Parallel Processing Letters, Vol. 13, No. 2 (2003) 225–244

World Scientific
www.worldscientific.com

SCHEDULING STRATEGIES FOR MIXED DATA AND TASK PARALLELISM ON HETEROGENEOUS CLUSTERS

O. BEAUMONT
LaBRI, UMR CNRS 5800, Bordeaux, France
Olivier.Beaumont@labri.fr

A. LEGRAND*, L. MARCHAL† and Y. ROBERT‡
LIP, UMR CNRS-INRIA 5668, ENS Lyon, France
**Arnaud.Legrand@ens-lyon.fr*
†*Loris.Marchal@ens-lyon.fr*
‡*Yves.Robert@ens-lyon.fr*

Received December 2002
Revised April 2003
Accepted by J. Dongarra & B. Tourancheau

ABSTRACT

We consider the execution of a complex application on a heterogeneous "grid" computing platform. The complex application consists of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks. There are dependences (precedence constraints) between these tasks and these dependences are organized as a tree. A typical example is the repeated execution of the same algorithm on several distinct data samples. We use a non-oriented graph to model the grid platform, where resources have different speeds of computation and communication. We show how to determine the optimal steady-state scheduling strategy for each processor (the fraction of time spent computing and the fraction of time spent communicating with each neighbor). This result holds for a quite general framework, allowing for cycles and multiple paths in the platform graph.

Keywords: Heterogeneous processors; scheduling; mixed data and task parallelism; steady-state.

1. Introduction

In this paper, we consider the execution of a complex application, on a heterogeneous "grid" computing platform. The complex application consists of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks. There are dependences (precedence constraints) between these tasks. A typical example is the repeated execution of the same algorithm on several distinct data samples. Consider the simple tree graph depicted in Figure 1. This tree models the algorithm. There is a main loop which is executed several times. Within each loop iteration, there are four tasks to be performed on some matrices. Each loop

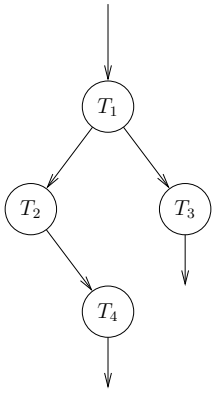

Fig. 1. A simple tree example.

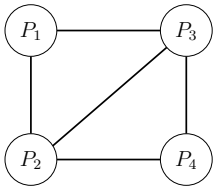

Fig. 2. A simple platform example.

iteration is what we call a problem instance. Each problem instance operates on different data, but all instances share the same *task graph*, i.e. the tree graph of Figure 1. For each node in the task graph, there are as many task copies as there are iterations in the main loop.

We use another graph, the *platform graph*, for the grid platform. We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of an undirected graph. See the example in Figure 2 with four processors and five communication links. Each node is a computing resource (a processor, or a cluster, or whatever) capable of computing and/or communicating with its neighbors at (possibly) different rates. The underlying interconnection network may be very complex and, in particular, may include multiple paths and cycles (just as the Ethernet does).

We assume that one specific node, referred to as the master, initially holds (or generates the data for) the input tasks of all problems. The question for the master is to decide which tasks to execute itself, and how many tasks to forward to each of its neighbors. Due to heterogeneity, the neighbors may receive different amounts of work (maybe none for some of them). Each neighbor faces in turn the same dilemma: determine how many tasks to execute, and how many to delegate to other processors. Note that the master may well need to send tasks along multiple paths to properly feed a very fast but remote computing resource.

Because the problems are independent, their execution can be pipelined. At a given time-step, different processors may well compute different tasks belonging to different problem instances. In the example, a given processor P_i may well compute the tenth copy of task T_1, corresponding to problem number 10, while another processor P_j computes the eight copy of task T_3, which corresponds to problem number 8. However, because of the dependence constraints, note that P_j could not begin the execution of the tenth copy of task T_3 before that P_i has terminated the execution of the tenth copy of task T_1 and sent the required data to P_j (if $i \neq j$).

Because the number of tasks to be executed on the computing platform is expected to be very large (otherwise why deploy the corresponding application on a distributed platform?), we focus on *steady-state* optimization problems rather than on standard *makespan* minimization problems. Minimizing the makespan, i.e. the total execution time, is a NP-hard problem in most practical situations [16, 30, 13], while it turns out that the optimal steady-state can be characterized very efficiently, with low-degree polynomial complexity. For our target application, the optimal steady state is defined as follows: for each processor, determine the fraction of time spent computing, and the fraction of time spent sending or receiving each type of tasks along each communication link, so that the (averaged) overall number of tasks processed at each time-step is maximum. In addition, we will prove that the optimal steady-state scheduling is very close to the absolute optimal scheduling: given a time bound K, it may be possible to execute more tasks than with the optimal steady-state scheduling, but only a constant (independent of K) number of such extra tasks. To summarize, steady-state scheduling is both easy to compute and implement, while asymptotically optimal, thereby providing a nice alternative to circumvent the difficulty of traditional scheduling.

Our application framework is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [27], factoring large numbers [14], the Mersenne prime search [25], and those distributed computing problems organized by companies such as Entropia [15]. Several papers [29, 28, 18, 17, 39, 6, 4] have recently revisited the master-slave paradigm for processor clusters or grids, but all these papers only deal with independent tasks. To the best of our knowledge, the algorithm presented in this paper is the first that allows precedence constraints in a heterogeneous framework. In other words, this paper represents a first step towards extending all the work on mixed task and data parallelism [35, 11, 26, 1, 36] towards heterogeneous platforms. The steady-state scheduling strategies considered in this paper could be directly useful to applicative frameworks such as DataCutter [9, 34].

The rest of the paper is organized as follows. In Section 2, we introduce our base model of computation and communication, and we formally state the steady-state scheduling to be solved. In Section 3, we provide the optimal solution to this problem, using a linear programming approach. We prove the asymptotic optimality of steady-state scheduling in Section 4. We work out two examples in Section 5. We briefly survey related work in Section 6. Finally, we give some remarks and conclusions in Section 7.

2. The Model

We start with a formal description of the application/architecture framework. Next we state all the equations that hold during steady-state operation.

2.1. *Application/architecture framework*

The application

- Let $\mathcal{P}^{(1)}, \mathcal{P}^{(2)}, \ldots, \mathcal{P}^{(n)}$ be the n problems to solve, where n is large
- Each problem $\mathcal{P}^{(m)}$ corresponds to a copy $G^{(m)} = (V^{(m)}, E^{(m)})$ of the same *tree graph* (V, E). The number $|V|$ of nodes in V is the number of task types. In the example of Figure 1, there are four task types, denoted as T_1, T_2, T_3 and T_4.
- Overall, there are $n.|V|$ tasks to process, since there are n copies of each task type.

The architecture

- The target heterogeneous platform is represented by a directed graph, the *platform graph.*
- There are p nodes $P_1, P_2, \ldots, P_p$ that represent the processors. In the example of Figure 2 there are four processors, hence $p = 4$. See below for processor speeds and execution times.
- Each edge represents a physical interconnection. Each edge $e_{ij} : P_i \to P_j$ is labeled by a value c_{ij} which represents the time to transfer a message of unit length between P_i and P_j, in either direction: we assume that the link between P_i and P_j is bidirectional and symmetric. A variant would be to assume two unidirectional links, one in each direction, with possibly different label values. If there is no communication link between P_i and P_j we let $c_{ij} = +\infty$, so that $c_{ij} < +\infty$ means that P_i and P_j are neighbors in the communication graph. With this convention, we can assume that the interconnection graph is (virtually) complete.
- We assume a *full overlap, single-port* operation mode, where a processor node can simultaneously receive data from one of its neighbor, perform some (independent) computation, and send data to one of its neighbor. At any given time-step, there are at most two communications involving a given processor, one in emission and the other in reception. Other models can be dealt with, see [4, 2].

Execution times

- Processor P_i requires $w_{i,k}$ time units to process a task of type T_k.
- Note that this framework is quite general, because each processor has a different speed for each task type, and these speeds are not related: they are *inconsistent* with the terminology of [10]. Of course, we can always simplify the model. For instance we can assume that $w_{i,k} = w_i \times \delta_k$, where w_i is the inverse of the relative

speed of processor P_i, and δ_k the weight of task T_k. Finally, note that routers can be modeled as nodes with no processing capabilities.

Communication times

- Each edge $e_{k,l} : T_k \rightarrow T_l$ in the task graph is weighted by a communication cost $data_{k,l}$ that depends on the tasks T_k and T_l. It corresponds to the amount of data output by T_k and required as input to T_l.
- Recall that the time needed to transfer a unit amount of data from processor P_i to processor P_j is $c_{i,j}$. Thus, if a task $T_k^{(m)}$ is processed on P_i and task $T_l^{(m)}$ is processed on P_j, the time to transfer the data from P_i to P_j is equal to $data_{k,l} \times c_{i,j}$; this holds for any edge $e_{k,l} : T_k \rightarrow T_l$ in the task graph and for any processor pair P_i and P_j. Again, once a communication from P_i to P_j is initiated, P_i (resp. P_j) cannot handle a new emission (resp. reception) during the next $data_{k,l} \times c_{i,j}$ time units.

2.2. *Steady-state equations*

We begin with a few definitions:

- For each edge $e_{k,l} : T_k \rightarrow T_l$ in the task graph and for each processor pair (P_i, P_j), we denote by $s(P_i \rightarrow P_j, e_{k,l})$ the (average) fraction of time spent each time-unit by P_i to send to P_j data involved by the edge $e_{k,l}$. Of course $s(P_i \rightarrow P_j, e_{k,l})$ is a nonnegative rational number. Think of an edge $e_{k,l}$ as requiring a new file to be transferred from the output of each task $T_k^{(m)}$ processed on P_i to the input of each task $T_l^{(m)}$ processed on P_j. Let the (fractional) number of such files sent per time-unit be denoted as $sent(P_i \rightarrow P_j, e_{k,l})$. We have the relation:
$$s(P_i \rightarrow P_j, e_{k,l}) = sent(P_i \rightarrow P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) \quad (1)$$
which states that the fraction of time spent transferring such files is equal to the number of files times the product of their size by the elemental transfer time of the communication link.
- For each task type $T_k \in V$ and for each processor P_i, we denote by $\alpha(P_i, T_k)$ the (average) fraction of time spent each time-unit by P_i to process tasks of type T_k, and by $cons(P_i, T_k)$ the (fractional) number of tasks of type T_k processed per time unit by processor P_i. We have the relation
$$\alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k}\,. \quad (2)$$

We search for rational values of all the variables $s(P_i \rightarrow P_j, e_{k,l})$, $sent(P_i \rightarrow P_j, e_{k,l})$, $\alpha(P_i, T_k)$ and $cons(P_i, T_k)$. We formally state the first constraints to be fulfilled.

Activities during one time-unit. All fractions of time spent by a processor to do something (either computing or communicating) must belong to the interval $[0, 1]$, as they correspond to the average activity during one time unit:

$$\forall P_i, \forall T_k \in V, 0 \leq \alpha(P_i, T_k) \leq 1 \tag{3}$$

$$\forall P_i, P_j, \forall e_{k,l} \in E, 0 \leq s(P_i \rightarrow P_j, e_{k,l}) \leq 1\,. \tag{4}$$

One-port model for outgoing communications. Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i, \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_i \rightarrow P_j, e_{k,l}) \leq 1 \tag{5}$$

where $n(P_i)$ denotes the neighbors of P_i. Recall that we can assume a complete graph owing to our convention with the $c_{i,j}$.

One-port model for incoming communications. Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i, \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_j \rightarrow P_i, e_{k,l}) \leq 1\,. \tag{6}$$

Note that $s(P_j \rightarrow P_i, e_{k,l})$ is indeed equal to the fraction of time spent by P_i to receive from P_j files of type $e_{k,l}$.

Full overlap. Because of the full overlap hypothesis, there is no further constraint on $\alpha(P_i, T_k)$ except that

$$\forall P_i, \sum_{T_k \in V} \alpha(P_i, T_k) \leq 1\,. \tag{7}$$

For technical reasons it is simpler to have a single input task (a task without any predecessor) and a single output task (a task without any successor) in the task graph. To this purpose, we introduce two fictitious tasks, T_{begin} which is connected to the root of the tree and accounts for distributing the input files, and T_{end} which is connected to every task with no successor in the graph. Because these tasks are fictitious, we let $w_{i,begin} = w_{i,end} = 0$ for each processor P_i and no task of type T_{begin} is consumed by any processor (i.e. it requires no time at all).

Using T_{end}, we model two different situations: either the results (the output files of the tree leaves) do not need to be gathered and should stay in place, or all the output files have to be gathered to a particular processor P_{dest} (for visualization or post processing, for example).

In the first situation (output files should stay in place) no file of type $e_{k,end}$ is sent between any processor pair, for each edge $e_{k,end} : T_k \rightarrow T_{end}$. This is ensured by the following equations:

$$\begin{aligned}
&\forall P_i, cons(P_i, T_{begin}) = 0 \\
&\forall P_i, \forall P_j \in n(P_i), \forall e_{k,end} : T_k \to T_{end}\,, \\
&\qquad \begin{cases} sent(P_i \to P_j, e_{k,end}) = 0 \\ sent(P_j \to P_i, e_{k,end}) = 0\,. \end{cases}
\end{aligned} \tag{8}$$

Note that we can let $data_{k,end} = +\infty$ for each edge $e_{k,end} : T_k \to T_{end}$, but we need to add that $s(P_i \to P_j, e_{k,end}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) = 0$ (in other words, $0 \times +\infty = 0$ in this equation).

In the second situation, where results have to be collected on a single processor P_{dest} then, as previously, we let $w_{i,begin} = 0$ for each processor P_i, but we let $w_{dest,end} = 0$ (on the processor that gathers the results) and $w_{i,end} = +\infty$ on the other processors. Files of type $e_{k,end}$ can be sent between any processor pair since they have to be transported to P_{dest}. In this situation, equations 8 have to be replaced by the following equations:

$$\begin{aligned}
&\forall P_i, cons(P_i, T_{begin}) = 0 \\
&\forall P_i \neq P_{dest}, cons(P_i, T_{end}) = +\infty \\
&cons(P_{dest}, T_{end}) = 0
\end{aligned} \tag{9}$$

2.3. *Conservation laws*

The last constraints deal with *conservation laws*: we state them formally, then we work out an example to help understand these constraints.

Consider a given processor P_i, and a given edge $e_{k,l}$ in the task graph. During each time unit, P_i receives from its neighbors a given number of files of type $e_{k,l}$: P_i receives exactly $\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l})$ such files. Processor P_i itself executes some tasks T_k, namely $cons(P_i, T_k)$ tasks T_k, thereby generating as many new files of type $e_{k,l}$.

What does happen to these files? Some are sent to the neighbors of P_i, and some are consumed by P_i to execute tasks of type T_l. We derive the equation:

$$\begin{aligned}
&\forall P_i, \forall e_{k,l} \in E : T_k \to T_l, \sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) \\
&\quad = \sum_{P_j \in n(P_i)} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)\,.
\end{aligned} \tag{10}$$

It is important to understand that equation 10 really applies to the steady-state operation. At the beginning of the operation of the platform, only input tasks are available to be forwarded. Then some computations take place, and tasks of other types are generated. At the end of this initialization phase, we enter the steady-state: during each time-period in steady-state, each processor can simultaneously

perform some computations, and send/receive some other tasks. This is why equation 10 is sufficient, we do not have to detail which operation is performed at which time-step.

In fact, equation 10 does not hold for the master processor P_{master}, because we assume that it holds an infinite number of tasks of type T_{begin}. It must be replaced by the following equation:

$$\forall e_{k,l} \in E : T_k \to T_l \text{ with } k \neq begin,$$

$$\sum_{P_j \in n(P_{\text{master}})} sent(P_j \to P_{\text{master}}, e_{k,l}) + cons(P_{\text{master}}, T_k)$$

$$= \sum_{P_j \in n(P_{\text{master}})} sent(P_{\text{master}} \to P_j, e_{k,l}) + cons(P_{\text{master}}, T_l)\,. \qquad (11)$$

Note that dealing with several masters would be straightforward, by writing equation 11 for each of them.

3. Computing the Optimal Steady-State

The equations listed in the previous section constitute a linear programming problem, whose objective function is the total throughput, i.e. the number of tasks T_{end} consumed within one time-unit:

$$\sum_{i=1}^{p} cons(P_i, T_{end})\,. \qquad (12)$$

Here is a summary of the linear program:

Steady-State Scheduling Problem SSSP(G)

Maximize

$TP = \sum_{i=1}^{p} cons(P_i, T_{end})\,,$

subjectto

$\forall i, \forall k,$ $0 \leq \alpha(P_i, T_k) \leq 1$

$\forall i, j, \forall e_{k,l} \in E,$ $0 \leq s(P_i \to P_j, e_{k,l}) \leq 1$

$\forall i, j, \forall e_{k,l} \in E,$ $s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l})(data_{k,l} \times c_{i,j})$

$\forall i, \forall k,$ $\alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k}$

$\forall i,$ $\sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_i \to P_j, e_{k,l}) \leq 1$

$\forall i,$ $\sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_j \to P_i, e_{k,l}) \leq 1$

$\forall i,$ $\sum_{T_k \in V} \alpha(P_i, T_k) \leq 1$

$\forall i,$ $cons(P_i, T_{begin}) = 0$

$\forall i, j, \forall e_{k,end}$ $sent(P_i \to P_j, e_{k,end}) = 0$

$\forall i, \forall e_{k,l} \in E,$

$\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k)$
$= \sum_{P_j \in n(P_i)} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)$

$\forall e_{k,l} \in E \text{with} k \neq begin,$

$\sum_{P_j \in n(P_{\text{master}})} sent(P_j \to P_{\text{master}}, e_{k,l}) + cons(P_{\text{master}}, T_k)$
$= \sum_{P_j \in n(P_{\text{master}})} sent(P_{\text{master}} \to P_j, e_{k,l}) + cons(P_{\text{master}}, T_l)$

We can state the main result of this paper:

Theorem 1. *The solution to the previous linear programming problem provides the optimal solution to SSSP(G)*

Because we have a linear programming problem in rational numbers, we obtain rational values for all variables in polynomial time (polynomial in the sum of the sizes of the task graph and of the platform graph). When we have the optimal solution, we take the least common multiple of the denominators, and thus we derive an integer period for the steady-state operation.

4. Asymptotic Optimality

In this section we prove that steady-state scheduling is asymptotically optimal. Given a task graph (extended with the initial and final tasks T_{begin} and T_{end}), a platform graph and a time bound K, define $opt(G, K)$ as the optimal number of tasks that can be computed using the whole platform, within K time-units. Let $TP(G)$ be the solution of the linear program SSSP(G) defined in Section 3. We have the following result:

Lemma 1. $opt(G, K) \leq TP(G) \times K$

Proof. Consider an optimal scheduling. For each processor P_i, and each task type T_k, let $t_{i,k}(K)$ be the total number of tasks of type T_k that have been executed by P_i within the K time-units. Similarly, for each processor pair (P_i, P_j) in the platform graph, and for each edge $e_{k,l}$ in the task graph, let $t_{i,j,k,l}(K)$ be the total number of files of type $e_{k,l}$ tasks that have been forwarded by P_i to P_j within the K time-units. The following equations hold true:

- $\sum_k t_{i,k}(K).w_{i,k} \leq K$ (time for P_i to process its tasks)
- $\sum_{P_j \in n(P_i)} \sum_{k,l} t_{i,j,k,l}(K).data_{k,l}.c_{i,j} \leq K$ (time for P_i to forward outgoing tasks in the one-port model)
- $\sum_{P_j \in n(P_i)} t_{j,i,k,l}(K).data_{k,l}.c_{i,j} \leq K$ (time for P_i to receive incoming tasks in the one-port model)
- $\sum_{P_j \in n(P_i)} t_{j,i,k,l}(K) + t_{i,k}(K) = \sum_{P_j \in n(P_i)} t_{i,j,k,l}(K) + t_{i,l}(K)$ (conservation equation holding for each edge type $e_{k,l}$)

Let $cons(P_i, T_k) = \frac{t_{i,k}(K)}{K}$, $sent(P_i \to P_j, e_{k,l}) = \frac{t_{i,j,k,l}(K)}{K}$. We also introduce $\alpha(P_i, T_k) = cons(P_i, T_k).w_{i,k}$ and $s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}).data_{k,l}.c_{i,j}$. All the equations of the linear program SSSP(G) hold, hence $\sum_{i=1}^{p} cons(P_i, T_{end}) \leq TP(G)$, the optimal value.

Going back to the original variables, we derive:

$$opt(G, K) = \sum_i t_{i,end}(K) \leq TP(G) \times K\,.$$

□

Basically, Lemma 1 says that no scheduling can execute more tasks than the steady state scheduling. There remains to bound the loss due to the initialization and clean-up phases to come up with a well-defined scheduling algorithm based upon steady-state operation. Consider the following algorithm (assume K is large enough):

- Solve the linear program SSSP(G): compute the maximal throughput $TP(G)$, compute all the values $\alpha(P_i, T_k)$, $cons(P_i, T_k)$, $s(P_i \rightarrow P_j, e_{k,l})$ and $sent(P_i \rightarrow P_j, e_{k,l})$. Determine the time-period T. For each processor P_i, determine $per_{i,k,l}$, the total number of files of type $e_{k,l}$ that it receives per period. Note that all these quantities are independent of K: they only depend upon the characteristics $w_{i,k}$, $c_{i,j}$, and $data_{k,l}$ of the platform and task graphs.
- Initialization: the master sends $per_{i,k,l}$ files of type $e_{k,l}$ to each processor P_i. To do so, the master generates (computes in place) as many tasks of each type as needed, and sends the files sequentially to the other processors. This requires I units of time, where I is a constant independent of K.
- Similarly, let J be the time needed by the following clean-up operation: each processor returns to the master all the files that it holds at the end of the last period, and the master completes the computation sequentially, generating the last copies of T_{end}. Again, J is a constant independent of K.
- Let $r = \lfloor \frac{K-I-J}{T} \rfloor$.
- Steady-state scheduling: during r periods of time T, operate the platform in steady-state, according to the solution of SSSP(G).
- Clean-up during the J last time-units: processors forward all their files to the master, which is responsible for terminating the computation. No processor (even the master) is active during the very last units ($K - I - J$ may not be evenly divisible by T).
- The number of tasks processed by this algorithm within K time-units is equal to $steady(G, K) = (r + 1) \times T \times TP(G)$.

Clearly, the initialization and clean-up phases would be shortened for an actual implementation, using parallel routing and distributed computations. But on the theoretical side, we do not need to refine the previous bound, because it is sufficient to prove the following result:

Theorem 2. *The previous scheduling algorithm based upon steady-state operation is asymptotically optimal:*

$$\lim_{K \rightarrow +\infty} \frac{steady(G, K)}{opt(G, K)} = 1, .$$

Proof. Using Lemma 1, $opt(G, K) \leq TP(G).K$. From the description of the algorithm, we have $steady(G, K) = ((r+1)T).TP(G) \geq (K - I - J).TP(G)$, hence the result because I, J, T and $TP(G)$ are constants independent of K. □

5. Working Out Some Example

5.1. *A toy example*

In this section we fully work out a numerical instance of the application/architecture platform given in Figures 1 and 2. We start by extending the task graph with T_{begin} and T_{end}, as illustrated in Figure 3.

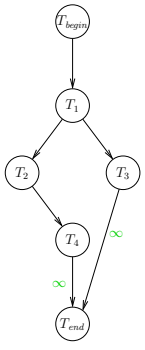

Fig. 3. Extending the tree (task) graph with T_{begin} and T_{end}.

We assume that P_1 is the master processor. We outline the conservation equations which hold for P_1:

$$\begin{aligned}
&sent(P_2 \to P_1, e_{1,2}) + sent(P_3 \to P_1, e_{1,2}) + cons(P_1, T_1) \\
&\quad = cons(P_1, T_2) + sent(P_1 \to P_2, e_{1,2}) + sent(P_1 \to P_3, e_{1,2}) \\
&sent(P_2 \to P_1, e_{1,3}) + sent(P_3 \to P_1, e_{1,3}) + cons(P_1, T_1) \\
&\quad = cons(P_1, T_3) + sent(P_1 \to P_2, e_{1,3}) + sent(P_1 \to P_3, e_{1,3}) \\
&sent(P_2 \to P_1, e_{2,4}) + sent(P_3 \to P_1, e_{2,4}) + cons(P_1, T_2) \\
&\quad = cons(P_1, T_4) + sent(P_1 \to P_2, e_{2,4}) + sent(P_1 \to P_3, e_{2,4}) \\
&sent(P_2 \to P_1, e_{3,end}) + sent(P_3 \to P_1, e_{3,end}) + cons(P_1, T_3) \\
&\quad = cons(P_1, T_{end}) + sent(P_1 \to P_2, e_{3,end}) + sent(P_1 \to P_3, e_{3,end}) \\
&sent(P_2 \to P_1, e_{4,end}) + sent(P_3 \to P_1, e_{4,end}) + cons(P_1, T_4) \\
&\quad = cons(P_1, T_{end}) + sent(P_1 \to P_2, e_{4,end}) + sent(P_1 \to P_3, e_{4,end})
\end{aligned}$$

Similarly, the following conservation equations hold for P_2:

$$\begin{aligned}
&sent(P_1 \to P_2, e_{begin,1}) + sent(P_3 \to P_2, e_{begin,1}) + sent(P_4 \to P_2, e_{begin,1}) \\
&\quad + cons(P_2, T_{begin}) = cons(P_2, T_1) + sent(P_2 \to P_1, e_{begin,1}) \\
&\quad + sent(P_2 \to P_3, e_{begin,1}) + sent(P_2 \to P_4, e_{begin,1})
\end{aligned}$$

$$
\begin{aligned}
&sent(P_1 \to P_2, e_{1,2}) + sent(P_3 \to P_2, e_{1,2}) + sent(P_4 \to P_2, e_{1,2}) \\
&\quad + cons(P_2, T_1) = cons(P_2, T_2) + sent(P_2 \to P_1, e_{1,2}) \\
&\quad + sent(P_2 \to P_3, e_{1,2}) + sent(P_2 \to P_4, e_{1,2}) \\
&sent(P_1 \to P_2, e_{1,3}) + sent(P_3 \to P_2, e_{1,3}) + sent(P_4 \to P_2, e_{1,3}) \\
&\quad + cons(P_2, T_1) = cons(P_2, T_3) \\
&\quad + sent(P_2 \to P_1, e_{1,3}) + sent(P_2 \to P_3, e_{1,3}) + sent(P_2 \to P_4, e_{1,3}) \\
&sent(P_1 \to P_2, e_{2,4}) + sent(P_3 \to P_2, e_{2,4}) + sent(P_4 \to P_2, e_{2,4}) \\
&\quad + cons(P_2, T_2) = cons(P_2, T_4) + sent(P_2 \to P_1, e_{2,4}) \\
&\quad + sent(P_2 \to P_3, e_{2,4}) + sent(P_2 \to P_4, e_{2,4}) \\
&sent(P_1 \to P_2, e_{3,end}) + sent(P_3 \to P_2, e_{3,end}) + sent(P_4 \to P_2, e_{3,end}) \\
&\quad + cons(P_2, T_3) = cons(P_2, T_{end}) + sent(P_2 \to P_1, e_{3,end}) \\
&\quad + sent(P_2 \to P_3, e_{3,end}) + sent(P_2 \to P_4, e_{3,end}) \\
&sent(P_1 \to P_2, e_{4,end}) + sent(P_3 \to P_2, e_{4,end}) + sent(P_4 \to P_2, e_{4,end}) \\
&\quad + cons(P_2, T_4) = cons(P_2, T_{end}) + sent(P_2 \to P_1, e_{4,end}) \\
&\quad + sent(P_2 \to P_3, e_{4,end}) + sent(P_2 \to P_4, e_{4,end})
\end{aligned}
$$

Next we add numerical values for the $w_{i,k}$, the $c_{i,j}$ and the $data_{k,l}$: see Figure 4. The values of the $data_{k,l}$ are indicated along the edges of the task graph. The values of the $c_{i,j}$ are indicated along the edges of the platform graph. For the sake of simplicity, we let $w_{i,k} = w_i \times \delta_k$ for all tasks T_k, where the corresponding values for w_i are indicated close to the nodes of the platform graph and the corresponding values for δ_k are indicated close to the nodes of the dependency graph. The master processor P_1 is circled in bold.

We feed the values $c_{i,j}$, $w_{i,k}$ and $data_{k,l}$ into the linear program, and compute the solution using a tool like the Maple simplex package [12] or `lp_solve` [7]. We obtain the optimal throughput $TP = 39/280$. This means that the whole platform is equivalent to a single processor capable of processing 39 tasks every 280 seconds. The actual period is equal to 63000. The resulting values for $cons(P_i, T_k)$ are gathered in Table 1.

The resulting values for $s(P_i \to P_j, e_{k,l})$ can be summarized in the following way:

- $P_1 \to P_2$: a fraction 0.31% of the time is spent communicating (files for edge) $e_{begin,1}$, a fraction 10.49% of the time is spent communicating $e_{1,2}$, a fraction 6.65% of the time is spent communicating $e_{1,3}$, and a fraction 4.05% of the time is spent communicating $e_{2,4}$;

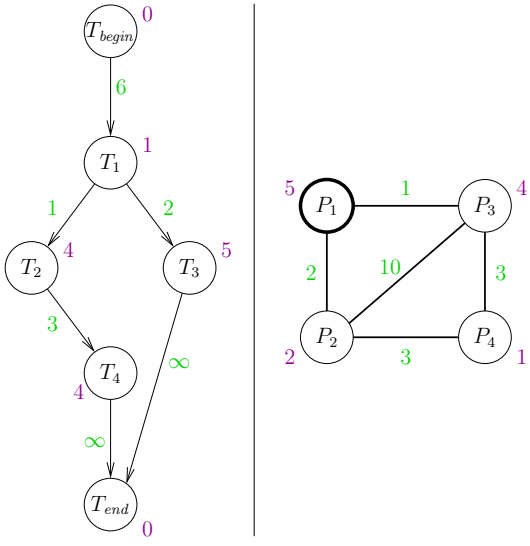

Fig. 4. The application/architecture example with numerical values.

Table 1. Optimal solutions for $cons(P_i, T_k)$.

	T_1	T_2	T_3	T_4
P_1	75.3%	0	7.6%	7.6%
P_2	0	0	39.9%	39.9%
P_3	0	0	19.9%	19.9%
P_4	24.7%	100%	32.6%	32.6%
Total	39 tasks every 280 seconds			

- $P_1 \rightarrow P_3$: a fraction 3.13% of the time is spent communicating $e_{begin,1}$, and a fraction 2.77% of the time is spent communicating $e_{1,3}$;
- $P_2 \rightarrow P_4$: a fraction 0.31% of the time is spent communicating $e_{begin,1}$, a fraction 10.48% of the time is spent communicating $e_{1,2}$, a fraction 1.10% of the time is spent communicating $e_{1,3}$;
- $P_3 \rightarrow P_1$: a fraction 5.11% of the time is spent communicating $e_{2,4}$;
- $P_3 \rightarrow P_2$: a fraction 0.51% of the time is spent communicating $e_{2,4}$ (as the bandwidth of $P_3 \leftrightarrow P_2$ is very low, it is not worth using this link too much, which explains why this fraction is so low);
- $P_3 \rightarrow P_4$: 3.13% of the time is spent communicating $e_{begin,1}$;
- $P_4 \rightarrow P_2$: 0.98% of the time is spent communicating $e_{2,4}$;
- $P_4 \rightarrow P_3$: 8.41% of the time is spent communicating $e_{2,4}$;

It is worth pointing out that the optimal solution is not trivial, in that processors do not execute tasks of all types. In the example, the processors are equally efficient

on all task types: $w_{i,k} = w_i \times \delta_k$, hence only relative speeds count. We could have expected each problem to be processed by a single processor, that would execute all the tasks of the problem, in order to avoid extra communications; in this scenario, the only communications would correspond to the input cost $c_{begin,1} = 6$. However, the intuition is misleading. In the optimal steady state solution, some processor do not process some task types at all (see P_2 and P_3) , and some task types are executed by one processor only (see T_2). This example demonstrates that in the optimal solution, the processing of each problem may well be distributed over the whole platform.

An intuitive alternate strategy is the *coarse-grain* approach, where we require each problem instance to be processed by a single processor: this processor would execute all the tasks of the problem instance, thereby avoiding extra communications; in this scenario, the only communications correspond to the input cost $c_{begin,1} = 2$. We make several comments to this approach:

- First, there still remains to decide how many problems will be executed by each processor. Our framework does provide the optimal answer: simply replace the task graph by a single task, and compute its weight on a given processor as the sum of the task durations on that processor.
- The number of equations will be be smaller with the *coarse-grain* approach, but the time to solve the linear program is not a limiting factor: the answer is instantaneous for the toy example. Larger examples require a few seconds of off-the-shelf CPU time (see below).
- The *coarse-grain* approach may be simpler to implement, so it is a good strategy to compute its throughput, and to compare it to the optimal solution. The idea is to balance the gain brought by the optimal solution against the additional complexity of the implementation. In the example, the throughput is equal to 311 tasks every 2520 seconds, 11.4 % less than the optimal steady state solution.

Furthermore, it may well be the case that some processors are more efficient for certain task types (think of a number-cruncher processor, a visualization processor, etc): then the *coarse-grain* approach will not be efficient (and even unfeasible if some processors cannot handle some tasks at all). We handle these constraints by writing the equations in a general setting, using the unknowns $w_{i,k}$ (the time for P_i to process task type T_k).

To illustrate this point, consider the example again and let $w_{3,2} = 1$. Using a coarse grain approach, the throughput would be equal to ≈ 0.13 task per second, whereas the optimal throughput is equal to ≈ 0.170 task per second. which is an improvement of more than 31%. This illustrates the full potential of the mixed data and task parallelism approach.

5.2. *A more realistic example*

In this section we work out a numerical instance of the application/architecture platform given in Figures 5. The values of the $data_{k,l}$ are indicated along the edges

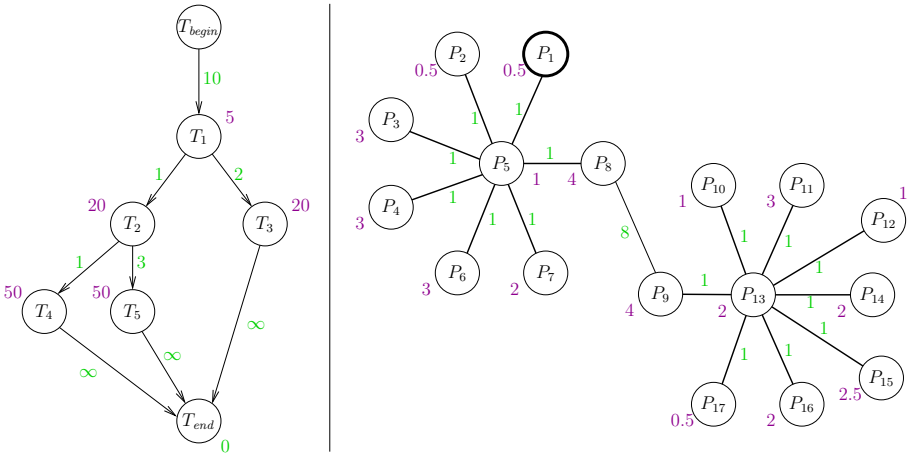

Fig. 5. The application/architecture example with numerical values.

of the task graph. The values of the $c_{i,j}$ are indicated along the edges of the platform graph. For the sake of simplicity, we let $w_{i,k} = w_i \times \delta_k$ for all tasks T_k, where the corresponding values for w_i are indicated close to the nodes of the platform graph and the corresponding values for δ_k are indicated close to the nodes of the dependency graph. The master processor P_1 is circled in bold.

Solving the equations took less than two seconds with `lp_solve` [7] on a P-III 800MHz. The results are summarized in Table 2.

Table 2. Optimal solutions for $cons(P_i, T_k)$.

	T_1	T_2	T_3	T_4	T_5
P_1	100.000%	0	15.039%	15.039%	15.039%
P_2	0	51.987%	10.541%	10.541%	10.541%
P_3	0	0	3.201%	3.201%	3.201%
P_4	0	0	3.201%	3.201%	3.201%
P_5	0	0	9.603%	9.603%	9.603%
P_6	0	0	3.201%	3.201%	3.201%
P_7	0	0	4.801%	4.801%	4.801%
P_8	0	0	2.401%	2.401%	2.401%
P_9	0	0	0	0	0
P_{10}	0	0	9.603%	9.603%	9.603%
P_{11}	0	0	0	0	0
P_{12}	0	0	9.603%	9.603%	9.603%
P_{13}	0	0	4.801%	4.801%	4.801%
P_{14}	0	19.205%	1.600%	1.600%	1.600%
P_{15}	0	0	3.201%	3.201%	3.201%
P_{16}	0	28.808%	0	0	0
P_{17}	0	0	19.205%	19.205%	19.205%
Total	0.0867816 task per second				

In the optimal solution, the two clusters (P_1 to P_8 and P_9 to P_{17}) are almost equally working, despite the slow inter-cluster communication link and the location of the input files on P_1. Note that the *coarse-grain* approach leads to a throughput equal to ≈ 0.0590517 task per second. The improvement of our method over a *coarse-grain* approach is then of more than 46%.

6. Related Problems

There are two papers that are closely related to our work. First, the proof of the asymptotic optimality of steady-state scheduling (Section 4) is inspired by the paper of Bertsimas and Gamarnik [8], who have used a fluid relaxation technique to derive asymptotically optimal scheduling algorithms. They apply this technique to the job shop scheduling problem and to the packet routing problem. It would be very interesting to extend their results to a heterogeneous framework.

The second paper is by Taura and Chien [37], who consider the pipeline execution of task graphs onto heterogeneous platforms. This is exactly the problem that we target in this paper, except that they do not restrict to tree-shaped task graphs. Taura and Chien make the hypothesis that all copies of a given task type must be executed on the same processor. This hypothesis was intended as a simplification, but it renders the problem NP-complete, and the authors had to resort to sophisticated heuristics. With Taura and Chien's hypothesis, the problem obviously remains NP-complete for tree-shaped task graphs while we were able to compute the optimal solution in polynomial time, just because we allowed the execution of all the copies of the same task type to be shared by several processors.

More generally, there are several related papers, which we classify along the following main lines:

Scheduling task graphs on heterogeneous platforms. Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors, see Maheswaran and Siegel [23], Oh and Ha [24], Topcuoglu, Hariri and Wu [38], and Sih and Lee [31] among others. Unfortunately, all these heuristics assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications. Recent papers by Hollermann, Hsu, Lopez and Vertanen [19], Hsu, Lee, Lopez and Royce [20], and Sinnen and Sousa [33, 32], suggest to take communication contention into account. Among these extensions, scheduling heuristics under the one-port model (see Johnsson and Ho [21] and Krumme, Cybenko and Venkataraman [22]) are considered in [3]: just as in this paper, each processor can communicate with at most another processor at a given time-step.

Master-slave on the computational grid. Master-slave scheduling on the grid can be based on a network-flow approach (see Shao, Berman and Wolski [29] and Shao [28]), or on an adaptive strategy (see Heymann, Senar, Luque and Livny [18]). Note that the network-flow approach of [29, 28] is possible only when using a

full multiple-port model, where the number of simultaneous communications for a given node is not bounded. Enabling frameworks to facilitate the implementation of master-slave tasking are described in Goux, Kulkarni, Linderoth and Yoder [17], and in Weissman [39].

Mixed task and data parallelism. There are a very large number of papers dealing with mixed task and data parallelism. We quote the work of Subhlok, Stichnoth, O'Hallaron and Gross [35], Chakrabarti, Demmel and Yelick [11], Ramaswamy, Sapatnekar and Banerjee [26], Bal and M. Haines [1], and Subhlok and Vondran [36], but this list is by no means meant to be comprehensive. We point out, however, that (to the best of our knowledge) none of the papers published in this area is dealing with heterogeneous platforms.

7. Conclusion

In this paper, we have dealt with the implementation of mixed task and data parallelism onto heterogeneous platforms. We have shown how to determine the best steady-state scheduling strategy for a tree-shaped task graph and for a general platform graph, using a linear programming approach.

This work can be extended in the following three directions:

- The first idea is to extend the approach to arbitrary task graphs, i.e. general DAGs instead of trees. In fact, our approach can be extended to general DAGs, but with a complexity which is proportional to the number of paths in the DAG [5]. A large number of task graphs, such as trees, fork-join graphs, etc, do have a polynomial number of paths. But for instance the Laplace task graph, which is an oriented two-dimensional grid, has a number of paths exponential in its node number. At the time of this writing, we do not whether the problem can be solved in polynomial time for arbitrary task graphs.
- On the theoretical side, we could try to solve the problem of maximizing the number of tasks that can be executed within K time-steps, where K is a given time-bound. This scheduling problem is more complicated than the search for the best steady-state. Taking the initialization phase into account renders the problem quite challenging.
- On the practical side, we need to run actual experiments rather than simulations. Indeed, it would be interesting to capture actual architecture and application parameters, and to compare heuristics on a real-life problem suite, such as those in [9, 34].

Acknowledgment

We would like to thank the reviewers for their constructive comments and suggestions.

References

[1] H. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

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

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

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2002*. IEEE Computer Society Press, 2002. Extended version available as LIP Research Report 2001-25.

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

[6] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In D. S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, editors, *Cluster'2001*, pages 419–426. IEEE Computer Society Press, 2001. Extended version available as LIP Research Report 2001-13.

[7] Michel Berkelaar. LP_SOLVE: Linear Programming Code. URL: `http://www.cs.sunysb.edu/ algorith/implement/lpsolve/implement.shtml`.

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

[9] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002.

[10] T. D. Braun, H. J. Siegel, and N. Beck. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel and Distributed Computing*, 61:810–837, 2001.

[11] S. Chakrabarti, J. Demmel, and K. Yelick. Models and scheduling algorithms for mixed data and task parallel programs. *J. Parallel and Distributed Computing*, 47:168–184, 1997.

[12] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *Maple Reference Manual*, 1988.

[13] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[14] James Cowie, Bruce Dodson, R.-Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Joerg Zayer. A world wide number field sieve factoring record: on to 512 bits. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - Asiacrypt '96*, volume 1163 of *LNCS*, pages 382–394. Springer Verlag, 1996.

[15] Entropia. URL: `http://www.entropia.com`.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[17] J. P Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society Press, 2000.

[18] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

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

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

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

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

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

[24] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, LNCS 1123. Springer Verlag, 1996.

[25] Prime. URL: `http://www.mersenne.org`.

[26] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

[27] SETI. URL: `http://setiathome.ssl.berkeley.edu`.

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

[29] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.

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

[31] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

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

[33] O. Sinnen and L. Sousa. Exploiting unused time-slots in list scheduling considering communication contention. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *EuroPar'2001 Parallel Processing*, pages 166–170. Springer-Verlag LNCS 2150, 2001.

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

[35] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Fourth ACM SIGPLAN Symposium on Priciples & Practices of Parallel Programming*. ACM Press, May 1993.

[36] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel and Distributed Computing*, 60:297–319, 2000.

[37] K. Taura and A. A. Chien. A heuristic algorithm for mapping communicating tasks

on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 2000.

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

[39] J. B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.