Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Building and Scheduling Coarse Grain Task Graphs

M. Cosnard
E. Jeannot

February 1997

Research Report N$^{\text{o}}$ 97-03

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Building and Scheduling Coarse Grain Task Graphs

M. Cosnard
E. Jeannot

February 1997

Abstract

It has been shown recently that it is possible to efficiently schedule direct acyclic task graphs. A tool like Pyrros [14] is able to generate a parallel program, if the user specifies precedence constraints between the tasks. This approach has two major drawbacks: it is hard for the user to perform a good analysis of the source code and scheduling a DAG is costly in terms of memory and time.

In [3], a new model of parallel computation, the parameterized task graph is introduced. The Parameterized Task Graph (PTG) is a compact, problem size independent representation of some frequently used direct acyclic task graphs. In [4, 10], techniques and a tool (PlusPyr), are designed to generate such a representation from an annotated sequential program. In this report, we review the PTG and associated techniques, we give a general framework and an algorithm to dynamically schedule the parameterized graph and generate the associated code.

Keywords: Automatic parallelization, task graph, affine dependences, static scheduling, dynamic allocation.

Résumé

On a montré récemment qu'il est possible d'ordonnancer efficacement un graphe de tâches acyclique. Un outil comme Pyrros [14] est capable de générer un programme parallèle si l'utilisateur spécifie les contraintes de précédence entre les tâches. Cette approche a deux défauts majeurs : il est difficile pour l'utilisateur de faire une bonne analyse du code source et ordonnancer un graphe acyclique est coûteux en termes de temps et de mémoire.

Dans [3], un nouveau modèle de calcul parallèle, le graphe de tâches paramétré a été introduit. Le GTP est une représentation compacte, indépendante de la taille du problème, de quelques graphes de tâches acycliques orientés fréquemment utilisés. Dans [4, 10], des techniques et un outil (PlusPyr) sont construit pour générer une telle représentation à partir d'un programme séquentiel annoté. Dans ce rapport, nous revoyons le GTP et les techniques associés, nous donnons une structure générale et un algorithme pour ordonnancer dynamiquement le graphe paramétré et générer le code associé.

Mots-clés: Parallélisation automatique, graphe de tâches, dépendances affines, ordonnancement statique, allocation dynamique.

Building and Scheduling Coarse Grain Task Graphs*

M. Cosnard, E. Jeannot
Laboratoire de l'Informatique du Parallélisme, CNRS,
Ecole Normale Supérieure de Lyon, 69364 Lyon FRANCE
{cosnard,ejeannot}@lip.ens-lyon.fr

February 14, 1997

1 Introduction

In the field of automatic parallelization, two issues are mainly discussed. The first issue consists in mapping the data among the processors of the machine and then placing the computations where the data needed to perform them are located. This is done, for instance, by the fine grain techniques, dealing mostly with loop nests. It is also the approach taken by HPF [1, 13]. The second approach, which is ours, puts the computations onto the processors and let the data flow along the network of the machine, to the processors. This technique is most suitable for coarse grain computation, when the computations are bigger than the communications, and has been successful in the domain of regular problems such as those encountered in dense linear algebra. It is now possible, given a task graph, to automatically schedule and map the tasks, and then build a parallel program for distributed memory parallel computer [14]. In that case, the user has to analyze the program in order to find the dependences, and then build the task graph. Such analysis is long, hard and source of many errors.

In this paper, we present a complete line of automatic parallelization for most of the compute-intensive kernels found in numerical scientific applications. The user gives an annotated Fortran program as input. PlusPyr, [10], generates an intermediate model called the parameterized task graph, [3, 4], which is a problem size independent representation of the program. Then, one possibility is to instantiate the parameters and then build the task graph. This task graph can be scheduled by static scheduler [10]. This leads to two remarks. First, since big size problems imply big task graphs, it is high memory costly to schedule such big problems. Second, it is not possible to derive a generic program from a static graph, this implies to repeat the analysis and the schedule if the values of the parameters change. In order to avoid those two problems, we propose here, a new method which consists in building a generic program that schedules dynamically the task graph. This method requires the values of the parameters to be given at run time. We present a scheduling algorithm called PTGDS that has only a small part of the complete DAG in memory at a given time of the execution, and that explores the DAG using the parameterized task graph. We show this algorithm has a time complexity of $O((e+v)(\log v+p))$, is low memory costly,

*This work has been supported by the Region Rhône-Alpes and by the Eureka EuroTops project funded by the French Ministry of industry.

and is within a bound of two of the optimal schedule on an unbounded number of processors (for coarse grain DAG).

This paper is organized as follows. Section 2, gives the definitions used along the paper. Section 3, deals with the parameterized task graph, and the PlusPyr tool. Section 4, discusses the issue of static scheduling. Section 5, describes the general framework of the dynamic scheduling, including PTGDS algorithm. Section 6, gives the results concerning the performances of PTGDS. Finally, section 7 shows the experiments we have done. In section 8, conclusions and future works are given.

2 Definitions

We will use all along this paper the following general definitions:

- A *direct acyclic task graph*, is defined by the tuple $G = (V, E, T, C)$. V is the set of nodes, each node representing a task. In the following we should say node or task indifferently. E is the set of edges. There is an edge from task i to task j if there is dependence between task i and task j (i.e. task j must be executed after the end of task i). T is the set of task weights (or task durations), $\tau_i \in T$ will represent the duration of task i. 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 [6]:
$$g(G) = \min_{i=1:v}(\tau_i / \max_j c_{i,j})$$

- We call *cluster* a group of tasks which are going to be executed on the same processor, i.e. for any cluster C, if task i and task j are in C then $c_{i,j} = 0$.

- We call a cluster *non linear* if there exist two tasks in this cluster which are independent in G, otherwise we call it *linear*.

- A *clustering* is a mapping of all the tasks of G into clusters. The clustering is called linear if all the clusters are linear otherwise it is called non-linear. Clustering algorithms have been first proposed by Sarkar in [11]. Gerasoulis and Yang have discussed the impact of granularity for clustering algorithms in [6]

The model of execution for acyclic task graphs is called macro data flow. For each task, it first receives all the needed data, computes without interruption and then sends the results to its sons.

We do not allow task duplication, because duplication results in an increase of the memory required. This is incompatible with one of our goals which is to have a low memory cost algorithm.

In this paper we do not discuss the issues concerning the architecture of the target machine. We will suppose to deal with a clique, hence the communication cost will be just a function of the size of the message. We shall recall here that, in the wormhole routing mode, which is widely used now, the communication cost is not very sensible to the architecture of the machine, except if contention is high[2].

3 The parameterized task graph

The parameterized task graph has been proposed by Cosnard and Loi [3] as a solution for automatically deriving task graphs from sequential programs involving the use of parameters that has to be instantiated at execution time. It is beyond the scope of this paper to present a detailed description of the parameterized task graph. We shall only give a brief description of this model. Let us present first the input language to be used by our automatic task graph builder, called PlusPyr [10].

```
param n
real a(n,n+1)
for k = 1 to  n-1 do
 task /*T1*/
  for l = k + 1 to n do
   a(l,k)=a(l,k)
          /a(k,k) /*S1*/
  endfor
 endtask
 for j = k + 1 to n+1 do
  task /*T2*/
   for i = k + 1 to n do
    a(i,j)=a(i,j)-a(k,j)
           *a(i,k)/*S2*/
   endfor
  endtask
 endfor
endfor
```

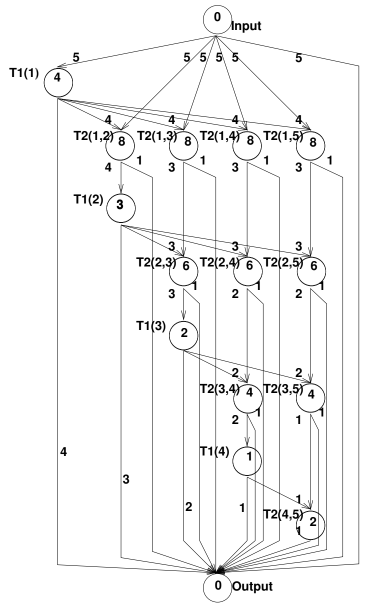

Figure 1: Gaussian Elimination explicit task graph

3.1 The input program model

Variable data types are restricted to simple types, and n-dimensional arrays of these types. We restrict also the control structure to *static* FOR-loops. A statement is a variable assignment to an expression. A task specification construct is also provided. The user (or an automatic preprocessor) may use it in order to group statements into atomic execution units. Hence, a task is nothing else than a set of statements. Figure 1 shows the source program for the kji form of Gaussian Elimination in which two tasks T_1 and T_2 are defined.

We assume that any valid input program complies with the following rules :

- Task constructs cannot be nested.

- Every assignment statement is lexically included in a task construct.
- Loop counter names and parameter names may not be used as the left hand side of an assignment.
- All the array subscript expressions and FOR-loop bound expressions are integer-valued affine functions of program parameters, integer constants and enclosing loop indices.

3.2 Task definitions

A parameterized task graph is a set of statement definitions (iteration vector, englobing task, enclosing loop bounds), a set of task definitions (iteration vector, first and last statement) and a set of communication rules. We shall not enter into a precise definition of what is a task nor a precise semantic of a communication rule (for more detail see [3]). One should simply know that a task is a set of statements, and its iteration vector describes the parameters of this task that will vary during the execution of the program. For instance, in the program in figure 1 there are two tasks $T1$ and $T2$, and $T2$ has (k, j) as iteration vector since it is enclosed in loops k and j. The communication rules are composed of emission and reception rules, which describe symbolically generic communications between tasks. For each task, they define which data will be sent to which task, and which data will be received from which task.

Two more tasks are always added to the set of tasks defined by the user. One task is *T-INPUT*, which is the first task to be executed, it writes all the variables of the program and hence is used to input the values of all the data. The other task is *T-OUTPUT*, which is the last task to be executed and reads all the variables, hence is used to grab the results.

3.3 Example of communication rules

Figure 2 shows an example of the reception rules derived from the program in figure 1. Let take into consideration rule number 9. It says that

$$\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T1(k)\} : \{A(i,k)|k+1 \leq i \leq n\}$$

This means that task $T2(k, j)$, for all $2 \leq k \leq n-1$ and all $k+1 \leq j \leq n+1$, will receive from $T1(k)$ the pivot column $A(i, k)$ for all $k+1 \leq i \leq n$.

All these rules describe symbolically the communications handled by tasks *T-INPUT*, *T1*, *T2*, and *t-OUTPUT*, during the execution of the program. They also give the symbolic value of the communication volume along the generic edge (for rule 9 it is $n-k$).

3.4 Building the parameterized task graph

Michel Loi [10] has build a tool (*PlusPyr*) that is able, given an annotated sequential program, to analyze it and then derive the parameterized task graph, and the symbolic computation cost of each task. These analysis are based on the work done by Feautrier on integer parametric programming (see [5, 12] for an introduction). We shall not discuss here the complete method used by PlusPyr, (for more details see [3, 4]).

1. $\{T1(k)|k=1\} \leftarrow \{T-INPUT\} : \{A(k,k)\}$
2. $\{T1(k)|k=1\} \leftarrow \{T-INPUT\} : \{A(l,k)|k+1 \leq l \leq n\}$
3. $\{T1(k)|2 \leq k \leq n-1\} \leftarrow \{T2(k-1,k)\} : \{A(k,k)\}$
4. $\{T1(k)|2 \leq k \leq n-1\} \leftarrow \{T2(k-1,k)\} : \{A(l,k)|k+1 \leq l \leq n\}$
5. $\{T2(k,j)|k=1, k+1 \leq j \leq n+1\} \leftarrow \{T-INPUT\} : \{A(k,j)\}$
6. $\{T2(k,j)|k=1, k+1 \leq j \leq n+1\} \leftarrow \{T-INPUT\} : \{A(i,j)|k+1 \leq i \leq n\}$
7. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T2(k-1,j)\} : \{A(k,j)\}$
8. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T2(k-1,j)\} : \{A(i,j)|k+1 \leq i \leq n\}$
9. $\{T2(k,j)|2 \leq k \leq n-1, k+1 \leq j \leq n+1\} \leftarrow \{T1(k)\} : \{A(i,k)|k+1 \leq i \leq n\}$
10. $\{T-OUTPUT\} \leftarrow \{T-INPUT\} : \{A(i,j)|j+1 \leq i \leq n\}$
11. $\{T-OUTPUT\} \leftarrow \{T1(j)|1 \leq j \leq n-1\} : \{A(i,j)|j+1 \leq i \leq n\}$
12. $\{T-OUTPUT\} \leftarrow \{T2(i-1,j)|2 \leq i \leq n, i \leq j \leq n+1\} : \{A(i,j)\}$

Figure 2: Reception rules for the Gaussian Elimination

4 Static scheduling

Once the parameterized task graph (PTG) is built, it is very easy, giving values to the parameters, to derive the direct acyclic task graph of the application [10]. That representation of the source code can now be used, by static schedulers like *Pyrros* [14], to schedule and to map the tasks, and then to produce a parallel code. Pyrros is a tool built at Rutgers University by Gerasoulis and Yang. They have currently integrated PlusPyr as a front end of Pyrros in a tool called *Pyrros+* [8]. Figure 3 shows how it is connected to PlusPyr. Pyrros uses DSC as scheduling algorithm. DSC (Dominant Sequence Clustering [15]), is a clustering algorithm which has a computational complexity of $O((e+v)(\log v)$ and a space complexity of $O(e+v)$. In [9], Kwok and Ahmad present and describe various static scheduling algorithms. They propose a new static scheduling algorithm: DCP (Dynamic Critical Path) which, despite his efficiency, has a high computational complexity of $O(v^3)$.

5 Dynamic scheduling of parameterized task graphs

We present a general framework to schedule dynamically the PTG. This work originates from two remarks:

1. DSC, in Pyrros, requires the program to be expressed as a static direct acyclic task graph, which is memory costly. In the Gaussian elimination of a matrix of order 1000, the associated task graph as

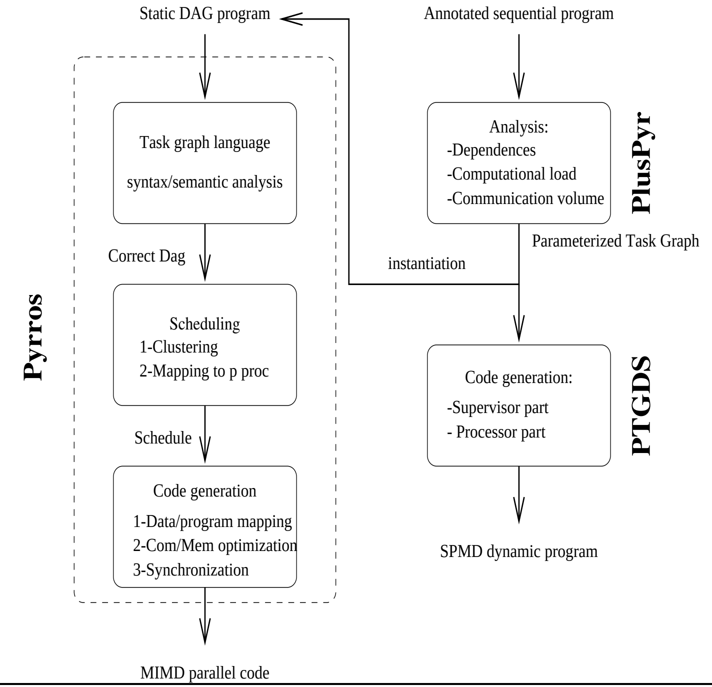

Figure 3: Pyrros and PlusPyr schemes

about 500000 nodes and 1000000 edges. With 20 bytes for each node and 12 bytes for each edge, the graph requires at least 22 M-bytes of memory,

2. it is necessary to know parameters values at compile-time to use Pyrros. But those values may not be known at compile-time, or in a case of a generic program, it is required to give those values at run-time.

Our dynamic method is based on the parameterized task graph, and hence solves those two problems. As a matter of fact, a PTG is problem-size independent, and only needs the parameter values at run-time.

5.1 General overview

The method has the following characteristics:

- the entry is a PTG, i.e. the communication rules, the code of each generic task, and the parameters
- it results in an SPMD program, composed of two parts:
  1. the first part is a scheduling algorithm which is executed by a supervisor. This algorithm, called PTGDS (Parameterized Task Graph Dynamic Scheduler) explores the DAG using the

communication rules of the PTG. It locally schedules each task. Once a task T is assigned to a processor p, it orders dynamically processor p to execute task T,

2. the second part is executed by the processors, it is called the processor part. The processor part is composed of the task code, and receives order from the supervisor. This part executes the tasks, and handles the communications.

- the values of the parameters are given at run-time,
- only a small part of the DAG is known at any given time. But the DAG can virtually be explored entirely. That allows an important improvement in term of the space complexity.

5.2 PTGDS

This algorithm is executed by a dedicated processor, the supervisor. PTGDS schedules the tasks: it determines on which processor and when each task has to be executed. After instantiation of the parameters, using the analysis of the code performed by *PlusPyr*, and the parameterized task graph, it is possible for each task T to know its duration, the set of its sons, the set of its fathers, and the volume of communications between its fathers or its sons.

Figure 4, gives the general scheme of PTGDS 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 4: The PTGDS algorithm

Before scheduling a task T, PTGDS checks that all the predecessors of T have already been assigned to a processor. If some predecessors of T need to be scheduled, it calls recursively the procedure for these tasks and then allocates T to the processor that minimizes its start time.

In the DAG, *T-OUTPUT* is the last task to be executed. So, the first call of this recursive procedure is *schedule(T-OUTPUT)*, which ensures that all the predecessors of *T-OUTPUT* (all the tasks needed for the computation of the final result) will be scheduled, and executed.

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 [7]), to store all the values of *allocated(T)*. When all the sons of task T have been scheduled, the test of the line 3 will never be performed again for T. Then, we can remove from the AVL, all the informations about T. Finally, lines 7 and 8 allow a major reduction of the memory used during execution of the algorithm.

When a task T is assigned to processor P, the other processors have to send to P the data needed to the computation of T.

5.3 The processors part

Each processor executes the same algorithm: it waits for orders and executes them. Orders can be of 3 types: *execute* a task, *send* data to another processor, *receive* data from another processor. The execution orders are queued in a FIFO manner. This ensures that each processor executes tasks in the order given by PTGDS. The sending orders are put in a pool. When a task T is finished the algorithm checks if, in that pool, there are data to be sent that has just been computed by T. If so, the algorithm sends those data to the corresponding processor.

5.4 The AVL tree

As long as there are some sons of a given task T that have to be scheduled, we store in an AVL tree informations about T. These informations are stored on a node of an AVL, and are: state of the task (scheduled/ unscheduled), number of sons to be scheduled, processor where the task is mapped, starting time of the task. The search, the insertion and the deletion of a node in a AVL of size s are done in $O(\log s)$.

6 Analysis of PTGDS

In this section we study the general performances of PTGDS. First we show that the schedule given by PTGDS is valid. Then we present a theoretical result concerning its computational complexity, and finally we show that, as long as we deal with coarse grain DAG, the parallel time computed is within a bound of two to the optimal schedule.

6.1 Validity of the schedule

Lemma 1 *The schedule computed by PTGDS is valid.*

Proof of lemma 1:

1. The precedence constraints are respected. A task is scheduled after all its predecessors are scheduled. Its starting time is computed such that it is greater than the end time plus the max of the communication times of its fathers scheduled on the other processors.

2. The resource constraints are respected. When a task is allocated to a processor, its starting time is computed such that it is greater than the end time of the previous task scheduled on the same processor.

■

6.2 Computational complexity

Theorem 1 *The computational complexity of the scheduling algorithm is $O((e+v)(\log v + p))$, where p is the number of processor e is the number of edges and v the number of vertex in the DAG.*

Proof of theorem 1: We use an AVL tree, to store information about tasks. Let us assume there are s nodes in the AVL tree.

The proof of this theorem is done by computing the number of time each line is executed from the beginning to the end of the algorithm execution. We multiply that total number of executions by the cost of one execution to obtain the total cost of each line. Then, we add the total cost of each line to derive the computational complexity of the algorithm.

Let us consider figure 4. The procedure *schedule* is called once per task T. Each task is represented by an node in the DAG, so there are v calls *schedule(task T)*.

Line 2 (and 6), we use the PTG to find all the fathers of T. Thus, finding the next father of a task is done in constant time. During the whole execution of the algorithm, all the fathers of all the tasks are explored, this means that all the edges of DAG are examined. Hence, the total cost of line 2 and 6 is $O(e)$.

Let us compute how many time the test of line 3 is performed. For a given task T this test is performed a number of time equal to the total number of fathers of T. So, for all the tasks in the DAG this test is done a number of time equal to the number of edges of the DAG. PTGDS uses the AVL tree to determine if *allocated(T')* is true. This is done in $O(\log s)$. Hence, the total cost of the line 3 test is $O(e \log s)$.

Let us consider the allocation of all the tasks. We do a greedy allocation of each task (we choose the processor that minimizes the starting time of a given task), so we need to examine each edge once for the allocation of all the tasks. Each time an edge is examined it costs $O(\log s)$ to find the informations, in the AVL, concerning the father, and it takes $O(p)$ to update all the processor. For each task it takes $O(p)$ to choose the processor that minimizes the start time. We compute the duration of a task, using the PTG, in constant time. So the total cost of the whole allocation process (line 5) is $O(e \log s + ep + vp)$.

Since we maintain in the AVL the number of sons that have to be scheduled for each task, the test line 7 is done in $O(\log s)$. As for the test line 3 the total cost of line 7 is $O(e \log s)$.

Line 8 is the deletion of informations on T' in the AVL. This deletion occurs at most once for all the v tasks T'. The cost of one deletion is $O(\log s)$, so the total cost of line 8 is $O(v \log s)$.

Finally, the cost of one line 11 assignment is $O(logs)$ (it's an insertion in the AVL). It is done once for all the v tasks T. So the total cost of line 11 is $O(v \log s)$.

That proves, finally, that the total computational cost of PTGDS is $O((e+v)(\log s + p))$. To prove the theorem, we upper-bound [1] the size of the AVL tree by v , the total number of tasks in the DAG. ∎

6.3 Bounds on PTGDS

We shall discuss here bounds on the total execution time of a DAG scheduled by PTGDS using an unbounded number of processors. Such a scheduling is often called clustering. Hence we shall use clPTGDS to recall that the number of processors is supposed unbounded (at most the number of tasks).We prove that for a coarse grain DAG, clPTGDS is within a factor of two to the optimal clustering.

[1] As proved in the next sections this bound is not very tight, since for the Gaussian elimination $s = \sqrt{v}$.

Here are some particular definitions needed for these sections.

- Let $\ll$ denote the following order relation: $t \ll t'$ means that t must be executed before t'. A clustering is said ordered if for any pair of tasks t and t' in the same cluster, then either $t \ll t'$ or $t' \ll t$. In the following, we shall only consider ordered clusterings in which the order is compatible with the tasks dependences, i.e. $t \ll t'$ if there is a dependence from t to t' or if t and t' are independent. Note that a linear clustering is ordered.
- We call $ST_P(t)$ the starting time of task t computed by clPTGDS and $ST_C(t)$, the starting time of task t for any clustering C. Let $cl_C(t)$ be the cluster of task t for clustering C, $cl_P(t)$ be the cluster of task t for the clustering produced by clPTGDS. We say that ST_C is valid if it is compatible with the task dependence relation in G and the order relation in each cluster. It is optimal if there is no valid starting time function scheduling a task before ST_C.
- In an ordered clustering, let $before(t)$ be the task executed just before t in its cluster, let $son(t)$ be the set of the sons of t in G and $father(t)$ be the set of the fathers of t in G.
- Let G be the task DAG. For any ordered clustering C, let $G_C = (V_C, E_C)$ be the *associated* graph defined as follows: $V_C = V$ and $e = (t_1, t_2) \in E_C$ if and only if
  - either $t_1 \in father(t_2)$ and $cl_C(t_1) \neq cl_C(t_2)$
  - or $t_1 = before(t_2)$.

In the following, we shall only consider ordered clusterings.

Lemma 2 *clPTGDS produces an ordered clustering.*

Proof of lemma 2: According to lemma 1 clPTGDS derives a valid clustering. Each task is allocated to a processor and is given a starting time. The order of this clustering is then natural: if t and t' are in the same cluster then $t \ll t' \Leftrightarrow ST_P(t) < ST_P(t')$ ■

Lemma 3 *For any ordered clustering C such that for any tasks t_1 and t_2 where $t_1 \in father(t_2)$ and, $cl_C(t_1) = cl_C(t_2)$, then $t_1 \ll t_2$, the starting time function computed using the ASAP-rule is valid and is optimal.*

ASAP-rule:

$$ST(t) = max(max_{t'}(ST(t') + \tau_{t'} + c_{t',t}), ST(t'') + \tau_{t''}, 0)$$

where $t' \in father(t)$ and $cl_C(t) \neq cl_C(t')$, and $t'' = before(t)$.

Proof of lemma 3:

1. The starting time function using the ASAP-rule is valid. It respects the precedence constraints because:

(a) the starting time is greater than the completion time plus the communication time of all its father which are not on the same cluster,

(b) the starting time is greater than the completion time of all the fathers put on the same cluster (it is greater than the end time of the task executed just before and all the fathers are executed before it).

It respects the resource constraints because no task can start before the end of the task scheduled on the same cluster and executed just before it.

2. The starting time function using the ASAP-rule on ordered cluster is optimal. We show by contradiction that it is not possible for any tasks to have a smaller starting time than the one given by the ASAP-rule. Let t be a task such that we have found a valid starting time $ST(t) < ST_{ASAP}(t)$. Let us consider G_C the graph associated with C. We shall show by recurrence on the depth of C_G that there must exist a task t' in C_G such that:

   (a) t' has no father in C_G (it is a root node), thus t' is a root in G.

   (b) $ST(t') < ST_{ASAP}(t')$

   This is contradictory with the fact that, since t' has no predecessor, the ASAP-rule had given it 0 as starting time, and no task can start before 0. Concluding the lemma.

   Let us do the recurrence: Let n be the depth of node t in G_C (i.e. the longest path from t to a root node). We shall prove that for all $n \geq 1$, there always exists a node t' such that t' is a root node, and $ST(t') < ST_{ASAP}(t')$.

   (a) $n = 1$. For all $t' \in father(t)$ and $cl_C(t) \neq cl_C(t')$, t cannot start before all its fathers have completed and sent their data to it. Hence $ST(t) \geq max_{t'}(ST(t') + \tau_{t'} + c_{t',t})$.Let $t'' = before(t)$, t cannot start before t'' has ended. Thus, $ST(t) \geq ST(t'') + \tau_{t''}$. Since the clusters are ordered, we cannot change the order of the task. Then the ASAP-rule, and the fact that $ST(t) < ST_{ASAP}(t)$, imply that at least one task t', father of t in G_C, is such that $ST(t') < ST_{ASAP}(t')$. Since n=1, t' is a root node.

   (b) Let us assume that the property is true for n-1. Hence t has depth n. Using the same reasoning as before, we can find a task t' with depth at most $n-1$ such that $ST(t') < ST_{ASAP}(t')$. This concludes the recurrence.

∎

In the following theorem, we derive a bound on the quality of clPTGDS with respect to the optimal clustering.

Theorem 2 *For any DAG G, let $PT_P(G)$ be the parallel time of clPTGDS, $PT_{opt}(G)$ the optimal parallel time, and $g(G)$ the granularity of G. Then, on an unbounded number of processors:*

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

To prove this theorem we need the following lemma:

```
Execute clPTGDS on G;
forall cluster C
    forall task t of C taken by decreasing starting time
        if it exists t' in C such that t and t' are independent then
            put t in a new empty cluster;
            recompute the starting time of all the tasks in G according to ASAP-rule;
```

Figure 5: *Linearization algorithm*

Lemma 4 *For any linear clustering, on an unbounded number of processors, we have:*

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

Proof of lemma 4: This has been done by Gerasoulis and Yang in [6]. ∎

According to lemma 4, if we prove that there always exists a linear clustering which has worse parallel time than the one produced by clPTGDS, then the proof of theorem 2 is immediate. We shall prove that the algorithm in figure 5 builds such a linear clustering. We are going to modify the clustering given by clPTGDS but not the order of the tasks in a given cluster. If clPTGDS puts t and t' in the same cluster, then for any derived clustering C where t and t' are still in the same cluster then $ST_P(t) > ST_P(t') \Rightarrow ST_C(t) > ST_C(t')$.

Theorem 3 *The clustering C produced by the execution of the linearization algorithm is linear and has the following property: for all task t, $ST_C(t) \geq ST_P(t)$.*

In order to prove theorem 3, we wish to say that, if one removes a task from its cluster and put it in a new cluster, and if all its fathers are delayed then this task is delayed too. This is what is expressed in the following lemma.

Lemma 5 *For all ordered clustering C where t is the only task of its cluster (i.e. $cl_C(t) = \{t\}$), and if for all task $t' \in father(t)$, $ST_C(t') \geq ST_P(t')$ then $ST_C(t) \geq ST_P(t)$.*

Proof of lemma 5: In the new clustering C, t is put in an empty cluster, so we have $ST_C(t) = max_{t' \in father(t)}(ST_C(t') + \tau_{t'} + c_{t',t})$. According to clPTGDS we know that a task is put in the cluster that minimizes its start time. So, $ST_P(t) \leq max_{t' \in father(t)}(ST_P(t') + \tau_{t'} + c_{t',t})$ otherwise t could have been put in an empty cluster. Since $ST_C(t') \geq ST_P(t')$ the result is derived easily. ∎

We also need a lemma to express that, if a task is delayed and the order of the clusters is conserved, then no task can start earlier.

Lemma 6 *Let C be an ordered clustering, with all the starting times computed according to the ASAP-rule. Let us consider a new clustering C', with task t delayed ($ST_{C'}(t) > ST_C(t)$). Then, the recomputed starting time of any task t' according to the ASAP-rule is such that $ST_{C'}(t') \geq ST_C(t')$.*

Proof of lemma 6: Let us delay task t, and recompute the starting time of all the tasks according to the ASAP-rule. Let $G_{C'}$ be the graph associated with clustering C'. Then, for all task t':

- if there is a path between t and t' in $G_{C'}$, then $ST_{C'}(t') \geq ST_C(t)$ because of the ASAP-rule,
- if there is no path between t and t' in $G_{C'}$ then $ST_{C'}(t') = ST_C(t)$ because of the ASAP-rule.

■

Proof of theorem 3: All the clusters are linear because each time two independent tasks are found in the same cluster, the one that has the greatest starting time is put in a new empty cluster. Thus, at the end of the process there is no independent task in the same cluster.

To prove the second part of the theorem we call C_i the i^{th} clustering, i.e. the new clustering obtained after lines 5 and 6 have been executed i times.

We prove that for all task t:

- Part 1: $ST_{C_1}(t) \geq ST_{LC}(t)$
- Part 2: if $ST_{C_k}(t) \geq ST_{LC}(t)$, then $ST_{C_{k+1}}(t) \geq ST_{LC}(t)$

Part 1: Let t' be the task put on a new empty cluster at step 1.

1. If $t = t'$. Since the list-clustering has allocated t to the cluster that minimizes its starting time, $ST_{C_1}(t) \geq ST_{LC}(t)$.
2. If $t \neq t'$. Then $ST_{C_1}(t') \geq ST_{LC}(t')$ otherwise the list-clustering algorithm would have put t' on an empty cluster.
   (a) If t and t' are independent then
      i. if t and t' were in the same cluster then, since (line 3) we are considering tasks by decreasing starting time in a given cluster, we have $ST_{LC}(t) \leq ST_{LC}(t')$ (otherwise t would have already been put in a new empty cluster) hence $ST_{C_1}(t) = ST_{LC}(t)$,
      ii. if t and t' were not in the same cluster, let us consider G_{LC} the graph associated with the clustering built by the list-clustering. If it exists a path in G_{LC} between t' and t, since $ST_{C_1}(t') \geq ST_{LC}(t')$, then, according to lemma 6, $ST_{C_1}(t) \geq ST_{LC}(t)$. If there is no path in G_{LC} between t' and t, then $ST_{C_1}(t) = ST_{LC}(t)$.

   (b) If t and t' are dependent then
      i. if it exists a path in G from t to t' then $ST_{C_1}(t) = ST_P(t)$
      ii. if it exists a path in G from t' to t then $ST_{C_1}(t) \geq ST_P(t)$

Part 2: We suppose that for every task t $ST_{C_k}(t) \geq ST_{LC}(t)$. At step $k+1$ we have found two tasks t_0 and t_1 such that $cl_{C_k}(t_1) = cl_{C_k}(t_0)$, $t_1 \ll t_0$, and t_0 and t_1 are independent in G. Furthermore, for every task t_2 such that $cl_{C_k}(t_2) = cl_{C_k}(t_0)$ and $t_0 \ll t_2$, t_0 and t_2 are dependent (otherwise t_2 would already have been put on a new empty cluster). The algorithm puts t_0 in a new cluster. It recomputes the starting times, such that each task starts as soon as possible, keeping the order in the clusters, and building the clustering C_{k+1}. We prove that for each task t, $ST_{C_{k+1}}(t) \geq ST_{LC}(t)$.

1. If $t = t_0$, then for all task $t' \in father(t)$, $ST_{C_{k+1}}(t') = ST_{C_k}(t') \geq ST_{LC}(t')$, According to lemma 5 we can say that $ST_{C_{k+1}}(t) \geq ST_{LC}(t)$.
2. If $t \neq t_0$, we already know that $ST_{C_{k+1}}(t_0) \geq ST_{LC}(t_0)$.
   (a) If t and t_0 are independent then,
      i. if t and t_0 were in the same cluster then, since (line 3) we are considering tasks by decreasing starting times in a given cluster, we have $ST_{C_k}(t) \leq ST_{C_k}(t_0)$, hence $ST_{C_{k+1}}(t) = ST_{C_k}(t)$.
      ii. if t and t_0 were not in the same cluster, let consider G_{LC} the graph associated with the clustering built by the list-clustering algorithm. If there is a path in G_{LC} between t_0 and t, since $ST_{C_{k+1}}(t_0) \geq ST_{LC}(t_0)$, then, according to lemma 6, $ST_{C_{k+1}}(t) \geq ST_{C_k}(t)$. If there is no path in G_{LC} between t_0 and t, then $ST_{C_{k+1}}(t) = ST_{C_k}(t)$.

   (b) If t and t_0 are dependent then
      i. if there is a path in G from t to t_0 then $ST_{C_{k+1}}(t) = ST_{C_k}(t)$
      ii. if there is a path in G from t_0 to t then $ST_{C_{k+1}}(t) \geq ST_k(t)$

∎

Proof of theorem 2: Theorem 3 proves that there always exists a linear clustering with worse parallel time, and Lemma 4 shows that any linear clustering is within the required bounds. ∎

Finally we have this immediate corollary:

Corollary 1 *For any coarse grain DAG ($g(G) = 1$):*

$$PT_{clPTGDS} \leq 2 \times PTopt$$

7 Experiments

Theoretical results presented in the previous sections show that PTGDS is fast, and behaves well on an unbounded number of processors. Here we wish to show that, a dynamic SPMD program executing the schedule founded by PTGDS has good characteristics with regards to the parallel time and the memory cost.

7.1 Experiments on real DAG

We have done experiments on real DAG (i.e. DAG built by PlusPyr, with Fortran program).

Figure 6 shows the value of the speedup versus the number of processors for the Gaussian elimination of order 1000. This result is a simulation, i.e. we just run the schedule algorithm. The communication and computation costs are computed with the following model: according to the target parallel machine, there are three parameters which are: ω, the time taken by one elementary instruction, α, the startup time of a communication, and β the transmission rate[2]. This figure shows that for 32 processors the speedup is around 26.

Figure 7 shows the value of the speedup versus the number of processors, for computing the Mandelbrot set. This has been done using an SPMD program implementing the supervisor and the processors part. The execution of that program has been performed on a cluster of SPARC 4 workstations linked by Ethernet. The DAG corresponding to this program is a "join DAG", where all the tasks are independent and send there results to the task *T-OUTPUT*. We see that for 4 processors we obtain a speedup slightly greater than 3.5.

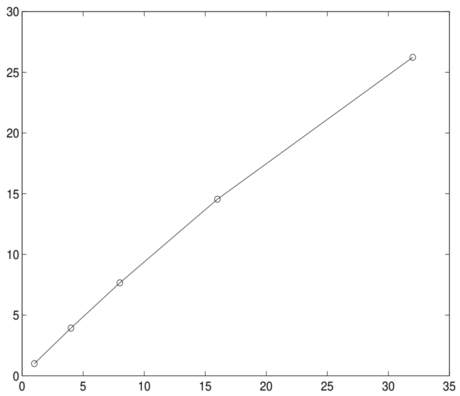

Figure 6: Speedup versus number of processors (Gauss $1000 * 1000$)

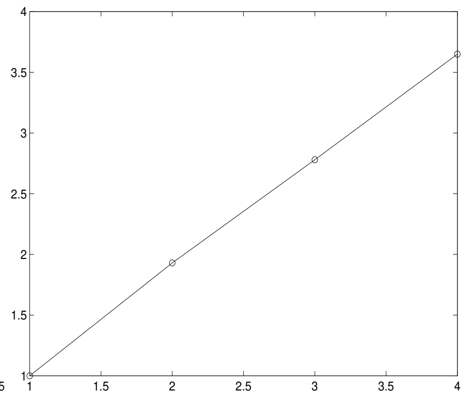

Figure 7: Speedup versus number of processors (Mandelbrot)

The memory cost of PTGDS is directly proportional to the maximum number of edges and of nodes in memory during its execution, and the maximum size of the AVL tree. Thus during the execution of the PTGDS on the Gaussian elimination,we have recorded these values and compared them to the total number of edges and nodes of the DAG.

Figure 8 shows that the total number of edges and nodes increases quadratically with respect to the matrix order. Figure 9 shows that the maximum number of edges and nodes in memory increases linearly with respect to the matrix order. We see also that the maximum size of the AVL tree increases linearly (in that case it is almost the same as the node number and the two curves are superposed) with respect

[2] Here we have taken $\alpha = 16$ μs/double, $\beta = 2.5$ ms and for one operation on a double $\omega = 0.882$ μs, which are the values for sparc 5 linked by Ethernet on PVM

to the matrix order. Thus we conclude that for the Gaussian elimination the space complexity of our algorithm is $O(\sqrt{v+e})$.

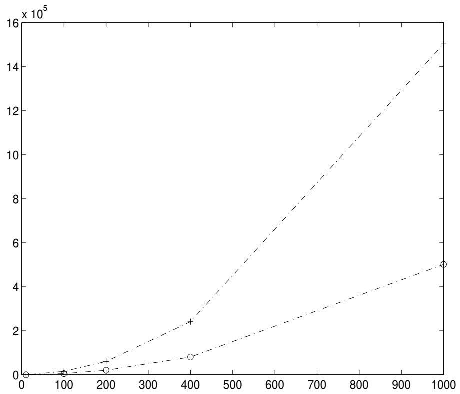

Figure 8: Total number of edges (+) and nodes (o) in the DAG, for Gaussian elimination versus matrix order

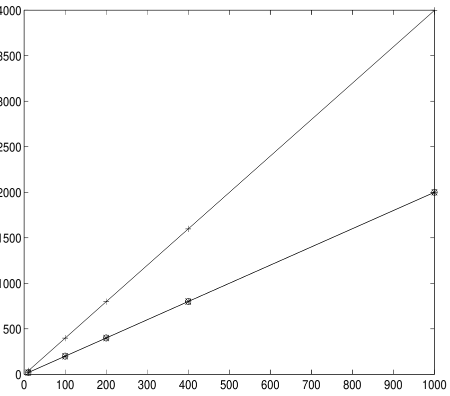

Figure 9: Maximum number of edges (+) nodes (o), in memory, and maximum size of the AVL (*)

7.2 Experiments on random graphs

We have also done experiments on random graphs in order to see how PTGDS behaves, in term of memory cost, on a large number of graphs.

Table 1, 2, 3 show the results of experiments on random graphs.

Table 1: Experiments on weakly linked random graphs

layers	tasks	max AVL	# nodes	max nodes	# edges	max edges
100	40	67(4)	3982(200)	217(19)	13888(709)	227(13)
100	80	69(4)	8140(381)	359(20)	28416(1310)	241(17)
200	40	129(10)	7982(541)	309(27)	27851(1897)	431(32)
200	80	127(5)	15437(735)	434(16)	53914(2555)	428(14)
400	40	254(17)	15780(874)	484(30)	55097(3083)	840(61)
400	80	253(14)	31755(1696)	632(50)	110961(5919)	839(52)
800	40	497(32)	31425(2005)	863(51)	109730(7008)	1636(105)
800	80	509(30)	64754(3538)	999(61)	226328(12384)	1671(103)

Each graph has a mean number of layers (column *layers*), a mean number of tasks per layers (column *tasks*) and a mean number of sons per tasks. In the weakly (resp. fairly, strongly) linked random graphs, each task has an average number of sons equal to 2 (resp. 4, 8). The sons are chosen among the tasks on the two next layers. Each row represents a type of graph. We have built 10 graphs of each type and computed the mean value of the maximum number of nodes in the AVL, the mean value of the number of nodes (column *# nodes*)), tasks (column *# edges*), and finally, the mean value of the maximum number of nodes and edges in memory (column *max nodes* and *max edges*). Into parenthesis is given the standard deviation of the value.

Table 2: Experiments on fairly linked random graphs

layers	tasks	max AVL	# nodes	max nodes	# edges	max edges
100	40	58(2)	3930(249)	183(6)	21276(1360)	192(10)
100	80	57(3)	7685(495)	306(8)	41841(2750)	192(10)
200	40	112(7)	7920(432)	238(10)	42952(2343)	358(20)
200	80	114(5)	15916(909)	359(10)	86876(4997)	366(16)
400	40	220(13)	15820(941)	347(14)	85944(5074)	684(38)
400	80	221(14)	31725(1888)	479(9)	173364(10363)	690(44)
800	40	439(23)	31805(1612)	574(25)	172881(8785)	1348(71)
800	80	443(20)	64307(3168)	702(17)	351626(17361)	1363(58)

Table 3: Experiments on strongly linked random graphs

layers	tasks	max AVL	# nodes	max nodes	# edges	max edges
100	40	55(2)	4062(209)	174(4)	36975(1960)	195(9)
100	80	52(2)	7679(411)	293(4)	71118(3783)	191(8)
200	40	103(6)	7771(568)	222(8)	70956(5282)	335(20)
200	80	104(6)	15886(916)	344(4)	147690(8514)	347(18)
400	40	200(10)	15331(867)	318(11)	140223(7913)	629(31)
400	80	211(13)	32693(2033)	446(13)	304498(18913)	666(42)
800	40	406(25)	31413(1989)	523(25)	287626(18126)	1247(76)
800	80	406(29)	63187(4428)	643(30)	589001(41372)	1251(88)

Whatever the type of the graph is, there is always less nodes or edges in memory than the respectively total number of nodes and edges in the DAG. The first remark we are able to do is that our measures seems relevant since the standard deviation is always lower than 10% of the value. The second interesting remarks is that the size of the AVL is always very small with regards to the number of nodes, and furthermore, is proportional to the number of layers in our graph (it is also the case for the Gaussian elimination as shown in figure 9). The size of the AVL (the number of task kept in memory for further computations) is an important value since it is a parameter of the computational complexity of PTGDS. Concerning the memory required by nodes and the edges during the execution, it is always a very small part of the total size of the graph. Then, we can conclude that our algorithm is low memory costly.

Nevertheless, it is not obvious that the memory required to schedule such graphs increases linearly when the number of tasks increases quadratically.

8 Concluding remarks

In this paper we have presented a complete line of automatic parallelization for coarse grain applications. This work is based on the PlusPyr tool, which built the parameterized task graph. This leads in the construction to a generic SPMD program which is dynamic, needs the parameters values at runtime, and is supported by PTGDS, a scheduling algorithm. We have proven here that PTGDS has a complexity of $O((e+v)(\log v + p))$, and is within a factor of two of the optimal for coarse grain task graph on an unbounded number of processors. We have done experiments in order to see how behaves our method on real problems concerning the parallel time and the memory cost.

Our current and future work are directed toward a generalization of the use of the parameterized task graph, in order to derive a symbolic static scheduling algorithm with performances comparable to PTGDS.

9 Acknowledgment

The authors would like to thank Michel Loi for providing us with the PlusPyr software, and for many valuable discussions.

References

[1] T. Brandes et al, *HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using HPF*, In J.J. Dongarra and B. Tourancheau, editors, Third Workshop on Environments and Tools for Parallel Scientific Computing, Faverges, August 1996. SIAM.

[2] S. Chintor and R. Enbody, *Performance degradation in large wormhole-routed interprocessor communication networks*, Proc. of ICPP'90, vol I, pp. 424-428.

[3] M. Cosnard and M. Loi, *Automatic task graph generation techniques*, Parallel Processing Letters, **5,4** (1995), 527-538.

[4] M. Cosnard and M. Loi, *A Simple Algorithm for the Generation of Efficient Loop Structures.* International Journal of Parallel Processing, to appear.

[5] P. Feautrier, *Dataflow analysis of array and scalar references*, Int. Journal of Parallel Programming **20** (1991), no. 1, 23–53.

[6] A. Gerasoulis, T. Yang. *On the Granularity and Clustering of Direct Acyclic Task Graphs.* IEEE Transaction on Parallel and Distributed Systems, Vol 4, No 6, June 1993.

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

[8] J. Jiao, *Software Support For Parallel Processing of Irregular and Dynamic Computations*, PhD Thesis, Rutgers University, New-Jersey, USA, 1996.

[9] 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, vol. 7, no. 5, May 1996.

[10] M. Loi, *Construction et exécution de graphes de tâches acyclique à gros grains*, PhD thesis, ENS Lyon, France, 1996.

[11] V. Sarkar *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, Cambridge, MA, M.I.T. Press, 1989.

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

[13] M. Ujaldon et al. *Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation*, Technical Report TR 95-5, University of Vienna, 1995.

[14] T. Yang and A. Gerasoulis, *Pyrros: Static task scheduling and code generation for message passing multiprocessors*, 6th ACM International Conference on Supercomputing (Washington, D.C.), July 1992, pp. 428–437.

[15] T. Yang and A. Gerasoulis, *DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors*, IEEE Transaction on Parallel and Distributed Systems, vol 5, No 9, september 1994, pp.951-967.