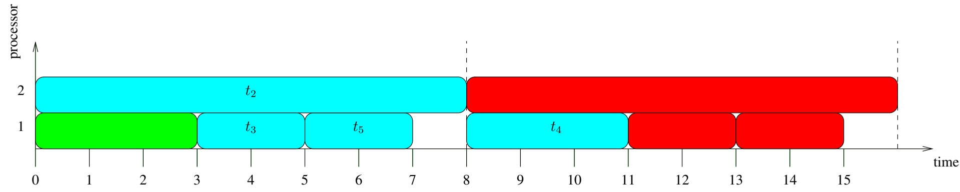


Fig. 13. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 11$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

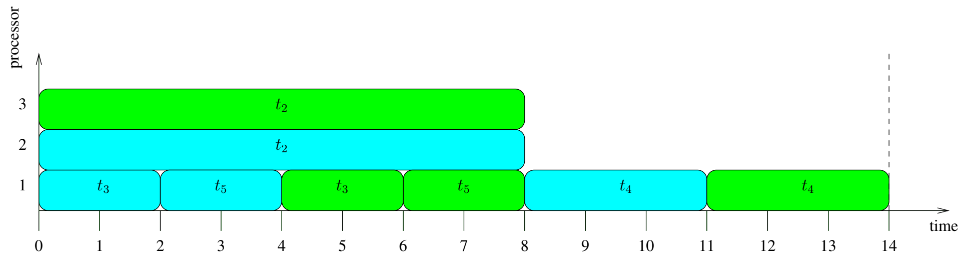


Fig. 14. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 14$ and $\mathcal{T} = \frac{1}{7}$ when task t_2 is replicable. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

replicable, the third processor could be used to replicate the computation of this task, therefore leading to the schedule depicted in Fig. 14.

Replicating t_2 leads to a schedule of period $\mathcal{P} = 14$ but which executes two data items per period. It obtains a throughput of $\mathcal{T} = \frac{2}{14} = \frac{1}{7}$, which is better than without replication. The latency is the maximum time a data item spends in the system. Without replication, all the data items spend the same time in the system. With replication, this statement no longer holds. In the example, the cyan data item spends 11 time units in the system whereas the green one spends 14. The latency of the schedule is then $\mathcal{L} = 14$. If t_4 was replicable as well, two copies could be executed in parallel, improving the throughput to $\mathcal{T} = \frac{2}{11}$ and the latency to $\mathcal{L} = 11$. A fourth processor could be used to pipeline the execution of t_4 and reach a period of $\mathcal{P} = 8$ and, hence, a throughput of $\mathcal{T} = \frac{1}{4}$.

A schedule with replication is no longer *cyclic* but instead is periodic, with the definitions of Section 4.1. Such a schedule can be seen as a pipelined execution of an unrolled version of the graph. The overall schedule should be specified by

hal-00521712, version 1 - 28 Sep 2010

giving a periodic schedule of length l (the time between the start of the first task of the first data item of the period and the completion of the last task of the last data item of the period) presenting how to execute K consecutive data items as well as its period $\mathcal{P}$. Verifying that the schedule is valid is done in the same way the verification is done for classical elementary schedules: one needs to expand all the periods that have a task running during the schedule, that is to say the ones that start during the elementary schedule and in the l time units before. Such a schedule has a throughput of $\mathcal{T} = \frac{K}{\mathcal{P}}$ and the latency should be computed as the maximum latency of the data items in the elementary schedule.

Note that if all tasks are replicable, the whole task graph can be replicated on all the m processors. Each processor executes sequentially exactly one copy of the application. This lead to a schedule of latency and period $\mathcal{P} = \mathcal{L} = \sum_i p_i$, and a throughput of $\mathcal{T} = \frac{\sum_i p_i}{m}$.

A fairly common constraint when dealing with replication is to forbid out-of-order execution. The point is that a monolithic task needs to process the data items in the right order to provide the correct result. This problem mainly appears when processors are heterogeneous. See the dealable stage constraint [Col04] for details.

4.2.7 *Model Variations.* In most cases, heterogeneity does not change drastically the scheduling model. However, the compact schedule description must then contain the processor allocation, i.e., say which task is executed onto which processor. Otherwise the formulations stay similar.

A technique to reduce latency is to consider *duplication* [AK98; VCK+08]. Duplicating a task consists in executing the same task more than once on different processors for every data item. Each task receives its data from one of the duplicates of each of its predecessors. Hence, this allows more flexibility for dealing with data dependency. The idea is to reduce the communication overheads at the expense of increasing the computation load. The major difference of duplication replication is, in duplication a single data item is executed in each period, whereas in replication, several data items can be executed in each period.

Communication models affect the schedule formulation. The easiest communication model is the one-port model where a machine communicates with a single one at a time. Therefore, in the schedule, each machine is represented by two processors, one for the computations and one for the communications. A valid schedule needs to "execute" a communication task at the same time on the communication processor of both machines involved in the data transfer. A common variation on the one-port model is to forbid communication and computation overlap. This model is used in [HO99]. In this case, there is no need for a communication processor; the communication tasks have to be scheduled on the computation processor [BRSR07].

To deal with more than one communication at a time, a realistic model would be to split the bandwidth equally among the communications. However such models are more complicated to analyze and are therefore not used in practice. Two ways of overcoming the problem exist. The first one is to consider the k-port model where each machine has a bandwidth B divided equally in k channels. The scheduling problem is then considered by using k communication processors per machine. This model has been used in [VCK+08].

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

When only the throughput matters (and not the latency), it is enough to ensure that no network link is overloaded. One can reconstruct a periodic schedule explicitly using the model detailed previously, considering each network link as a processor. This model has been used in [TC99].

4.3 Complexity

The goal of this section is to provide reference points for the complexity of the pipelined scheduling problem. Lots of works are dedicated to highlighting the frontier between polynomial and NP-hard optimization problems in pipelined scheduling.

The complexity of classical scheduling problems have been studied in [Bru07]. One of the main contributions was to determine some constraint changes that always make the problem harder. Some similar results are valid on pipelined scheduling. For instance, heterogeneous versions of problem are always harder than the homogeneous counterpart, since homogeneous cases can be easily represented as heterogeneous problem instances but not vice versa. Arbitrary task graph or architecture graph is always harder than the structured counterpart and in general considering a superset of graph makes problems harder. Also, removing communications makes the problem easier.

As seen in the previous examples, throughput optimization is always NP-hard for general mappings but polynomial instances can be found for interval mappings. The communication model plays a key role in complexity. The optimization of latency is usually equivalent to the optimization of the makespan in classical DAG scheduling [KA99b]. Multi-objective optimization problems are always more difficult than their single-objective counterparts [TB07].

The complexity of linear graph problems has been widely studied since it roots the general DAG case and most of the structured graph ones [BR08; BR10; BRSR07; ABR08; BRSR08; BRT09; BR09; ABMR10]. The large number of variants of those scheduling problems makes complexity issues very difficult. An exhaustive list of complexity results can be found in [Ben09]. We provide in Tables II and III a summary of complexity results for period and latency optimization problems, which hold for all communication models. *Fully Hom.* platforms refer to homogeneous computations and communications. *Comm. Hom.* platforms add one level of heterogeneity (heterogeneous related processors). Finally, *Hetero.* platforms are fully heterogeneous (*Comm. Hom.* with heterogeneous communications links).

For the period minimization problem, the reader can refer to [BR08] for the variant with no replication, and to [BR10] otherwise (results denoted with *(rep.)*). For the latency minimization problem, we report here results with no data-parallelism; otherwise the problem becomes NP-hard as soon as processors have different speed, with no communication costs [BR10].

5. SOLVING PROBLEMS

The goal of this section is to give methods to solve the pipelined scheduling problem using exact algorithms or heuristic techniques.

Table II. Summary of complexity results for period.

	Fully Hom.	***Comm. Hom.***	***Hetero.***
one-to-one	polynomial	polynomial, *NP-hard (rep.)*	NP-hard
interval	polynomial	NP-hard	NP-hard
general	NP-hard, *polynomial (rep.)*	NP-hard	

Table III. Summary of complexity results for latency.

	Fully Hom.	***Comm. Hom.***	***Hetero.***
one-to-one	polynomial [BR09]		NP-hard [BRSR08]
interval	polynomial [BR09]		NP-hard [BRT09]
general	polynomial [BRSR08]		

5.1 Scheduling a Chain of Tasks on Identical Processors with Interval Mappings

The first problem that we consider has been presented in Section 4.2.1. It consists in scheduling a chain of n tasks onto m processors without communication and enforcing the interval mapping constraint. Section 4.2.1 states that the latency of such schedules is constant, however the throughput can be optimized by minimizing the length of the longest interval.

The optimization of the throughput problem is the same combinatorial problem as the known chain-on-chain problem which has been solved by a polynomial algorithm in [Bok88], and then refined to reach lower complexity in [Iqb92; Nic94; MO95]. For very large problems, some heuristics have also been designed to reduce the scheduling times even further (see [PA04] for a survey). The first algorithm was based on a shortest path algorithm in an assignment graph. The approach below has a lower complexity, and is easier to understand.

The core of the technique is the Probe function that takes as a parameter the processing time of the tasks and the length of the longest interval $\mathcal{P}$. It constructs intervals $\{I_1, \ldots, I_m\}$ such that $\max_{1 \leq j \leq m} L_j \leq \mathcal{P}$, or shows that no such intervals exist (remember that $L_j = \sum_{i \in I_j} p_i$, where p_i is the processing time of task t_i). Probe recursively allocates the first x tasks of the chain to the first processor so that $\sum_{i \leq x} p_i \leq \mathcal{P}$ and $\sum_{i \leq x+1} p_i > \mathcal{P}$ until no task remains and returns the schedule. If the number of intervals is less than the number of processors, this function builds a schedule having no interval more than $\mathcal{P}$. Otherwise, no schedule of maximal interval length less than $\mathcal{P}$ exists with m processors. It can be easily shown that the schedules constructed are dominant, i.e., if a schedule exists then there is one respecting this construction.

The last problem is to choose the optimal value for the threshold $\mathcal{P}$. The optimal value is obtained by using a binary search on the possible values of $\mathcal{P}$, which tested using the Probe function. This construction is polynomial but has a quite high complexity. It is possible to reduce the complexity of the Probe function using prefix sum arrays and binary search so that fewer values of $\mathcal{P}$ can be tested by analyzing the processing time values. In the general case, the lowest complexity is reached by using Nicol's algorithm [Nic94] with Han's Probe function [HNC92] leading to a complexity of $O(n + m^2 \log(n) \log(n/m))$ (see [PA04] for details).

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

The same idea can be used to deal with different problems. For instance, with non-overlapping communication following the one-port model, the same idea applies. There may be inefficient cutting points (where cutting after or before is always better) which can be removed by scanning the chain once. Then the same algorithm may be used to solve the problem optimally, see [Iqb92] for additional details.

The same algorithm can also be used to solve optimally the case with related processor heterogeneity (processor speeds differ) if the order in which a data item goes through the processors is known. This is the case on dedicated hardware where the processor network forces the order of execution between the processors. However, if this order is not known, the problem is NP-complete in the strong sense [BRSR07], even without taking communication costs into account. Nevertheless, there are too many permutations to try, but the Probe algorithm sets a solid ground to build heuristics upon.

[BR08] proposes three heuristics to build interval mappings for optimizing the throughput on heterogeneous processors. The first one, called SPL, starts by assigning all the tasks to the fastest processor and then greedily splits the largest interval by unloading work to the fastest available processor. The splitting point is chosen so as to minimize the period of the new solution. The two other heuristics BSL and BSC use a binary search on the period of the solution. This period is used as a goal in the greedy allocation of the tasks to the processors. BSL allocates the beginning of the chain of tasks to the processor that will execute the most computations while respecting the threshold. On the other hand, BSC chooses the allocation which is the closest to the period.

[KN10] proposes an algorithm to schedule a chain using interval mappings on Grid computing systems (related processors, bounded multi-port, communication and computation overlapping, no replication), considering routing through intermediate nodes. The heuristic is based on the Dijkstra shortest path algorithm and their benchmark shows that its performance is within 30% of the general mapping optimal solution.

Solving the problem of optimizing both the latency and the throughput of a linear pipeline application has been considered in [SV96; BRSR07; BKRSR08]. Bi-objective optimization problems are usually solved by providing a set of efficient solutions. This set of solutions is generated by using an algorithm which targets values of one objective while optimizing the other one. The solution space is explored by executing this algorithm with different values of the threshold which cover the efficient part of the solution space.

[SV95] addresses the problem of scheduling a chain of tasks on homogeneous processors to optimize the throughput of the application without computation and communication overlapping. It covers a large scale of problems since it addresses moldable tasks with dedicated communication functions and replicable tasks. The network is supposed to be homogeneous but the details of the communication model are abstracted by explicitly giving the communication time in the instance of the problem. The technique used is based on dynamic programming and leads to a polynomial algorithm. This result can be extended by adding a latency dimension in the dynamic program to allow the optimization of the latency under throughput

constraint [SV96].

[BRSR07] propose heuristics that optimize the throughput and latency when link bandwidths are identical but processors have different speeds (one-port communications without overlap). Six heuristics are presented, enforcing a constraint on either the throughput or the latency. All six heuristics are similar to SPL, they start by allocating all the tasks to the fastest processor and split the interval iteratively. The differences are that each interval may be split in two to the fastest available processor or split in three to the two fastest processors available. The other differences are about the solution chosen; it could be the one that maximizes one objective or a ratio of improvement. [BKRSR08] propose an integer linear programming formulation to solve the problem optimally (and with heterogeneous bandwidths). The solving procedure takes a long time even on a simple instance of 7 tasks and 10 processors (a few hours on a modern computer) but allows to assess the absolute performance of the previously proposed heuristics.

5.2 Solving Chains with General Mappings

Using general mappings instead of restricting to interval mappings leads to better throughput. Without replication or communication, the optimization of the throughput on homogeneous processors is NP-complete by reduction to 3-PARTITION. In fact, the mathematical problem is to partition n integers $p_1, \ldots, p_n$ into m sets $A_1, \ldots, A_m$ so that the length of the largest set $\max_j \sum_{i \in A_j} p_i$ is minimized. This mathematical formulation is the same as scheduling independent tasks on identical processors to minimize the makespan which has been studied for a long time.

On homogeneous processors, the classical List Scheduling algorithm schedules tasks greedily on the least loaded processor, and it is a 2-approximation [Gra66], i.e., the value of the obtained solution is at most twice the optimal value. Sorting tasks by non increasing processing times leads to the Largest Processing Time (LPT) algorithm which is known to be a 4/3-approximation algorithm [Gra69]. An approximation scheme (i.e., approximation algorithm with arbitrary precision) based on binary search and dynamic programming has been proposed in [HS87].

When processors become heterogeneous, the link with the classical makespan optimization problem stays valid. If processors are uniform (they compute at different speeds), the throughput optimization problem is the same as scheduling independent tasks on uniform processors. This problem admits a 2-approximation algorithm similar to LPT [GIS77]. [HS88] provides an elaborate approximation scheme with very high runtime complexity as well as a simple 3/2-approximation algorithm. If processors are unrelated (i.e., their speeds depend on the task they are handling), the throughput optimization problem is the same as scheduling independent tasks on unrelated processors to minimize the makespan. It can be shown than there exists no approximation algorithm with a ratio better than 3/2 [LST90]. Moreover, a 2-approximation algorithm based on binary search and linear programming is known [LST90].

The results on classical scheduling problems stays valid even if the graph is not linear as long as the performance index is the throughput and communication occur. However, it still provides an interesting baseline to study the impact of communications.

[KN10] considers the problem of scheduling a chain on a grid computer with

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

routing to optimize the throughput. Processors are heterogeneous (related) and communications follow the bounded multi-port model with overlapping. They first consider the case with replication (called multi-path in their terminology). This case is solved optimally in polynomial time by a flow-based linear programming formulation (somehow similar to LPsched [dNFJG05]). They also consider the case without replication (single-path). The problem becomes NP-complete, but they still provide a Integer Linear Program to solve it optimally. They also propose a polynomial heuristic based on the Dijkstra shortest path algorithm that only construct interval mappings. Their experiments show that the heuristic is within 30% of the Integer Linear Programming solution.

[BRT09] provides a polynomial algorithm to optimize the latency of a pipelined chain on heterogeneous (related) network of processor under the one-port model. The algorithm is based on a dynamic programming formulation.

5.3 Structured Application Graphs

We show in this section how to solve the problem of scheduling pipelined series-parallel graphs of moldable tasks. This problem has been presented in Section 4.2.4.

[CNNS94] optimizes both the latency and the throughput of the solution by optimizing the latency under throughput constraint. They are optimizing the latency by computing, for each node of the binary decomposition tree, the optimal latency achievable for all number of processors using dynamic programming: $\mathcal{L}(S(l,r,m)) = \min_j \mathcal{L}(l,j) + \mathcal{L}(r,m-j)$ and $\mathcal{L}(P(l,r,m)) = \min_j \max(\mathcal{L}(l,j), \mathcal{L}(r,m-j))$. The throughput constraint is ensured by setting the latency of the leaves to infinity on processor allocations that would not respect the throughput constraint.

Evaluating the latency of a node for a given number of processors require $O(m)$ computations and there are $2n-1 \in O(n)$ nodes to estimate in the tree for m different values of the number of processors. The overall complexity of the algorithm is $O(nm^2)$. As it is usual with dynamic programming, evaluating the recursive function only gives the value of the minimal latency in schedule of throughput respecting the constraint but the allocation can be deduced from the table in linear time.

[CNNS94] contains some techniques to reduce the time complexity on some cases by exploiting the convexity of the processing time of the Series operator. They also explain how to solve the problem of optimizing the throughput under latency constraint efficiently.

[HM94] and its refinement [CHM95] are interested in optimizing the throughput of pipelined trees for databases applications. They are considering homogeneous processors, no communication overlap, and the bandwidth bounded multi-port model. Therefore the load of a processor is the sum of the weights of the nodes executed by this processor plus the weights of the edges to other processors. Since latency is not a concern here, there is no fine grain scheduling of the instruction but only a flow-like solution where each processor has a large buffer of tasks to execute. The main contribution of [HM94] is the definition of a monotone tree which is a modified version of a tree where two nodes linked by too high communication edge are merged. They show that such a modification is optimal.

[CHM95] presents two approximation algorithms for the previous problem. Both are based on a two-phase decomposition: first the tree is decomposed into a forest

hal-00521712, version 1 - 28 Sep 2010

by removing some edges; then the trees are allocated to processors using LPT. Removing an edge incurs communication costs to both extremities of the edge. It is shown that if the obtained forest does not have too large trees and the load is kept reasonable, then LPT will generate an approximation of the optimal solution. Two tree decomposition algorithms follow. The first one is a simple greedy algorithm of approximation ratio 3.56, the second one is a more complex greedy algorithm of approximation ratio 2.87.

5.4 Scheduling with Replication

[SV95] addresses the problem of scheduling a chain of moldable tasks to optimize the throughput using replicated interval mappings: if a task is replicated, its whole interval is replicated too. The algorithm uses dynamic programming to find the intervals I, the number of processors per interval m_{int} and the number of replications r_{int} of the interval which minimizes the throughput $\mathcal{T} = \max_{int \in I} \frac{p(int, m_{int})}{r_{int}}$. This information does not give the periodic schedule that the system should follow. It just states where the tasks should be executed and a Demand Driven middleware will execute them correctly. Building a periodic schedule reaching the same throughput from the intervals and the number of time they should be replicated is possible. However, one needs to specify the execution of the Least Common Multiple of the number of replication $LCM_{int}(r_{int})$ data items to obtain a periodic schedule. Indeed, if one task is replicated two times and another one is replicated three times, the execution of six data items must be unrolled for the schedule to be periodic.

[SV96] adds the computation of the latency to [SV95]. Since the graph is a chain and all the intervals are executed independently, it is possible to build a schedule that reaches the optimal latency for a given processor allocation $\mathcal{L} = \sum_{int \in I} p(int, m_{int})$. The interval which constrains the throughput must be executed without idle time, the preceding tasks are scheduled as late as possible and the following tasks are scheduled as soon as possible. The optimization of the latency under throughput constraint is obtained using a dynamic programming algorithm, by forbidding the numbers of processors and numbers of replications for each interval that violates the throughput constraint.

5.5 General Method to Optimize the Throughput

[BHCF95] deals with executing a signal processing application on heterogeneous machines, where not all tasks can be executed on all type of processors. They schedule a precedence task graph of sequential monolithic tasks. Communications overlap and follow the bandwidth bounded multi-port model with latency. First they build a schedule using clustering to reduce the coarsen the graph. Then they apply an exponential algorithm that finds the optimal processor ordered schedule. Finally they improve iteratively the clustering. Having a processor ordered schedule means that the graph of the communications between the processors is acyclic and the authors claim that it helps ensuring the precedence constraints are respected.

[Bey01] deals with scheduling pipelined task graphs on the grid. The resources of the grid are exploited using replication. He propose the Filter Copy Pipeline (FCP) algorithm. FCP considers the application graph in a topological order and chooses the number of copies for each task so that it can process the data it receives without

getting a large backlog. In other words, if the predecessor of a task handles x data items per time unit, FCP replicates this task to handle x data items per time unit. Those replicates are allocated to processors using the earliest completion time rule. Later on, [SFB$^+$02] propose Balanced Filter Copies that allocates a processor to a single task and keeps tracks of the network consumption while computing the schedule.

[TC99] is concerned with scheduling a pipelined task graph on a heterogeneous network of heterogeneous processors with computation and communication overlap. Since the authors are only interested in the throughput, a solution to the problem reduces to a mapping of the tasks to the processors and the throughput of the solution is given by the most loaded processor or link. The algorithm starts by ordering the tasks in depth-first traversal of a clustering tree. Then the tasks are mapped to the processors using the following algorithm. The processors are, the one after the other, loaded with the first unallocated tasks that minimize the maximum of three quantities: the current load, the perfect load balance of the unallocated tasks on the unallocated processors, and the yet to be decided communications to the current processor from the unallocated tasks. Finally, the obtained schedule is iteratively improved by unscheduling some of the tasks on the most loaded processors and links and scheduling them again.

[YKS03] deals with scheduling arbitrary precedence task graphs on a Network of Workstation (NOW). The processors are heterogeneous (related) and allow for communication and computation overlap. It assumes a linear communication model without contention. Two objectives are optimized: the throughput and number of machines used from the NOW. The throughput is given by the user and then, the execution is cut in stages whose lengths is given by the throughput. A processor used in one stage is not reused in the next one so that the throughput can be guaranteed. The tasks are allocated using earliest time first heuristic. The authors propose then some techniques to compact the schedule, reducing the number of processors used.

5.6 General Method to Optimize Throughput and Latency

[GRRL05] is interested in scheduling a pipelined precedence task graph on a homogeneous cluster with communication and computation overlap to optimize both latency and throughput. The network is assumed to be completely connected and the delay model is used. The delay is computed using a latency plus bandwidth model, but no link congestion is considered.

The authors propose the EXPERT algorithm that optimizes the latency under a throughput constraint. Given a throughput goal $\mathcal{T} = 1/\mathcal{P}$, all tasks are partitioned in stages such that task t is allocated to the minimum number of stages k such that $topLevel(t) \leq k \times \mathcal{P}$. Then all the paths of the graph are considered in decreasing order of length including communication delays, and tasks of the same stage are clustered greedily as long as the cluster is smaller than $\mathcal{P}$. Finally, inter-stage clusters can be merged as long as the length of the resulting cluster is less than $\mathcal{P}$. Communication between the clusters are grouped at the end of the execution of the cluster.

[HO99] deals with arbitrary application graphs and homogeneous processors and network. The technique is designed for hypercube networks but can be adapted to

hal-00521712, version 1 - 28 Sep 2010

arbitrary networks. It assumes communication and computation can overlap. They are interested in optimizing both latency and throughput. The algorithm proposed only provides a processor allocation and the periodic schedule is reconstructed using a technique similar to the one presented in Section 4.2.5: in a period, the tasks are ordered in topological order. If a task does not get its precedences in the current iteration, it takes them from the previous iteration.

Given a targeted period, and therefore throughput, the proposed method has three steps. It first clusters the task in order of non-increasing communication requirement and keeps the size of the clusters less than the targeted period. Then the clusters are mapped to computation nodes to minimize the amount of communication. This is done by mapping the tasks randomly to the nodes. Then the processor set is cut in two equals parts and tasks are exchanged to decrease the communication on the processor cut. The communications in each part are then optimized recursively. Finally, the solution is improved iteratively by moving tasks between processors to decrease the load of the most loaded link.

[VCK+08] is dealing with optimizing the latency and throughput of arbitrary DAGs on homogeneous processors linked by a network of different bandwidth with communication/computation overlapping and using replication and duplication.

The algorithm is in three steps and takes a throughput constraint as a parameter. The first step generates clusters to match the throughput constraint. It considers the replication of tasks to deal with computational bottlenecks, and duplication of tasks and clustering to decrease communication bottlenecks. In a second step, it reduces the number of clusters to the number of processors in the system by merging clusters to minimize processor idle times. Finally, the latency is minimized by considering for each task of the critical path its duplication and clustering to its predecessor cluster or successor cluster.

6. CONCLUSION AND FUTURE WORK

In this survey, we presented an overview of pipelined workflow scheduling, a problem that asks for an efficient execution of a streaming application that operates on a set of consecutive data items. We described the components of application and platforms models, and how a scheduling problem can be formulated for a given application. We presented a brief summary of the solution methods for specific problems, highlighting the frontier between polynomial and NP-hard optimization problems.

Although there is a significant body of literature for this complex problem, realistic application scenarios still call for more work in the area, both theoretical and practical.

When developing solutions for realistic applications, one has to consider all the ingredients of the schedule as a whole, including detailed communication models and memory requirements (especially when more than one data item is processed in a single period). Such additional constraints make the development of efficient scheduling methods even more difficult.

As the literature shows, having structure either in the application graph or in the execution platform graph dramatically helps for developing effective solutions. We think that extending this concept to the schedule could be useful too. For

hal-00521712, version 1 - 28 Sep 2010

hal-00521712, version 1 - 28 Sep 2010

example, for scheduling arbitrary DAGs, developing structured schedules, such as convex schedules, has a potential for yielding new results in this area.

Finally, as the domain evolves, new optimization criteria must be introduced. In this paper, we have mainly dealt with throughput and latency. Other performance-related objectives arise with the advent of very large-scale platforms, such as increasing the reliability of the schedule (e.g., through task duplication). Environmental and economic criteria, such as the energy dissipated throughout the execution, or the rental cost of the platform, are also likely to play an increasing role. Altogether, achieving a reasonable trade-off between all these multiple and antagonistic objectives, will prove a very interesting algorithmic challenge.

ACKNOWLEDGMENTS

This work was supported in parts by the DOE grant DE-FC02-06ER2775; by AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3; by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802, and by the French ANR StochaGrid project. Anne Benoit and Yves Robert are with the Institut Universitaire de France.

REFERENCES

[ABDR09] Kunal Agrawal, Anne Benoit, Fanny Dufossé, and Yves Robert. Mapping filtering streaming applications with communication costs. In *SPAA'2009, the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, Canada, 2009. ACM Press.

[ABMR10] Kunal Agrawal, Anne Benoit, Loic Magnan, and Yves Robert. Scheduling algorithms for linear workflow optimization. In *IPDPS'2010, the 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010. IEEE Computer Society Press.

[ABR08] Kunal Agrawal, Anne Benoit, and Yves Robert. Mapping linear workflows with computation/communication overlap. In *ICPADS'2008, the 14th IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, December 2008. IEEE.

[AK98] I. Ahmad and Y. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transaction on Parallel Distributed Systems*, 9(9):872–892, September 1998.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'1967, the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, April 1967. ACM.

[BBG+09] Xavier Besseron, Slim Bouguerra, Thierry Gautier, Erik Saule, and Denis Trystram. *Fault tolerance and availability awarness in computational grids*, chapter 5. Fundamentals of Grid Computing. Chapman and Hall/CRC Press, December 2009. ISBN: 978-1439803677.

[Ben09] Anne Benoit. Scheduling pipelined applications: models, algorithms and complexity, July 2009. Habilitation à diriger des recherches, École normale supérieure de Lyon.

[Bey01] M. Beynon. *Supporting Data Intensive Applications in a Heterogeneous Environment*. PhD thesis, University of Maryland, 2001.

[BGGR09] Anne Benoit, Bruno Gaujal, Matthieu Gallet, and Yves Robert. Computing the throughput of replicated workflows on heterogeneous platforms. In *ICPP'2009, the 38th International Conference on Parallel Processing*, Vienna, Austria, 2009. IEEE Computer Society Press.

[BHCF95] Sati Banerjee, Takeo Hamada, Paul M. Chau, and Ronald D. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Transactions on Signal Processing*, 43(6):1468–1484, June 1995.

[BKC+01] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[BKRSR08] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Bi-criteria pipeline mappings for parallel image processing. In *ICCS'2008, the 8th International Conference on Computational Science*, Krakow, Poland, June 2008. LNCS Springer.

[BKRSR09] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria Scheduling of Pipeline Workflows (and Application to the JPEG Encoder). *International Journal of High Performance Computing Applications*, 2009.

[BLMR04] 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 *ISPDC'04, the 3rd International Symposium on Parallel and Distributed Computing/3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 296–302, Washington, DC, USA, 2004. IEEE Computer Society.

[BML+06] Shawn Bowers, Timothy M. McPhillips, Bertram Ludäscher, Shirley Cohen, and Susan B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW)*, pages 133–147, 2006.

[Bok88] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transaction on Computers*, 37(1):48–57, 1988.

[BR08] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal on Parallel and Distributed Computing*, 68(6):790–808, 2008.

[BR09] Anne Benoit and Yves Robert. Multi-criteria mapping techniques for pipeline workflows on heterogeneous platforms. In George A. Gravvanis, John P. Morrison, Hamid R. Arabnia, and David A. Power, editors, *Recent developments in Grid Technology and Applications*, pages 65–99. Nova Science Publishers, 2009.

[BR10] Anne Benoit and Yves Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 57(4):689–724, August 2010.

[BRGR10] Anne Benoit, Paul Renaud-Goud, and Yves Robert. Performance and energy optimization of concurrent pipelined applications. In *IPDPS'2010, the 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010. IEEE Computer Society Press.

[BRP99] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'1999, the 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[BRSR07] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria scheduling of pipeline workflows. In *HeteroPar'07, the 6th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Austin, Texas, USA, June 2007.

[BRSR08] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Optimizing latency and reliability of pipeline workflow applications. In *HCW'08, the 17th International Heterogeneity in Computing Workshop*, Miami, USA, April 2008. IEEE.

[BRT09] Anne Benoit, Yves Robert, and E. Thierry. On the complexity of mapping linear chain applications onto heterogeneous platforms. *PPL*, 19(3):383–397, March 2009.

[Bru07] Peter Brucker. *Scheduling Algorithms.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, fifth edition edition, 2007.

[CHM95] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling problems in parallel query optimization. In *PODS'1995, the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 255–265, New York, NY, USA, 1995. ACM Press.

[CkLW+00] Alok Choudhary, Wei keng Liao, Donald Weiner, Pramod Varshney, Richard Linderman, Mark Linderman, and Russell Brown. Design, implementation and evaluation of

hal-00521712, version 1 - 28 Sep 2010

parallel pipelined stap on parallel computers. *IEEE Transactions on Aerospace and Electronic Systems*, 36(2):655 – 662, April 2000.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *PPOPP'1993, the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM.

[CNNS94] Alok Choudhary, Bhagirath Narahari, David Nicol, and Rahul Simha. Optimal processor alignment for a class of pipeline computations. *IEEE Transaction on Parallel Distributed Systems*, 5(4):439–443, April 1994.

[Col04] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[CT02] Condor-Team. The directed acyclic graph manager (DAGMan), 2002. `http://www.cs.wisc.edu/condor/dagman`.

[DBGK03] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. *Grid Resource Management*, chapter Workflow management in GriPhyN. Springer, 2003.

[Dev09] UmaMaheswari C. Devi. Scheduling recurrent precedence-constrained task graphs on a symmetric shared-memory multiprocessor. In Springer, editor, *Euro-Par 2009 Parallel Processing*, pages 265–280, August 2009.

[dNFJG05] Luiz Thomaz do Nascimento, Renato A. Ferreira, Wagner Meira Jr., and Dorgival Guedes. Scheduling data flow applications using linear programming. *ICPP'2005, the 34th International Conference on Parallel Processing*, 0:638–645, 2005.

[DRST09] Pierre-Francois Dutot, Krzyztof Rzadca, Erik Saule, and Denis Trystram. Multi-objective scheduling. In Yves Robert and Frederic Vivien, editors, *Introduction to Scheduling.* CRC Press, November 2009.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhauser, 2000.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, LNCS, pages 1–34. Springer, 1997.

[GIS77] T. Gonzalez, O.H. Ibarra, and S. Sahni. Bounds for LPT schedules on uniform processors. *SIAM Journal on Computing*, 6:155–166, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[GRRL05] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Optimizing latency under throughput requirements for streaming applications on cluster execution. In *Cluster Computing, 2005. IEEE International*, pages 1–10, September 2005.

[GST09] Alain Girault, Erik Saule, and Denis Trystram. Reliability versus performance for critical applications. *Journal on Parallel and Distributed Computing*, 69(3):326–336, 2009.

[HC09] T. D. R. Hartley and U. V. Catalyurek. A component-based framework for the cell broadband engine. In *Proc. of 23rd Int'l. Parallel and Distributed Processing Symposium, The 18th Heterogeneous Computing Workshop (HCW 2009)*, May 2009.

[HCR+08] T. D. R. Hartley, U. V. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *Proc. of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

[HFB+09] T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Ozguner, and U. V. Catalyurek. Investigating the use of gpu-accelerated nodes for sar image formation. In *Proc. of the IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

hal-00521712, version 1 - 28 Sep 2010

[HM94] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *VLDB*, pages 36–47, 1994.

[HNC92] Y. Han, B. Narahari, and H.-A. Choi. Mapping a chain task to chained processors. *Information Processing Letter*, 44:141–148, 1992.

[HO99] Stephen L. Hary and F. Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transaction on Parallel Distributed Systems*, 10(8):838–851, 1999.

[HP03] B. Hong and V.K. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *ICPP'2003, the 32th International Conference on Parallel Processing*. IEEE Computer Society Press, 2003.

[HS87] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of ACM*, 34:144–162, 1987.

[HS88] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539 – 551, 1988.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys'2007, the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[Iqb92] Mohammad Ashraf Iqbal. Approximate algorithms for partitioning problems. *International Journal of Parallel Programming*, 20(5):341–361, 1992.

[JV96] J. Jonsson and J. Vasell. Real-time scheduling for pipelined execution of data flow graphs on a realistic multiprocessor architecture. In *ICASSP-96: Proc. of the 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3314–3317, 1996.

[KA99a] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999.

[KA99b] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[KA02] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2002.

[KGS04] Jihie Kim, Yolanda Gil, and Marc Spraragen. A knowledge-based approach to interactive workflow composition. In *14th International Conference on Automatic Planning and Scheduling (ICAPS 04)*, 2004.

[KN10] Ekasit Kijsipongse and Sudsanguan Ngamsuriyaroj. Placing pipeline stages on a grid: Single path and multipath pipeline execution. *Future Generation Computer Systems*, 26(1):50 – 62, 2010.

[KRC+99] Kathleen Knobe, James M. Rehg, Arun Chauhan, Rishiyur S. Nikhil, and Umakishore Ramachandran. Scheduling constrained dynamic applications on clusters. In *Supercomputing'1999, the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, New York, NY, USA, 1999. ACM.

[LLP98] Myungho Lee, Wenheng Liu, and Viktor K. Prasanna. A mapping methodology for designing software task pipelines for embedded signal processing. In *Proc. of the Workshop on Embedded HPC Systems and Applications of IPPS/SPDP*, pages 937–944, 1998.

[LST90] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[MCG+08] A. Moreno, E. Csar, A. Guevara, J. Sorribesand T. Margalef, and E. Luque. Dynamic pipeline mapping (dpm). In Springer, editor, *Euro-Par 2008 Parallel Processing*, pages 295–304, August 2008.

[MGPD+08] Allan MacKenzie-Graham, Arash Payan, Ivo D. Dinov, John D. Van Horn, and Arthur W. Toga. Neuroimaging data provenance using the loni pipeline workflow environment. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW)*, pages 208–220, 2008.

hal-00521712, version 1 - 28 Sep 2010

[Mic09] Microsoft. AXUM webpage. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx, 2009.

[MO95] Fredrik Manne and Bjrn Olstad. Efficient partitioning of sequences. *IEEE Transaction on Computers*, 44(11):1322–1326, 1995.

[Nic94] David Nicol. Rectilinear partitioning of irregular data parallel computations. *Journal on Parallel and Distributed Computing*, 23:119–134, 1994.

[OGA+06] Thomas Oinn, Mark Greenwood, Matthew Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

[PA04] Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal on Parallel and Distributed Computing*, 64(8):974–996, 2004.

[PST05] J. E. Pecero-Sanchez and Denis Trystram. A new genetic convex clustering algorithm for parallel time minimization with large communication delays. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *PARCO*, volume 33 of *John von Neumann Institute for Computing Series*, pages 709–716. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[RA01] Samantha Ranaweera and Dharma P. Agrawal. Scheduling of periodic time critical applications for pipelined execution on heterogeneous systems. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 131–140, Washington, DC, USA, 2001. IEEE Computer Society.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O' Reilly, 2007.

[RKO+03] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo. The discovery net system for high throughput bioinformatics. *Bioinformatics*, 19(Suppl 1):i225–31, 2003.

[RS87] V. J. Rayward-Smith. UET scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[RSBJ95] V. Rayward-Smith, F.N. Burton, and G.J Janacek. *Scheduling Theory and its Applications*, chapter 7 - Scheduling Parallel Program Assuming Preallocation, pages 146–165. Wiley, 1995.

[SFB+02] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Supercomputing'2002, the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[SKS+09] O. Sertel, J. Kong, H. Shimada, U. V. Catalyurek, J. H. Saltz, and M. N. Gurcan. Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development. *Pattern Recognition*, 42(6):1093–1103, 2009.

[SP04] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Euro-Par 2004 Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[SRM06] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *CASES '06: ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2006.

[SV95] Jaspal Subhlok and Gary Vondran. Optimal mapping of sequences of data parallel tasks. In *PPOPP'1995, the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 134–143, New York, NY, USA, 1995. ACM.

[SV96] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *SPAA'1996, the 8th annual ACM symposium on Parallel algorithms and architectures*, pages 62–71, New York, NY, USA, 1996. ACM.

[TB07] V. T'kindt and J.-C. Billaut. *Multicriteria Scheduling*. Springer, 2007.

hal-00521712, version 1 - 28 Sep 2010

[TC99] Kenjiro Taura and Andrew Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW'1999, the Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 1999.

[TFG+08] George Teodoro, Daniel Fireman, Dorgival Guedes, Wagner Meira Jr., and Renato Ferreira. Achieving multi-level parallelism in filter-labeled stream programming model. In *ICPP'2008, the 37th International Conference on Parallel Processing*, 2008.

[VCK+07] Nagavijayalakshmi Vydyanathan, Umit V. Catalyurek, Tahsin M. Kurc, P. Sadayappan, and Joel H. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par 2007 Parallel Processing*, pages 173–183, 2007.

[VCK+08] Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows. In *ICPP'2008, the 37th International Conference on Parallel Processing*, pages 254–261, Washington, DC, USA, 2008. IEEE Computer Society.

[VCK+10] Nagavijayalakshmi Vydyanathan, Umit V. Catalyurek, Tahsin M. Kurc, P. Sadayappan, and Joel H. Saltz. Optimizing latency and throughput of application workflows on clusters. *Parallel Computing*, 2010. to appear.

[VTL82] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[YKS03] Mau-Tsuen Yang, Rangachar Kasturi, and Anand Sivasubramaniam. A pipeline-based approach for scheduling video processing algorithms on now. *IEEE Transaction on Parallel Distributed Systems*, 14(2):119–130, 2003.

hal-00521712, version 1 - 28 Sep 2010