Symbolic Partitioning and Scheduling of Parameterized Task Graphs

Michel Cosnard
LORIA INRIA Lorraine
615, rue du Jardin Botanique
54602 Villers les Nancy, France
Michel.Cosnard@loria.fr

Emmanuel Jeannot
LIP ENS-Lyon
46, allée d'Italie
69364 Lyon, France
ejeannot@ens-lyon.fr

Tao Yang
CS Dept. UCSB
Engr Building I
Santa Barbara, CA 93106, USA
tyang@cs.ucsb.edu

Abstract

The DAG-based task graph model has been found effective in scheduling for performance prediction and optimization of parallel applications. However the scheduling complexity and solution normally depend on the problem size. In this paper, we propose a symbolic scheduling scheme for a parameterized task graph which models coarse-grain DAG parallelism independent of the problem size. The algorithm first derives symbolic clusters to group of tasks in order to minimize communication while preserving parallelism and then it evenly assigns task clusters to processors. The runtime system executes clusters on each processor in a multithreaded fashion. This paper also presents preliminary experimental results to demonstrate the effectiveness of our techniques.

1. Introduction

This paper addresses scheduling issues for parallelism that can be modeled as directed acyclic dependence graphs (DAG) [22]. This model has been found useful in performance prediction and code optimization for parallel applications [1, 3, 8, 11, 13]. Recently task graphs are adopted by several DARPA research teams for performance prediction and modeling of parallel applications on current and future machine architectures[1].

The previous work on task scheduling [12, 16, 22, 25] for performance prediction and optimization has mainly focused on searching on a graph corresponding to a particular problem instance. The scheduling complexity and the derived solution depend on the graph size. Thus the scheme is not scalable for solving problems of large sizes since a large task graph may not fit memory. Also this approach is not adaptive to the runtime variation of the problem size; the change of the problem sizes requires scheduler to re-do the entire computation mapping and optimization. To address the aforementioned weakness, this paper proposes a symbolic scheduling algorithm using the parameterized task graph model.

The previous work in parallelizing compilers [2, 7, 9] has studied the compact parallelism representation based on fine-grain level dependence analysis. The unique aspect of our parameterized task model is that each task node is coarse grained, performing a chunk of computation. The past work has used fine-grain dependence information to partition computation, e.g. tiling [15, 21]; however there is no research work published to produce a coarse-grain level task dependence graph. We have addressed this issue in our early work [17], which proposes compile-time techniques for data aggregation and dependence summary. This paper focuses on symbolic scheduling of parameterized task graphs on distributed memory machines. The main optimization conducted in the previous DAG scheduling research is eliminating unnecessary communication to exploit data locality, overlapping communication with computation to hide communication latency, and exploiting task concurrency to balance loads among processors. Those optimization goals can be achieved by traversing a static task graph. The main challenge in designing a symbolic algorithm for a parameterized graph is that optimization must be performed on a compact representation of task nodes and the result must be effective for various problem size settings. Our strategy is to first perform symbolic clustering which assigns tasks to the same processor in order to reduce inter-task communication while still preserving available parallelism. Then we assign symbolic clusters uniformly to processors. Since each processor may own several symbolic task clusters, our runtime scheme

[1] http://www.cs.umd.edu/projects/hpsl/arpa/Aug19/DARPAworkshop.htm

```
param n
assert n >= 3
real a(n, n+1),s
for k = 1 to n-1 do
  task                              /*T1(k)*/
    s= 1 / a(k,k)
    for l = k + 1 to n do
      a(l,k) = a(l,k) * s
    endfor
  endtask
    for j = k + 1 to n+1 do
      task                        /*T2(k,j)*/
        for i= k + 1 to n do
          a(i,j) = a(i,j) -
                   a(k,j) * a(i,k)
        endfor
      endtask
    endfor
endfor
```

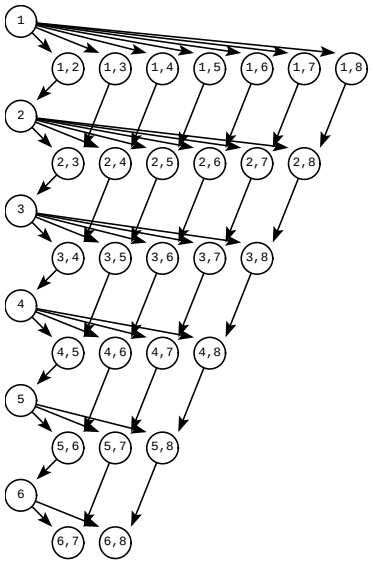

Figure 1. Gaussian Elimination and expanded task graph(n=7)

executes clusters on each processor using a multi-threaded executing fashion to overlap computation with communication.

This paper is organized as follows. Section 2 presents the concept of parameterized task graphs. Section 3 gives an overview of our symbolic scheduling algorithm. Section 4 presents our linear clustering algorithm. Section 5 presents the symbolic method for mapping clusters to processors. Section 6 discusses our runtime control scheme for executing task clusters in a multi-threaded manner. Section 7 presents some preliminary experimental results. Section 8 concludes the paper.

2. Parameterized Task Graphs

A parameterized task graph is represented by the following tuple $PTG = \{T, R, C\}$, where:

- T is the set of generic tasks
- R is the set of communication rules,
- C is the cost function of each task.

A generic task is defined by its name, its iteration vector and its code.

The communication set R is composed of reception rules and emission rules. Reception and emission rules are dual forms of each other. Reception rules describe the set of fathers of a given task. Emission rules describe the set of sons of a given task. Most of the time we will deal with emission rules, but keep in mind that for each emission rule we also have a reception rule.

An emission rule has the form:

$$Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y})|P$$

This means that if predicate P is true, task $Ta(\vec{u})$ sends data $D(\vec{y})$ to task $Tb(\vec{v})$. With each rule is associated polyhedron P which described the valid values of each iteration vector: it is a predicate on the variables of $\vec{u}$, $\vec{v}$ and $\vec{y}$.

2.1. An example

We illustrate a parameterized task graph using a Gaussian Elimination code. The graph is derived from an annotated sequential code by the PlusPyr tool built in our early work [5, 17]. Figure 1 is a Gaussian Elimination code as the input of the PlusPyr. Keywords *task* and *endtask* delimit the parts of the program that have to be executed in sequential.

This Gaussian elimination program has two generic tasks $T1(k)$ and $T2(k, j)$. Its parameterized task graph, computed by our tool PlusPyr, is shown in Figure 2. In general, PlusPyr is able to derive the parameterized task graph (PTG) from an annotated sequential program. The analysis performed by PlusPyr also gives the *symbolic computational cost* of

1. $T1(k) \longrightarrow T2(k,j) : A(i,k) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$
2. $T2(k,j) \longrightarrow T1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$
3. $T2(k,j) \longrightarrow T2(k+1,j) : A(i,j) | 1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n$

Figure 2. Emission rules for the Gaussian Elimination

each generic task. This analysis is based on integer parametric programming techniques developed by Feautrier and others [5, 6, 9, 23]. PlusPyr is a suitable tool for analyzing compute-intensive kernels that can be found in scientific programs.

2.2. Bijection rules

In this paper we target programs on dense matrices with static control (loop bounds and array references are affine function of loop indices), which are, in many applications, the most time-consuming part.

The communication rule $Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y}) | P$ is called a **bijection rule** if, for each instance of iteration vector $\vec{u}$ there exists exactly one instance of iteration vector $\vec{v}$, associated to $\vec{u}$ by this rule.In [4] we have shown how to determine automatically if a rule is a bijection rule or not.

In Figure 2, rule number 1 is a not a bijection rule. Indeed, we see that for one value of k (i.e. one instance of task $T1$), j can take all the values between $k+1$ and n ($n-k$ instances of $T2$).

3. Overview of the Method

We first give a summary of the main steps needed to compute and execute the scheduling of a parameterized task graph:

1. We first look at all the bijection rules in order to derive a symbolic linear clustering. We use a technique called "symbolic edge zeroing" to group tasks which have data dependence. A constraint we impose is that the instantiated clusters will always be linear for any parameter setting, i.e. no independent tasks are assigned to the same cluster. In this way, parallelism is preserved while unnecessary communication is reduced.
2. When the symbolic linear clustering is done, the second step is to evenly map clusters to processors. In this step, we will assign a symbolic identification to each cluster. Different clusters should have different identification numbers so that we evenly distribute clusters to processors. Another thing involved is the mapping and packaging of the data. This is done using the communication rules. When a rule is not zeroed this means that data will be sent from a processor to an other. The rules describe the kind of data to be sent, e.g. scalar, vector, matrix. We use this information for generating the massage packaging code.
3. The final step deals with the execution of the program. Our parallel program is a multi-threaded SPMD one. Each processor of the parallel machine manages two queues. The waiting queue, for the tasks waiting for data and the ready queue for tasks ready for execution. Each node executes n threads (n is fixed at the beginning of the execution). Each thread scans the ready queue, executes the ready task and sends data to the processors. An important aspect of this step is that a processor needs to find the entry tasks for each symbolic cluster assigned to that processor. In this way, task queues can be properly initialized.

4. Symbolic Linear Clustering

Let us consider the following rule: $R : Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y}) | P$

We define a **cluster** as a set of tasks which have to be executed on a same processor. We use the following notation: $\kappa(Ta, \vec{v})$, is a *function*, that gives for all the vectors $\vec{v}$ in polyhedron P, the cluster number of task Ta.

The **zeroing process** is defined as follows. Rule R is said *zeroed* if and only if $\kappa(Ta, \vec{u}) = \kappa(Tb, \vec{v})$ for all the valid instances of $\vec{u}$ and $\vec{v}$ in P. Intuitively, when a rule is zeroed every couple ($Ta(\vec{u})$,$Tb(\vec{v})$) will be put on the same processor leading to a *zero* cost communication between them.

A cluster is said **linear** if it does not contain two tasks which are independent. The justification to use linear clustering is that as proven by Yang and Gerasoulis [14], a linear clustering preserves parallelism and provides good schedules on an unbounded number of processors when tasks are coarse grained.

4.1. The symbolic edge zeroing algorithm

Our symbolic linear clustering for a parameterized task graph is summarized in Figure 3. The zeroing algorithm pro-

Input: a parameterized task graph PTG={T,R,D}
Output: The set $\mathcal{Z}$ of the rule that have to be zeroed.
$\mathcal{Z}$:=$\emptyset$
bijection_rule_set:=compute_bijection_rule(R)
sort(bijection_rule_set);
foreach r **in** bijection_rule_set **do**
  if not_in_conflict(r,$\mathcal{Z}$) **then**
    $\mathcal{Z}$+=r
  endif
enddo

Figure 3. The Zeroing Algorithm

ceeds as follow. It considers all the bijection rules. It sorts them such that the rules implying more communication will be zeroed first. The ordering of the rules is done by taking into consideration the dimension of the sent data (i.e. a scalar, a row or a column, a block of matrix). Then it checks if a given rule is not in conflict with a rule already zeroed. Two rules are in conflict if, once zeroed, the produced clustering is not linear. If the rule is not in conflict with any other zeroed rule then the algorithm adds this rule to the set of zeroed rules. The algorithm repeats this process until no more rule can be zeroed.

In the next subsection, we discuss how we detect a zeroing conflict which causes the clustering to be nonlinear.

4.2. The conflict problem

Let us consider $R1$ and $R2$, the two following emission rules:

$$R1 : Ta(\vec{u}) \longrightarrow Tb(\vec{v})|P_1$$
$$R2 : Tc(\vec{w}) \longrightarrow Td(\vec{z})|P_2$$

Suppose $R1$ is already zeroed. We want to check if $R2$ is not in conflict with $R1$. There is two kind of conflict: the fork conflict and the join conflict:

- We have a *fork conflict* if:
  1. $Ta = Tc$
  2. There exists an instance of $\vec{u}$ called $\vec{u_1}$, $\vec{u_1} \in P_1$ and an instance of $\vec{w}$, called $\vec{w_1}$, $\vec{w_1} \in P_2$ such that $Ta(\vec{u_1}) = Tc(\vec{w_1})$,
  3. Let $\vec{v_1}$ the image of $\vec{u_1}$ according to rule $R1$ and $\vec{z_1}$ the image of $\vec{w_1}$ according to rule $R2$. $\vec{v_1}$ and $\vec{z_1}$ are such that $Tb(\vec{v_1}) \neq Td(\vec{z_1})$
- We have a *join conflict* if:
  1. $Tb = Td$
  2. There exists an instance of $\vec{v}$, called $\vec{v_1}$, $v_1 \in P_1$ and an instance of $\vec{z}$ called $\vec{z_1}$, $\vec{z_1} \in P_3$ such that $Tb(\vec{v_1}) = Td(\vec{z_1})$
  3. Let $\vec{u_1}$ the source of $\vec{v_1}$ according to rule $R1$ and $\vec{w_1}$ the source of $\vec{z_1}$ according to rule $R2$. $\vec{u_1}$ and $\vec{w_1}$ are such that $Ta(\vec{u_1}) \neq Tc(\vec{w_1})$

In order to automatically compute if two rules are in *fork conflict* (the *join conflict* case is symmetric) we proceed as follows. Concerning step 2, we compute the set I, which is the intersection of all the tasks $Ta(\vec{u}), \vec{u} \in P_1$ with all the tasks $Tc(\vec{w}), \vec{w} \in P_2$. If I is not empty then step 2 is verified. Concerning step 3, we need $Tb = Td$ and we compute if $R1$ restricted to I is equal to $R2$ restricted to I, if not then the $R1$ and $R2$ are in *fork conflict*. We can compute I, the restriction of $R1$ and $R2$ to I symbolically using the *omega test* program of the university of Maryland [20].

Let come back to our example. Rule 2 and rule 3 can be in fork conflict. But rule 2 is valid if $j = k+1$ and rule 3 is valid if $k+2 \leq j \leq n+1$. Hence there is no instance of $T2(k,j)$ which have two sons: both $T2(k+1,j)$ and $T1(k+1)$. So rule 2 and 3 are not in conflict and can be can be both zeroed.

The zeroing algorithms build a set of bijection rules which are never in conflict.

Hence, we have the following theorem:

Theorem 1 *Let $G = (T, R, D)$ a PTG where R is composed of bijection rule and there is no rule in conflict then zeroing all the rules build linear clustering of G.*

Proof of theorem 1 Each rule is a bijection rule. Therefore, zeroing a rule leads, in the instantiated task graph, to zero exactly one edge for each instance of the rule.

No rules are in conflict. Therefore, zeroing all the rules leads, in the instantiated task graph, to a clustering with no independent tasks in the same cluster. ∎

5. Mapping of Symbolic Clusters

Given p processors, we assign symbolic clusters evenly to processors. Let c be the identification of a cluster instance, the processor of this cluster is c mod p. The main issue is how we assign an identification to such a cluster. The numbering of clusters must be assigned as uniform as possible in an integer space. In this way, clusters can be evenly distributed to processors.

We need to build a function $\kappa(Ta, \vec{u})$ that, for any generic task Ta and any valid instance $\vec{u}$ gives a cluster number, such that all the tasks in the same cluster will have the same number.

Let us consider rule:

$$R1 : Ta(\vec{u}) \longrightarrow Tb(\vec{v})|P_1$$

If this rule is zeroed then, in P_1, we must have:

$$\kappa(Ta, \vec{u}) = \kappa(Tb, \vec{v}) \quad (1)$$

The basis of our method is the same as the one used by Feautrier in [10]. As Feautrier we will suppose that, in its general form,

$$\kappa(Ta, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a} \quad (2)$$

where $\vec{\alpha_a}$ and $\vec{\beta_a}$ are vectors of unknowns which have to be found. We also have $\kappa(Tb, \vec{v}) = \vec{\alpha_b}\vec{u} + \vec{\beta_b}$ Then solving equation 1 leads to $\vec{\alpha_a}\vec{u} + \vec{\beta_a} = \vec{\alpha_b}\vec{v} + \vec{\beta_b}$.

From each rules we derive equations of this form leading to a system. Solving this system gives us the values of all the $\vec{\alpha}$'s and $\vec{\beta}$'s for every tasks. In order to do that we have modified Feautrier's algorithm and we are able to derive piecewise affine clustering functions when necessary. This leads to a more complicated function than the one described in equation 2 and this is done by splitting the PTG (see [4] for all the details). For our Gaussian Elimination example, this method finds:

$$\kappa(T2, (k, j)) = j$$

$$\kappa(T1, (k)) = k$$

6. Multi-threaded Execution of Clusters

So far each processor has been assigned a number of clusters. Assume that there are t clusters. Each cluster is a linear task chain and is considered a thread on that processor. At most t tasks can be executed at the same time on a processor. Multi-threading is important for hiding communication latency and overlapping computation with communication [24].

6.1. Finding a starting task

In order to initiate computation at each processor, we need to know the set of starting tasks (i.e. the set of tasks that have no father). This is done by symbolically subtracting the set of all the tasks with the set of the tasks that only receive data.

6.2. The execution scheme

We have shown how to build parameterized polyhedrons that describe the set of starting tasks. At run time, once the parameter values are given, each processor P_x scans the set of starting tasks. For each starting task it computes its processor P'_x. If $P_x = P'_x$ then it executes the task.

For each generic task we can build a polynomial that computes its number of fathers. This polynomial is computed from the reception rules and can be used for any valid instance of the iteration vector and any value of the parameters. We propose the following protocol for task ordering: we maintain a ready queue for the tasks ready to be executed and a waiting queue for the tasks that have already received data and are waiting for other data.

- When a task receives a data item:
  1. If this task is not in the waiting queue then we compute its number of fathers and decrement it. This number is called the *number of remaining fathers*. If its current number of remaining fathers is 0, we put this task in the ready queue, otherwise we put it in the waiting queue.
  2. If this task is in the waiting queue then we decrement its number of remaining fathers and, if this number reaches 0, we put it in the ready queue.
- When a thread is idle, it picks up a task in the ready queue (if not empty).

The execution of the tasks are done asynchronously which, contrary to the data parallel approach, does not involve any synchronization barrier. Therefore, the processors are never forced to be idle by the execution system.

The data to be sent are described by the emission rules. Hence, the message generation is a straightforward implementation of the data part of the emission rules.

7. Experimental Results

We have built a multi-threaded parallel program that executes the symbolic static allocation founded in the previous section. We have run it on the *popc* machine of LHPC. The popc (pile of PC) is a high performance cluster based on a modular architecture featuring Intel Pentium Pro processors (200 Mhz, 256 KB of cache L2 memory) interconnected by a fast MYRINET network. We use *BIP* as communication protocol. BIP [19](Basic Interface Protocol), allows the network to reach a 10 μs latency and a 100 Mbytes/s bandwidth.

Size of the matrix	1000	2000	2500
Number of tasks ($n^2/2 + 3n/2 - 2$)	501 498	2 002 998	3 128 748
Number of edges ($n^2 + n - 4$)	1 000 996	4 001 996	6 252 496
Number of float op ($(n^3 + n^2 - 2)/10^6$)	1 001	8 004	15 631
Size in MB for Pyrros	18	78	127

Table 1. Characteristics of the Gaussian Elimination DAG for Various Matrix Size

The multi-threaded API we use is called *PM2* [18]. PM2 (Parallel Multi-threaded Machine), allows the user to create threads remotely on a distance node, it can handle a lot of threads on a same node without high overhead (see figure 4)

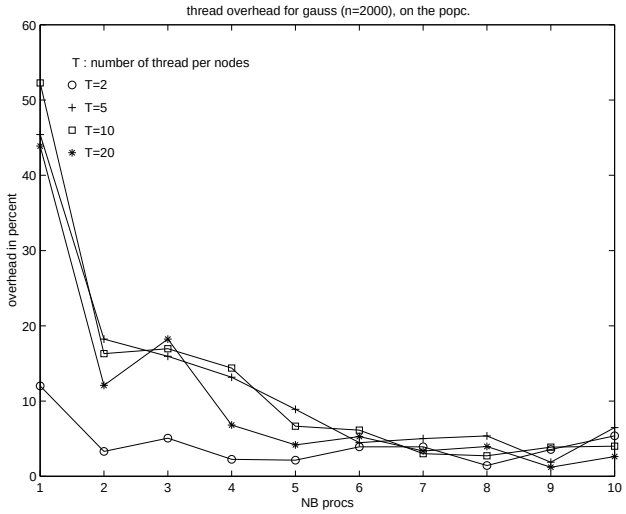

Figure 4. Overhead of the Use of Many Threads on One Node

and finally it can migrate threads easily (which is of no use so far). We have run our generic program for various size of the program parameters and various number of processors (see Figure 5). Table 1 gives the characteristics of the static DAG with respect to the size of the matrix.

Figure 4 shows the overhead percentage of execution time when using more than one thread per nodes. This overhead is important when using only one processor, because there is no communication, but it decreases with the number of processors. It lowers between 6.5% and 2.5% with ten processors. We believe this is mainly due to the overlapping of communication and a good scheduling of the threads. Figure 5 shows the speedup of our program for various size of matrix. We see the high scalability of our method and environment. We have a speedup of 8 for 11 processors when matrix size is 2500.

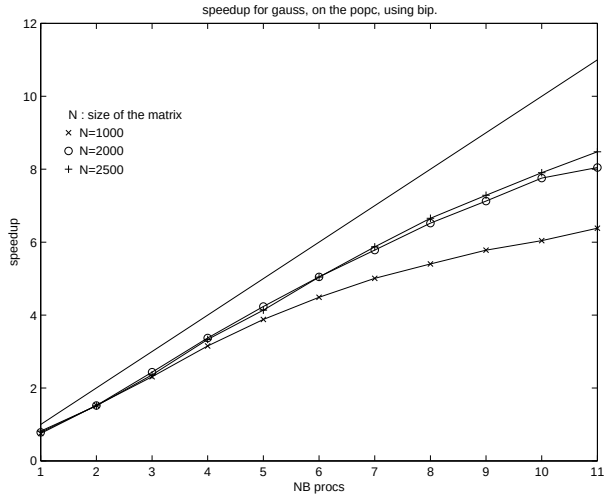

Figure 5. Speedup for Gaussian Elimination for Various Matrix Size

8. Conclusions

In this paper we have presented a method to symbolically schedule a parameterized task graph on distributed memory machines. This approach is useful when dealing with coarse grain applications and when the expanded task graph is too large to fit into memory.

Our contribution is twofold. First, we have modified Feautrier's algorithm for distributing computations among processors in order to take into account conditionals that arise with the PTG model. Our algorithm is then more general and well suited for our problem. Second, we have described a distributed and asynchronous method to schedule and execute our symbolic allocation using a multi-threaded environment.

Our first experiments are very promising and, in our future works, we will realize a code generator that transforms an annotated sequential program into a generic parallel one and executes the symbolic allocation on a distributed memory parallel computer.

9. Acknowledgment

This work is funded by the European Community Eurêka EuroTOPS Project, the NSF/CNRS grant 139812 and the NSF grant INT-95113361.

References

[1] V. S. Adve and M. K. Vernon. A Deterministic Model for Parallel Program Performance Evaluation. (Submited for publication).

[2] S. Amarasinghe, J. M. Anderson, M. S. Lam, and C. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[3] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor Runtime Support for Fine-Grained Irregular DAGs. In R. K. Kalia and P. Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications.*, New York, 1995. Nova Science Publishers.

[4] M. Cosnard, E. Jeannot, and T. Yang. Symbolic Partitionning and Scheduling of Parameterized Task Graphs. Technical Report RR1998-41, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, Sept. 1998. (`www.ens-lyon.fr/LIP/publis.us.html`).

[5] M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, 1995.

[6] M. Cosnard and M. Loi. A Simple Algorithm for the Generation of Efficient Loop Structures. *Internationnal Journal of Parallel Programming*, 24(3):265–289, June 1996.

[7] A. Darte and F. Vivien. Parallelizing Nested Loops with Approximation Distance Vectors: a Survey . *Parallel Processing Letters*, 7(2):133–144, 1997.

[8] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sunderam-Stukel, H. Wasserman, V. Adve, R. Bagrodia, J. Browne, E. Houstis, O. Lubeck, J. Rice, P. Teller, and M. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. In *Proceedings of the First International Workshop on Software and Performance (to appear)*, Santa Fe, USA, Oct. 1998.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *Internationnal Journal of Parallel Programming*, 20(1):23–53, 1991.

[10] P. Feautrier. Toward Automatic Distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[11] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *Proceedings of ACM/IEEE Supercomputing'96*, Pittsburgh, Nov. 1996.

[12] C. Fu and T. Yang. Space and time efficient execution of parallel irregular computations. In *sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'97)*, Las Vegas, June 1997.

[13] A. Gerasoulis, J. Jiao, and T. Yang. Scheduling of Structured and Unstructured Computation . In D. Hsu, A. Rosenberg, and D. Sotteau, editors, *Interconnections Networks and Mappings and Scheduling Parallel Computation*, pages 139–172. American Math. Society, 1995.

[14] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Direct Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[15] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, Jan. 1988. ACM SIGACT-SIGPLAN.

[16] Y.-K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[17] M. Loi. *Construction et exécution de graphe de tâches acycliques à gros grain*. PhD thesis, Ecole Normale Supérieure de Lyon, France, 1996.

[18] R. Namyst and J.-F. Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo'95)*, pages 279–285. Elsevier Science Publishers, Sep 1995.

[19] L. Prylli and B. Tourancheau. Bip: a New Protocol Designed for High Performance Networking on Myrinet. In *Workshop PC-NOW IPPS/SPDP'98*, Orlando, USA, Apr. 1998.

[20] W. Pugh. The omega test a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, Aug. 1992. (website: `http://www.cs.umd.edu/projects/omega`).

[21] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Systems*, 2(4):472–482, 1991.

[22] V. Sarkar. *Partitionning and Scheduling Parallel Program for Execution on Multiprocessors*. MIT Press, Cambridge MA, 1989.

[23] A. Schrijver. *Theory of linear and integer programming*. John Wiley & sons, 1986.

[24] X. Tang and G. R. Gao. How "Hard" is Thread Partitioning and How "Bad" is a List Scheduling Based Partitioning Algorithm? In *Proceedings of the tenth ACM Symposium on Parallel Algorithms and Architectures (SPAA98)*, Puerto Vallarta, Mexico, June 1998.

[25] T. Yang and A. Gerasoulis. DSC Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.