Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs

Michel Cosnard
LORIA - INRIA Lorraine
615, rue du Jardin Botanique
BP 101
54602 Villiers Les Nancy,France
e_mail: michel.cosnard@loria.fr

Emmanuel Jeannot and Laurence Rougeot
LIP, ENS de Lyon
46, allée d'Italie
69364 Lyon cedex 07,France
e_mail: {ejeannot,lrougeot}@ens-lyon.fr

Abstract

Scheduling large task graphs is an important issue in parallel computing since it allows to treat big size problems. In this paper we address the following problem: how to schedule a task graph, when it is too large to fit into memory? Our answer introduces the parameterized task graph (PTG), which is a symbolic representation of the task graph. We propose a dynamic scheduling algorithm which takes the PTG as an entry, and allows to generate a generic program. Performances of the method are studied as well as its limitations. We show that our algorithm finds good schedule for coarse grain task graphs, has a very low memory cost, and has good computational complexity. When the average number of operations of each task is large enough, we prove that the scheduling overhead is negligible with respect to the makespan. The feasibility of our approach is studied on several compute-intensive kernels found in numerical scientific applications.

1. Introduction

As the computational power of distributed memory parallel computer increases the need to solve very large problems appears. In the task parallelism approach, computations are allocated onto processors and there is one control flow per processor which is data driven. The *task graph* is a model well suited for such an approach. The task graph is a DAG where each node is a sequential task, and where edges correspond to dependences between tasks, mostly due to communication. A task is a set of sequential instructions that must be executed on one processor.

In the literature a lot of work has been done to schedule such task graphs [5], either allowing duplication of tasks on the processors [19, 20], or not [11, 17]. In order to generate a parallel program, static schedulers, like Pyrros [24], need to have the complete task graph in memory. This is not a feasible solution when dealing with very large problems because the task graph is too large to fit into memory. Furthermore, a task graph is built when all the parameters have been instantiated. Thus, if the parameters change, the analysis of the sequential program has to be repeated and a new task graph is then, rebuilt. Hence, static task graph scheduling does not allow to build a generic program. In *Cilk* [3], the task graph is scheduled at run-time. Cilk can handle big size problems but, communication cost is not taken into consideration. The Cilk system performs very well for tree-like style computation (min-max search, backtrack exploration, etc...) but, it has not been designed for scientific loop-nest computations. In [2, 13] run time methods to schedule task graphs are described addressing the problem of processor memory requirement, but these works do not consider DAG memory requirement. In [1], a tool *CASCH*, is presented. It allows to generate a schedule and a parallel code for a sequential program. Nevertheless CASCH uses standard static scheduling algorithms and consequently, has the same drawbacks as Pyrros.

In this paper we present and study a new approach that allows to solve very large problems (of order of a million tasks). This is a complete automatic parallelization line for most of the compute-intensive kernels found in numerical scientific applications. The input is an annotated fortran-like sequential program. A tool, PlusPyr [18], generates an intermediate program representation called the parameterized task graph (PTG) [9, 10]. The PTG is a compact program representation for DAG parallelism, and it requires a small amount of space to store because it is independent of program sizes. Once the parameters value is known, it is possible to use the parameterized task graph to build the task graph. We will not consider this possibility in the following. We propose a different approach which consists in building a generic program that schedules dynamically the task graph. This method requires the parameters value to be given at run time. The parameterized task graph dynamic scheduler (PT-

GDS), which has been studied in [7, 8], is a new algorithm which uses the parameterized task graph to explore the DAG and schedule the tasks. During the scheduling, the key point is that, at any moment, PTGDS has only a small part of the task graph in memory. This paper is focussed on the feasibility of such an approach. We study the parallelization of several computation intensive kernels. We show that the use of the parameterized task graph allows to schedule large problems. We discuss the limitation of our approach and derive some required conditions such that the scheduling overhead is negligible with respect to the parallel time.

This paper is organized as follows: Section 2, gives the definitions used along this paper. Section 3 deals with the PlusPyr software. In Section 4 we describe the dynamic approach and the PTGDS algorithm. Section 5 studies theoretical results concerning PTGDS behavior and our dynamic approach. Section 6 presents the experimental results we have obtained. Finally, in section 7 concluding remarks are given.

2. Definitions and models

All over this paper, we will use the following general definitions:

- A *task graph*, is an annotated directed acyclic graph (DAG), defined by the tuple $G = (V, E, T, C)$. V is the node set, each node representing a task. $v = |V|$ is the number of nodes. In the following we will say node or task indifferently. E is the edge set. There is an edge from task i to task j if there is a dependence between task i and task j (i.e. task j must be executed after the end of task i). $e = |E|$ is the number of edges. T is the task weight (or task duration) set, $\tau_i \in T$ will represent task i duration. C is the set of edge weights (or communication volumes), $c_{i,j} \in C$ represents the communication cost along the edge from node i to node j. It becomes 0 if the two tasks are mapped on the same processor.

- We call $g(G)$ the *granularity* of the task graph G. We use the definition given by Gerasoulis and Yang in [15]:

$$g(G) = \min_{i=1:v} \left(\min \left(\frac{\tau_i}{\max_{j \in pred(i)} c_{j,i}}, \frac{\tau_i}{\max_{j \in succ(i)} c_{i,j}} \right) \right)$$

- A task graph G is said *coarse grain* if $g(G) >= 1$, i.e. if all the communication costs of any task are smaller than the task duration.

The execution model for task graphs is called macro data flow. Each task first receives all the needed data, computes without interruption and then sends the results to its successors. We do not allow task duplication, because duplication results in an increase of the memory required by the scheduler. This is incompatible with one of our goals which is to have a low memory cost algorithm.

In this paper we do not discuss issues concerning the topology of the target machine. We will suppose to deal with a clique of processors. We shall recall here that, in the wormhole routing mode, which is widely used now, the communication cost is not very sensitive to the processor distance, except if contention is high[4]. Therefore we shall use the following parallel computer model:

- p is the number of processors (they are all identical).

- ω is the time taken by the execution of one elementary instruction. If s_i is the number of operations performed by task i (*the computational cost* of task i), we have $\tau_i = \omega s_i$.

- β is the startup time of a communication, and α is the transmission rate. If $l_{i,j}$ is the number of items sent from task i to task j, $c_{i,j} = \beta + \alpha l_{i,j}$.

3. PlusPyr and the parameterized task graph

We have proposed the parameterized task graph [9] as a solution for automatically deriving task graphs from sequential programs. It uses parameters which have to be instantiated in order to build the expanded task graph. It is mainly composed of *generic tasks code* and *communication rules*. A generic task is a set of instructions which have to be executed sequentially. The communication rules represent symbolic dependences between generic tasks.

The communication rules have two forms. Reception rules define data received by generic tasks. Emission rules describe the data sent by a generic task.

They also give the symbolic value of the communication volume along the generic edge .

As opposed to the *parameterized task graph*, in the following, we shall call *the expanded task graph*, or simply *the task graph*, the DAG built when values are given to the parameters.

PlusPyr [18], a tool built by Michel Loi, is able, given a sequential program, to derive the parameterized task graph. PlusPyr is also able, once the values of the parameters are given, to construct the expanded task graph. There are some limitations concerning the input language, for more details see [9]. The analysis performed by PlusPyr gives also the *symbolic computational cost* of each generic task. This analysis is based on the work done by Feautrier on integer parametric programming (see [12, 22] for an introduction) and [9, 10], for more details.

4. Parameterized task graph dynamic execution

In the literature, dynamic execution policy is mainly used in the following two cases : (1) dealing with nondeterministic programs [3], (2) load balancing and process migration [21, 23]. Here we use dynamic execution for the following reasons: (1) to obtain a generic parallel program for each sequential program : we give the parameters value at run-time, and then schedule the tasks during the execution. We use the parameterized task graph because it is problem size independent, (2) to handle big size problems. It is impossible to have a very large task graph in memory. The parameterized task graph allows to build at run-time, the small part of the DAG needed to perform task mapping optimization.

4.1. PTGDS, the scheduling algorithm

```
1  schedule(task T){
2  for each task T' in father(T) do
3    if not(allocated(T')) then schedule(T');
4  endfor
5  allocate T to the processor that minimizes its starting time;
6  for each task T' in father(T) do
7    if T is the last son of T' to be scheduled then
8      remove from memory all information on T'
9    endif
10 endfor
11 allocated(T)=true;
12 }
```

Figure 1. The PTGDS algorithm

PTGDS schedules the tasks: it determines on which processor and when each task has to be executed. For each task T, when the parameters value is known:

- We use the code analysis performed by *PlusPyr* to determine the duration of T.
- We use the parameterized task graph, to determine the set of the children of T, the set of the parents, and the communication volume between parents or children. Most of the time in the PTG the set of sons/fathers of T is described by a polyhedron. In the program, in order to have all the sons/fathers, we generate a loop nest that scans all the points of this polyhedron. We use a tool called *enum* made at the Université de Rennes [14] that transforms a parameterized polyhedron into a loop nest. The number of items sent between two tasks is also described by a polyhedron. In the program we generate a parameterized polynomial that enumerates the number of points in the polyhedron, i.e. the number of items. We use the *polylib* of the Université de Strasbourg [6], to build such a polynomial.

Figure 1 gives the general scheme of PTGDS algorithm.

PTGDS starts from a node which topologically is a descendant from all the other tasks ("*the output task*"). It recursively explores the DAG and, for each task T, it schedules all the fathers of T before allocating T to a processor. We also add a source node (the "*input task*") that is the predecessor of all the node in the DAG. Th input task is the only source node in the DAG, and is always schedule to the processor 1 at time 0.

Lines 7 and 8 are justified as follows: each time a task T is scheduled we put *allocated(T)=true*. Thus, we need a data structure (in our case an AVL tree [16]), to store all the values of *allocated(T)*. When all the sons of task T have been scheduled, the test on the line 3 will never be performed again for T. Then, we can remove from the AVL all the informations about T. Hence, lines 7 and 8 allow a major reduction of the memory used during execution of the algorithm.

4.2. Combining scheduling and dynamic execution

The target code is an SPMD generic program which uses a supervisor/executor protocol. One supervisor executes PTGDS, and sends orders to the processors. There is one executor per processor. We assume that the supervisor is executed on the parallel machine host. Each executor receives messages from the supervisor. Those messages may to execute a task or to send data to an other processor. For instance, when a task T is assigned to processor P, the supervisor orders the executors to send to P the data needed for the computation of T. Receptions of messages are handled independently by each executor. The execution is dynamic, since the supervisor sends orders to the executors while scheduling the task graph. Such a complete dynamic execution of the parameterized task graph will be called PTGDE (Parameterized Task Graph Dynamic Execution). Figure 2 gives the code of the supervisor and one executor. All the executors have the same code: the sequential generic task, and the communication protocol.

Each executor manages two lists. In *exec_list* are stored the execution messages that cannot be performed because the data are not yet present in the local processor memory. This list is updated when new data arrive or new data are computed. In *send_list* are stored the send messages that cannot be performed because the data have not already been computed by the processor. This list is updated when new data are computed.

5. Minimizing the impact of the scheduling overhead

In this section we show that it is possible to derive a simple upper bound on the average number of operations of

```
schedule(task T){
for each task T' in father(T) do
  if not(allocated(T')) then schedule(T');
endfor
send the order of executing T to the executor which, according to PTGDS, minimizes its starting time;
send, to executors which execute the fathers of task T, the order of transmitting data to the executor which execute T;
for each task T' in father(T) do
  if T is the last son of T' to be scheduled then
    remove from memory all information on T'
  endif
endfor
allocated(T)=true;
}
```

```
while true do{
 case type_message of
  execute :
   if all the required data are in memory
     then execute the task; send new computed data if required (check send_list);
          If possible execute a task with the new data computed (check exec_list);
     else store this message in exec_list;
  send :
   if all the required data are in memory
     then send the data;
     else store this message in send_list;
  receive :
   store the data in memory;
   execute task if possible (check exec_list);
}
```

Figure 2. PTGDE code. Left: the supervisor; Right: an executor

tasks in order to obtain an efficient execution on a parallel computer.

In [7, 8] we have shown that the computational complexity of PTGDS is $O((e+v)(\log v + p))$, and that on an unbounded number of processors for any DAG G:

$$PT_{opt}(G) \leq PT_P(G) \leq (1 + \frac{1}{g(G)}) PT_{opt}(G)$$

where $PT_P(G)$ is the parallel time found by PTGDS and $PT_{opt}(G)$ is the optimal parallel time.

Definition 1 *Let us call A_{op}, the average number of operations of tasks in G, i.e. $A_{op} = \frac{T_{seq}}{v\omega}$. Let $PTGDE(G,p)$ be the parallel program execution time, of the parameterized task graph G scheduled by PTGDS, on a p processors parallel computer. Let $T_{sup}(G,p)$ be the time taken by PTGDS on the supervisor to schedule G on p processors.*

Lemma 1 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p) + PT_P(G,p)$

Proof of lemma 1: Due to scheduling overhead the time taken by the executor to finish is greater than the total parallel time. Moreover, since simultaneously, PTGDS is executed on the supervisor, and the parallel code is executed on p executors, the total dynamic execution time is lower than the sequentialization of the supervisor time plus the executors time. ∎

In many cases (particularly, in all the section 6.1 examples), the number of edges of the task graph is on the same order as the number of nodes. The main limitation of our approach is that dynamic scheduling is costly when dealing with fine grain task graphs. Theorem 1 gives a sufficient condition on the average number of operation of tasks such that, during the the parameterized task graph dynamic execution, the scheduling time is negligible with comparison to the parallel time.

Theorem 1 *If $e = O(v)$ and $p \max(p, \log v) = o(A_{op})$ then*

$$PTGDE(G,p) = PT_P(G,p) + \epsilon$$

where ϵ is a negligible time with comparison to $PT_P(G,p)$.

Proof of theorem 1 Let $T_{sup}(G,p)$ be the supervisor time.

$$T_{sup}(G,p) = \text{schedule time} + \text{messages time}$$

The maximum amount of communication done by the supervisor is $O(e)$, since it has to communicate to the executors at most each time two processors have to send data

$$T_{sup}(G,p) = O((e+v)(p+\log v) + O(e)$$

Let $m = \max(p, \log v)$, and since $e = O(v)$:

$$T_{sup}(G,p) = O(vm) + O(v)$$

Then, it exists some constant k such that:

$$T_{sup}(G,p) = kvm\omega \quad (1)$$

We cannot have a superlinear speedup, thus:

$$PT_P(G,p) \geq \frac{T_{seq}}{p} \quad (2)$$

By theorem hypothesis, $pm = o(A_{op})$

$$pm = o(A_{op}) \quad \Rightarrow \quad pm = o(\frac{T_{seq}}{v\omega})$$
$$\Rightarrow \quad kvm\omega = o(\frac{T_{seq}}{p})$$

Replacing equation 1 and 2 leads to $T_{sup}(G,p) = o(PT_p(G,p))$. Since, according to lemma 1 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p) + PT_P(G,p)$, then

$$PTGDE(G,p) = PT_P(G,p) + \epsilon$$

■

6. Results and experiments

6.1. The programs examples

We present here the set of program examples we have use for our experiments. We have done extensive tests on the parallelization of this numerical kernels.

The Gaussian Elimination: this program is composed of two generic tasks: one for the computation of the pivot column, one for the update of the submatrix.

Gaussian Elimination and backward substitution: this program has two generic tasks: one for the "ijk" Gaussian Elimination, and a second one for solving the system by a backward substitution. This shows that we can easily analyze the concatenation of two programs.

The Givens algorithm: we have a simple program, with only one generic task. The main loop is composed of many instructions: it shows that complex computational costs can be handled.

Jordan Diagonalization: Jordan method is decomposed into two generic tasks, one to diagonalize the matrix and the other one to solve the system. In the two generic tasks, for-loops are sequentialized .

Power of a matrix: We compute the m^{th} power of matrix A of order n. . In this example, 3 parameters (n,m and l) are used. Hence, the computational cost and the granularity can be easily tuned.

For all this examples we have performed some experiments to validate our approach. First, we have done speedup simulations , memory cost measurement and scheduling overhead comparison to the parallel time.

All the tests are done with the following values (found on Sparc 5 workstation linked by Ethernet on PVM : $\alpha = 16\,\mu$s/double, $\beta = 2.5$ ms and for one operation on a double $\omega = 0.882\,\mu$s.

6.2. Speedup simulation

In Figure 3, the simulations show that when the matrix size is large the processors utilization is important, even for large machine.

6.3. Memory cost

In Table 6.3 we study the memory cost of PTGDS for several task graphs. We have instantiated parameters value and run PTGDS on the expanded graph. The column *# Tasks*, is the number of tasks in the DAG. PTGDS optimizes the number of tasks in memory. The list of tasks needed to allocate other tasks is handled by an AVL tree. Column *Max # tasks* is the maximum size of the AVL during the execution. The parameterized task graph is a description of the edges in the task graph. sometime an edge in the task graph is generated by several rules. Hence, when we use the communications rule to explore the DAG, the program generates more edges and tasks than there are in the DAG. Our program ensures the schedule correctness by checking duplicated edges and tasks. However, it is necessary to use all the rules in order to count the amount of data sent between tasks. Column *# Nodes* gives the number of tasks generated by the recursive DAG exploration. Column *Max # nodes* is the maximum number of tasks in memory during the execution. Column *# Edges* is the number of edges generated by the recursive DAG exploration. Column *Max # edges* is the maximum number of edges in memory during the execution. Edges and tasks are removed from memory when the corresponding recursive step is finished. These three data structures are the only one which vary with the parameters value.

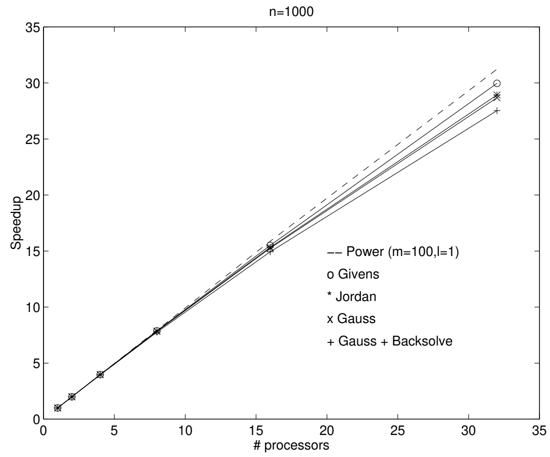

Figure 3. Speedup simulation vs. number of processors for several examples

Table 6.3 shows that the memory required to schedule the task graph is only a small portion of the total memory required by the whole task graph. The differences between $n = 100$ and $n = 1000$ show that required memory increases linearly, while (except for *Power*), the size of the DAG increases quadratically.

6.4. Comparison between scheduling overhead and execution time

In this section we show that, according to theorem 1, the

Program	# Tasks	Max # tasks	# Nodes	Max # nodes	# Edges	Max # edges
			$n = 100$			
Gauss	5150	200	20593	199	15348	398
Givens	4952	295	24856	198	14851	494
Gauss & BS	5052	201	30501	102	20100	303
Jordan	5052	200	40394	101	15052	301
Power (m=100)	10002	9805	30011	102	39802	10100
			$n = 1000$			
Gauss	501500	2000	2005993	1999	1503498	3998
Givens	499502	2995	2498506	1998	1498501	4994
Gauss & BS	500502	2001	3005001	1002	2001000	3003
Jordan	500502	2000	4003994	1001	1500502	3001
Power (m=100)	99202	98005	299111	102	396202	99200

Table 1. Memory cost of main data structures of PTGDS, for various matrix size

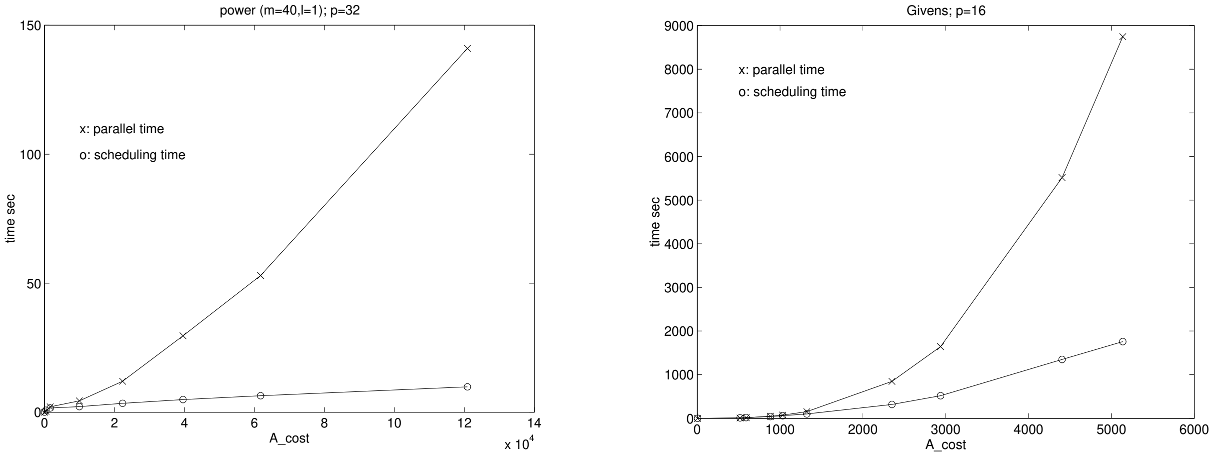

Figure 4. Execution time of PTGDS and execution time simulation vs. average number of operations of tasks

time taken by PTGDS to schedule the parameterized task graph is negligible when the average number of operations of tasks is big. In Figure 4 we have compared the time that would take *Power* and *Givens* to execute on a 32 processors machine having the characteristics given in this section, and the time taken to schedule the DAG, when the average number of operations of tasks increases. For *Power* we see that when $p^2 \approx 50.A_{op}$ then $PT(G) \approx 10.PTGDS(G)$. For the Givens algorithm we see that when $p^2 \approx 20.A_{op}$ then $PT(G) \approx 5.PTGDS(G)$.

7. Conclusion

In this paper we have presented a scheme of a complete line of automatic parallelization for some kind of programs. This work is based on the PlusPyr tool which builds the parameterized task graph. We have conducted experiments on various programs. We have given conditions when this approach is valid. Theoretical results show that PTGDS finds good schedule on an unbounded number of processors for coarse grain task graphs, and has a competitive computational complexity. In this paper, we have given a theoretical bound for the average number of operations a task should perform if we want the schedule overhead to be negligible in comparison to the parallel time. Our experiments show that, (1) PTGDS finds good schedules on a fixed number of processors, (2) the algorithm memory cost is low, so we can handle very large task graphs, (3) for some programs, with tasks large enough, PTGDS execution time is small with regards to the makespan. Thus, it is possible to schedule dynamically such programs.

In conclusion, we are doing parameterized task graphs dynamic scheduling in order to have generic programs and to deal with large task graphs. This method appears to be efficient on coarse grain task graphs, when the average number of operations of the tasks is high, as for instance, for block algorithms.

Future works is to the realize the code generator, and to study a new method to schedule statically parameterized task graphs.

8. Acknowledgments

This work is part of the European Community Eurêka EuroTOPS project. We thank Michel Loi for providing us with the PlusPyr software, Tao Yang of UCSB for very helpfull discussions about this paper and the anonymous referees for valuable comments and suggestions.

References

[1] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu. Automatic Parallelization and Scheduling on Multiprocessors using CASCH. In *ICPP'97*, Aug. 1997.

[2] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95*, Santa Barbara, California, July 1995.

[4] S. Chintor and R. Enbody. Performance Degradation in Large Wormhole-Routed Interprocessor Communication Networks. In *Proceedings of ICPP'90*, volume I, pages 424–428, 1990.

[5] P. Chretienne and C.Picouleau. *Scheduling Theory and its Applications*, chapter 4, Scheduling with Communication Delays: A Survey, pages 65–89. John Wiley and Sons Ltd, 1995.

[6] P. Clauss, V. Loechner, and D. K. Wilde. Deriving Formulae to Count Solutions to Parameterized Linear Systems using Ehrhart Polynomials: Applications to the Analysis of Nested-Loop Programs. Technical report, Université de Strasbourg, April 1997. research report RR 97-05 *http://icps.u-strasbg.fr/pub-97/pub-97-05.ps.gz.*

[7] M. Cosnard and E. Jeannot. Automatic Coarse-Grained Parallelization Techniques. In Grandinetti and Kowalik, editor, *NATO workshop : Advances in High Performance Computing*. Kluwer academic Publishers, 1997.

[8] M. Cosnard and E. Jeannot. Building and Scheduling Coarse Grain Task Graphs. Technical Report RR97-03, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, Feb. 1997.

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

[10] 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.

[11] H. El-Rewini, T. Lewis, and H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.

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

[13] 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.

[14] M. L. Fur. *Compilation de boucles dirigé par la distribution des donnés*. PhD thesis, Université de Rennes I, July 1995.

[15] 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.

[16] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of data structures in C*. W.H. Freeman and company, New-York, 1993.

[17] 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.

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

[19] M. Palis, J.-C. Liou, and D. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, Jan. 1996.

[20] C. Papadimitriou and M. Yannakakis. Toward an Architecture Independent Analysis of Parallel Algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[21] C. Perez. Load Balancing HPF Programs by Migrating Virtual Processors. In *Second International Workshop on High-Level Programming Models and Supportive Environments, HIPS97*. IEEE Computer Society Press, Apr. 1997.

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

[23] N. Shivarati, P. Krueger, and M. Singhal. Load Distributing for Localy Distributed Systems. *Computer*, 25(12):33–44, Dec. 1992.

[24] T. Yang and A. Gerasoulis. Pyrros: Static Task Scheduling and Code Generation for Message Passing Multiprocessor. In *Supercomputing'92*, pages 428–437, Washington D.C., July 1992. ACM.