Compact DAG Representation and its Dynamic Scheduling *

Michel Cosnard
LORIA - INRIA Lorraine
615, rue du Jardin Botanique
54602 Villiers Les Nancy
France

Emmanuel Jeannot
LIP, ENS de Lyon
46, allée d'Italie
69364 Lyon cedex 07
France

Abstract Scheduling large task graphs is an important issue in parallel computing. In this paper we tackle the two following problems : (1) how to schedule a task graph, when it is too large to fit into memory? (2) How to build a generic program such that parameter values of a task graph can be given at run-time? Our answers feature the parameterized task graph (PTG), which is a symbolic representation of the task graph. We propose a dynamic scheduling algorithm which takes a PTG as an entry and allows to generate a generic program. We present a theoritical study which shows that our algorithm finds good schedules for coarse grain task graphs, has a very low memory cost and a low 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. We also provide experimental results that demonstrate the feasibility of our approach using several compute-intensive kernels found in numerical scientific applications.

* **This work was supported by the Region Rhône-Alpes, by the Eureka EuroTops project funded by the French Ministry of industry and by the NSF/CNRS grant Number 139812**
Preliminary versions of this paper was presented in NATO ARWHPC'96 and IEEE IPPS'98

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

1 Introduction

As the computational power of distributed memory parallel computer increases, very large problems are then to be solved. 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 [20, 21], or not [11, 18, 27]. When dealing with very large problems, static schedulers may fail 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.

Concerning scheduling of task graphs, another approach consists of dynamically scheduling the task graph [2, 3, 9, 13]. Dynamic scheduling is often used for irregular computation or load balancing issue. In some approaches [3], the task graph is built at run time (it allows to treat big size problem), but they do not address loop-nest programs.

In this paper we present and study the performance of new approach that allows to solve very large problems (of the order of a million tasks) and build a generic program. 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 [19], generates an intermediate program representation called the parameterized task graph (PTG) [7, 8]. The PTG is a compact program representation for DAG parallelism, and it requires a small amount of space to be stored because it is independent of program sizes. We propose an approach which consists of building a generic program that dynamically schedules the task graph. This method requires the parameter values to be given at run time. The parameterized task graph dynamic scheduler (PTGDS) 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 focused on the feasibility of such an approach. We show that this algorithm has a time complexity of $O((e+v)(\log v + p))$. We prove that The memory cost of PTGDS is bounded by $O(\Delta + \Gamma)$ where Δ is the sum of the in-degree of the tasks along one chain and where Γ is the number of task with out-degree greater than two. We provide a proof that for coarse grain DAG and an unbounded number of processors PTGDS is within a bound of two from the optimal schedule. We study the parallelization of several computation intensive kernels, discuss the limitation of our approach and derive some required conditions such that the scheduling overhead is negligible with respect to the parallel time.

Section 2 gives the definitions used in this paper. Section 3 deals with the parameterized task graph, and the PlusPyr tool. Section 4 discusses the issues of dynamic and static scheduling, and related works. Section 5 describes the general framework of our dynamic scheduling approach, including PTGDS algorithm. Section 6 gives theoretical results concerning the performances of PTGDS. Finally, Section 7 shows the experiments that have been carried out. In Section 8, conclusions and future works are given.

2 Definitions

Throughout 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 this paper 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.

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

- A cluster is called *non linear* if two independent tasks in G exist in this cluster, if not it is called *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 [23]. Gerasoulis and Yang have discussed the impact of granularity for clustering algorithms in [15].

The execution model for acyclic 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.

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

- 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 The Parameterized Task Graph

Cosnard and Loi have proposed the parameterized task graph [7] as a solution to derive automatically task graphs from sequential programs. It uses parameters which have to be instantiated in order to build the expanded task graph. It is beyond the scope of this paper to present a detailed description of the parameterized task graph. Only a brief description of this model will be given. First we will describe the input language that will be used by our automatic task graph builder [19]. Second we describe the parameterized task graph which is mainly composed of *generic task codes* and *communication rules.*

```
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
```

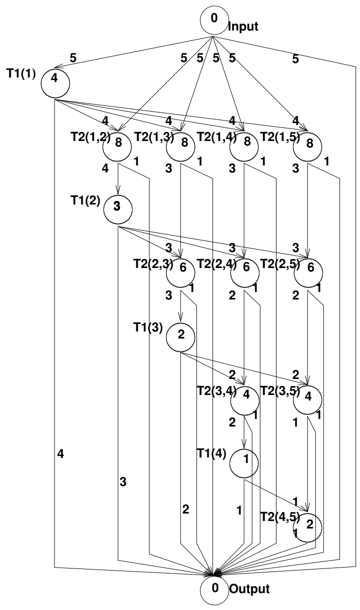

Figure 1: Gaussian Elimination Explicit Task Graph

3.1 The Input Program Model

Variable data types are restricted to simple types, and n-dimensional arrays of these types. We also restrict the control structure to 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 only composed of 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 abides by 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 The Parameterized Task Graph

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 will not give details of the definition of a task or the precise semantic of a communication rule (for more detail see [7]). It is enough to 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 other tasks are always added to the set of tasks defined by the user. One of them 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 is *T-OUTPUT*, which is the last task to be executed and reads all the variables, hence is used to gather the results.

3.3 Example of Communication Rules

Figure 2 shows an example of the reception rules derived from the program in Figure 1. Rule number 9

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

should be taken into consideration:

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

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 the pivot column $A(i,k)$ for all $k+1 \leq i \leq n$ from $T1(k)$.

All these rules symbolically describe 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 [19] has built a tool (*PlusPyr*). This tool 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 analyses are based on the work done by Feautrier on integer parametric programming (see [12, 24] for an introduction). The complete method used by PlusPyr will not be discussed here, (for more details see [7, 8]).

4 Static and Dynamic Scheduling

Once the parameterized task graph (PTG) is built, the direct acyclic task graph of the application is easily derived by giving values to the parameters [19]. That representation of the source code can now be used by static schedulers.

In the literature, a lot of static scheduling heuristics have been proposed. In [18], Kwok and Ahmad present various static scheduling algorithms. They propose a new static scheduling algorithm: DCP (Dynamic Critical Path) which, despite its efficiency, has a high computational complexity of $O(v^3)$. Some heuristics, like MH [10] schedules the task graph on an arbitrary processor network. Some algorithms, like MJD [20] or PY [21], use duplication of tasks in order to reduce the communication overhead. Duplication consists of executing the same task on different processors. In that way the communication between this task and its son executed on the same processors will be reduced. In this paper 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.

There also exists some tools that produces parallel codes. Pyrros is a tool built at Rutgers University by Gerasoulis and Yang [26]. It schedules and maps the tasks, and then produces a parallel code. PlusPyr has been integrated as a front end of Pyrros in a tool called *Pyrros+* [17]. Figure 3 shows how it is connected to PlusPyr. Pyrros uses DSC as scheduling algorithm. DSC (Dominant Sequence Clustering [27]), is a clustering algorithm which has a computational complexity of $O((e + v)(\log v)$ and a space complexity of $O(e + v)$.

In [1], a tool called CASCH, is presented. It allows to generate a schedule and a parallel code for a sequential program. CASCH uses standard static scheduling algorithms, found in the literature.

Some other works address the problem of scheduling the task graph at run-time. In *Cilk* [3], the task

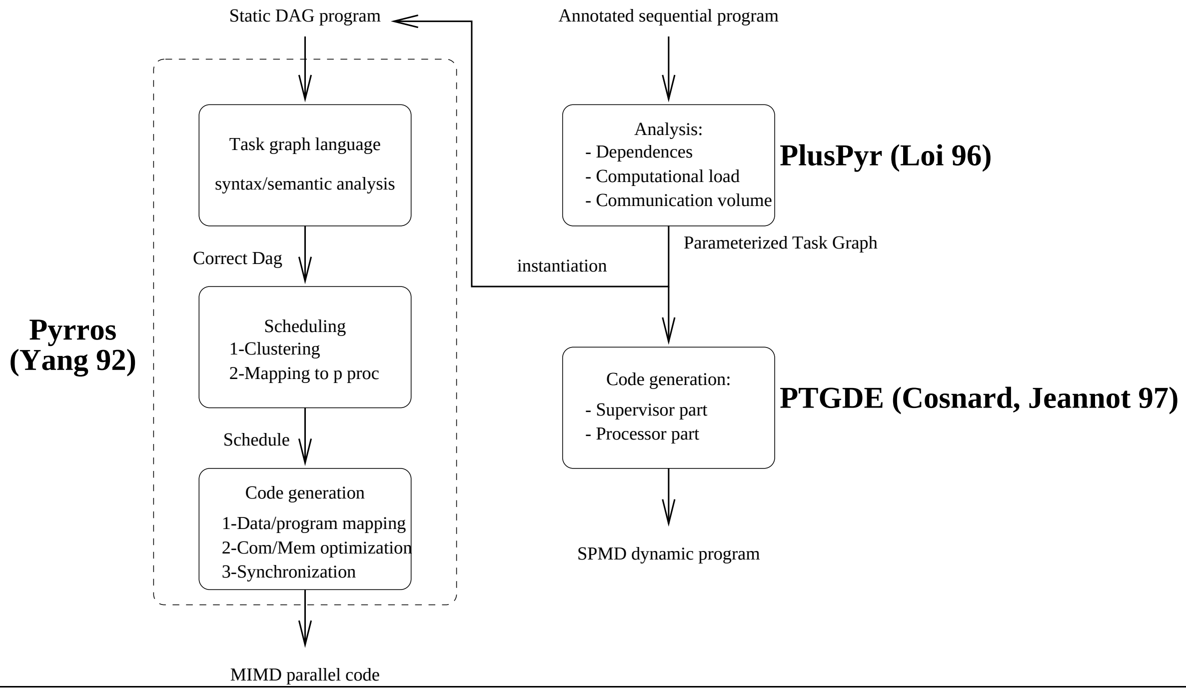

Figure 3: Pyrros and PlusPyr Schemes

graph is scheduled at run-time. The load balancing is done with the "work-stealing" policy. Cilk can handle big size problems but communication cost is not taken into consideration. The Cilk system gives good results with tree-like style computations (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 Athapascan-1 [9], parallelism is generated with a *fork* instruction which creates tasks at run-time.

5 Dynamic Scheduling of Parameterized Task Graphs

Dynamic execution policy is mainly used in the following two cases :

1. dealing with nondeterministic programs [3],

2. load balancing and process migration [22, 25].

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

1. static schedulers requires the program to be expressed as a static direct acyclic task graph, which is memory expensive. In the Gaussian elimination of a matrix of order 1000, the associated task graph has about 500000 nodes and 1000000 edges. As an entry for the Pyrros tool (each nodes is represented by its number its duration, the list of its son and the communication cost between with its sons) it requires 18 M-bytes of memory. As a matter of fact, a PTG is problem-size independent and it allows to build at run-time the small part of the DAG needed to perform task mapping optimization,

2. it is necessary to know parameters values at compile-time to use static schedulers. But those values may not be known at compile-time. In our approach, we give the parameters value, at run-time, and then schedule the tasks during the execution.

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, made 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 dynamically orders 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 using the communication rules. This allows an important improvement in terms of space

complexity.

5.2 The PTGDS Algorithm

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. 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* which has been developed at Université de Rennes [14]. It 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* from Université de Strasbourg [6], to build such a polynomial.

Figure 4, gives the general scheme of PTGDS algorithm.

Before scheduling a task T, PTGDS checks that all the predecessors of T have already been assigned to a processor. If predecessors of T need to be scheduled, it recursively calls the procedure for these tasks and then allocates T to the processor that minimizes its starting 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 [16]), to store all the values of *allocated(T)*. We

```
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

recall here that an AVL tree is a fast data structure that allows to insert, delete, and search elements in $O(\log n)$ where n is the number of elements in the AVL. When all the sons of task T have been scheduled, the test line 3 will never be performed again for T. Then, we can remove all the informations about T from the AVL. 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 AVL Tree

As long as there are some sons of a given task T that have to be scheduled, we store informations about T in an AVL tree. These informations are stored on a node of an AVL, and are the following: 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 an AVL of size s are done in $O(\log s)$.

```
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 executes 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
        computed data (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 5: PTGDE Code. Left: the Supervisor; Right: an Executor

5.4 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 command either the execution of the task or the sending of data to another. 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. Incoming 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 5 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. The execution messages that cannot be performed because the data are not yet present in the local processor memory are stored in *exec_list*. This list is updated when new data arrive or new data are computed. The send messages that cannot be performed because the data have not already been computed by the processor are stored in *send_list*. This list is updated when new data are computed.

6 Theoretical Results

In this section we study the general performances of PTGDS. First we show that the schedule given by PTGDS is valid. Second we present a theoretical result concerning its computational complexity and the memory cost. Then, we prove that, as long as we deal with coarse grain DAG, the parallel time computed is within a bound of two from the optimal schedule. Finally we show that it is possible to derive a simple upper bound on the average number of operations per tasks in order to obtain an efficient execution on a parallel computer.

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 have been scheduled. Its starting time is computed in such a way 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 assigned to a processor, its starting time is computed in such a way 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 processors e is the number of edges and v the number of vertices in the DAG.*

Proof of theorem 1: We use an AVL tree to store information about tasks. It is assumed that there are s nodes in the AVL tree.

To prove the theorem we compute the number of time each line is executed from the beginning to the end of the algorithm execution. We multiply the 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.

Consider Figure 4. The procedure *schedule* is called once per task T. Each task is represented by a node in the DAG, so there are v calls *schedule(task T)*.

In 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 $O(e)$.

We compute how many time the test of line 3 is performed. For a given task T this test is performed as many times as the total number of fathers of T. So, for all the tasks in the DAG this test is done as many times as 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 line 3 test is $O(e \log s)$.

Let us consider the assignments of all the tasks. this is done by 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 the p processors. 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 the number of sons that have to be scheduled for each task in the AVL, 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 $O(logs)$ (an insertion in the AVL of size s). It is done once for all the v tasks T. So the total cost of line 11 is $O(v \log s)$.

This proves that the 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 Memory Cost

In the PTGDS algorithm nothing is said about the order in which the tasks have to be scheduled. However, it is easy to see that the memory cost depends on this order. We discuss, here, the *worst case* that is to say the case where the order of the task scheduling implies the most memory utilization.

Theorem 2 *The memory cost of PTGDS is bounded by $O(\Delta + \Gamma)$ where Δ is the sum of the in-degree of the tasks along one chain and where Γ is the number of task with out-degree greater than two.*

Proof of theorem 2: The reason a task T is kept in memory by PTGDS is that one of its son is not allready scheduled. This leads to two cases:

case 1: The out-degree of T is greater than two. Let γ_T^+ the in degree of node T ($\Gamma = |\{T : \gamma_T^+ \geq 2\}|$). Let T' a task to be schedule and T one of its father. Before scheduling T' PTGDS schedule T. If T' is the only son of T ($\gamma_T^+ = 1$ then T is removed from memory. Otherwise ($\gamma_T^+ \geq 2$) T' has to be kept in memory until all its sons are scheduled. The worst case is attained when all the task of out-degree greater than 2 are remove from memory at the end of the process when the last task to be scheduled is the son of all of them.

case 2: The in-degree of one of its son is greater than 2. Let γ_T^- the in-degree of a task T. When PTGDS schedules T it is needed that its γ_T^- father are in memory. This property holds for all the task allong a chain C. Let $\Delta = \sum_{T \in C} \gamma_T^-$. In the worst case this chain is scheduled at the end of the process all the Δ fathers are allready in memory.

In the worst case both above cases occur. Hence, the memory cost of PTGDS is bounded by $0(\Delta + \Gamma)$. ∎

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

Discussion on the memory cost The memory cost of PTGDS proves that, in the worst case, all the graph has to be loaded in memory. However it appears that in most of the graphs we have tested only few tasks have an out-degree greater than two and it is very uncommon to find a chain of task with high in-degree. Nevertheless this bound can be considered as a characterization of the type of graphs where PTGDS has a good behaviour.

6.4 Bounds on PTGDS

We shall compute 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 to be unbounded (at most the number of tasks). We prove that for a coarse grain DAG, clPTGDS is within a factor of two from the optimal clustering.

Here are some particular definitions needed for these sections.

- Let $\ll$ denote the following order relation: $t \ll t'$ if task t must be executed before task t'. A clustering is said ordered, if for any pair of tasks t and t' in the same cluster, either $t \ll t'$ or $t' \ll t$. In the following section, 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 section, only ordered clusterings will be considered.

Lemma 2 *clPTGDS produces an ordered clustering.*

Proof of lemma 2: According to lemma 1 clPTGDS builds 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 fathers 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 ending 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 task to have a smaller starting time than the one given by the ASAP-rule. Let t be a task so that we have found a valid starting time $ST(t) < ST_{ASAP}(t)$. Let us consider G_C the graph associated with C. It will be shown by recurrence on the depth of G_C that there must exist a task t' in G_C so that:

   (a) t' has no father in G_C (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 gave it 0 as starting time, and no task can start before 0. This concludes the lemma.

   We should now 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 $n \geq 1$, a node t' always exists such as 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) If we assume that the property is true for n-1. Hence t has depth n. Using the same chain of arguments as before, we can find a task t' which depth is at most $n-1$ such that $ST(t') < ST_{ASAP}(t')$. This concludes the recurrence.

∎

```
Execute clPTGDS on G;
forall cluster C
    forall task t of C taken by decreasing starting time
        if t' exists 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 6: *Linearization Algorithm*

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

Theorem 3 *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:

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 [15]. ∎

According to lemma 4, if we prove that a linear clustering which has worse parallel time than the one produced by clPTGDS always exists , then theorem 3 is immediately proved. We will prove that the algorithm in Figure 6 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 4 *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 4, we wish to say that, if a task is removed from its cluster and put 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 the fact that, if a task is delayed and the order of the clusters is kept, 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. If we 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 4: 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 we are considering tasks by decreasing starting time in a given cluster(line 3), 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, consider G_{LC} the graph associated with the clustering built by the list-clustering. If a path in G_{LC} between t' and t exists, 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 a path in G from t to t' exists, then $ST_{C_1}(t) = ST_{LC}(t)$
      ii. if a path in G from t' to t exists, then $ST_{C_1}(t) \geq ST_{LC}(t)$

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. Then, the starting times are recomputed 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 we are considering tasks by decreasing starting times in a given cluster(line 3), 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, 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 $ST_{C_{k+1}}(t) \geq ST_{C_k}(t)$

■

Proof of theorem 3: Theorem 4 proves that a linear clustering always exists 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 ($g(G) = 1$):*

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

6.5 Minimizing the Impact of the Scheduling Overhead

In this section we derive a required condition such that the scheduling overhead is negligible with respect to the parallel time.

Definition 1 *Let us call A_{op}, the average number of operations per task 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 7 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p) + PT_P(G,p)$

Proof of Lemma 7: Due to scheduling overhead the time taken by the executor to finish is greater than the total parallel time. Moreover, since PTGDS is executed on the supervisor and the parallel code is executed simultaneously on p executors, the total dynamic execution time is lower than the sequentialization of the supervisor time plus the executor time. ∎

In many cases (particularly, in all the examples of section 7.1), the number of edges of the task graph is of the same order of the number of nodes. The main limitation of our approach is that dynamic scheduling is costly when dealing with fine grain task graphs. Theorem 5 gives a sufficient condition on the average number of task operations such that the scheduling overhead is negligible with comparison to the parallel time.

Theorem 5 *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 5 Let $T_{sup}(G,p)$ be the supervisor time.

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

The maximum amount of communication done by the supervisor is $O(e)$, since it may have 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:

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

We cannot have a superlinear speedup, thus:

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

By hypothesis, $pm = o(A_{op})$. This implies that: $pm = o(\frac{T_{seq}}{v\omega})$ and finally $kvm\omega = o(\frac{T_{seq}}{p})$. Replacing equation 1 and 2 leads to $T_{sup}(G,p) = o(PT_p(G,p))$. Since, according to lemma 7 $PT_P(G,p) \leq PTGDE(G,p) \leq T_{sup}(G,p) + PT_P(G,p)$, then: $PTGDE(G,p) = PT_P(G,p) + \epsilon$ ∎

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 Examples of Programs

We present here the set of program examples we used for our experiments. Extensive tests on the parallelization of these numerical kernels have been carried out.

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.

Mandelbrot: This program computes the Mandelbrot set. The accuracy of the computation is defined by the program parameters. The DAG corresponding to this program is a "join DAG", where all the tasks are independent and send their results to the task *T-OUTPUT*.

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.

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

Jordan Diagonalization: The 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 these examples experiments have been carried out to validate our approach. Speedup simulations, memory cost measurement and scheduling overhead comparison to the parallel time were done. All the tests were done using the following values (found on Sparc 5 workstation linked by Ethernet on PVM) : $\alpha = 16$ μs/double, $\beta = 2.5$ ms and for one operation on a double $\omega = 0.882$ μs.

7.2 Speedup Results

Figure 7 is a speedup simulation, it shows that, when matrix size is high the processor utilization is good. Figure 8 is the execution of the Mandelbrot program on a cluster of SPARC 4 workstations linked by Ethernet. We see that for 4 processors we obtain a speedup slightly greater than 3.5.

7.3 Memory Cost

7.3.1 Experiments on Real Examples

In Table 1 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 *# Nodes*, is the number of nodes in the DAG. PTGDS optimizes the number of tasks in memory. The list of tasks needed to allocate other tasks is

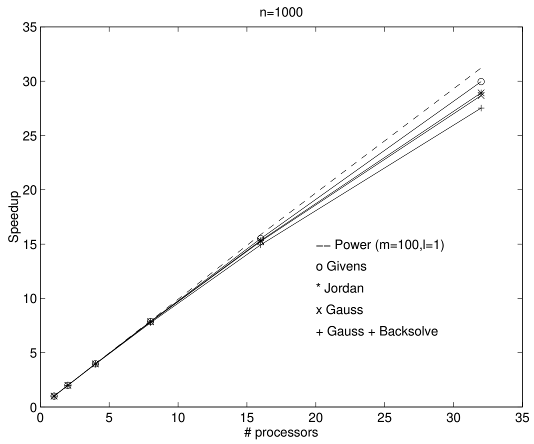

Figure 7: Speedup Simulation vs. Number of Processors, for Matrix Size=1000

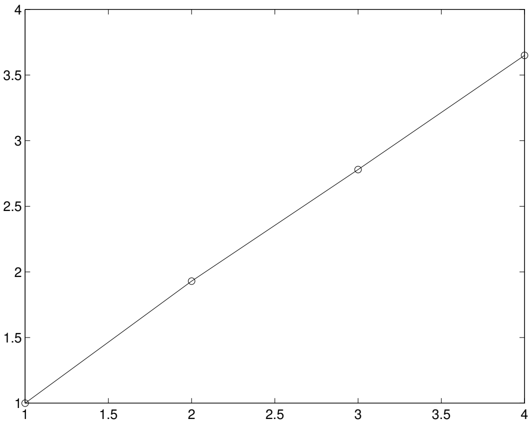

Figure 8: Speedup vs. Number of Processors (Mandelbrot)

handled by an AVL tree. Column *Max # nodes* 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 communication rules 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 *# Tasks* gives the number of tasks generated by the recursive DAG exploration. Column *Max # tasks* 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.

Table 1 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 the required memory increases linearly, while (except for *Power*), the size of the DAG increases quadratically.

Program	# Nodes	Max # nodes	# tasks	Max # tasks	# 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

7.3.2 Experiments on Random Graphs

We have carried out experiments on random graphs in order to see how PTGDS behaves, in term of memory cost, on a large number of graphs. Tables 2, 3 and 4 show the results of experiments on random graphs. 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*). The standard deviation of

Table 2: 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)

the value is given into parenthesis.

Table 3: 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 4: 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 may be, there are 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 seem relevant since the standard deviation is always lower than 10% of the value. The second interesting remark 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. 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 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 has a low memory cost. Nevertheless, it is not obvious that the memory required to schedule such graphs increases linearly when the number of tasks increases quadratically.

7.4 PTGDS Execution Time

Here we study when PTGDS execution time is small in comparison to the parallel time. We have run our algorithm for different cases.

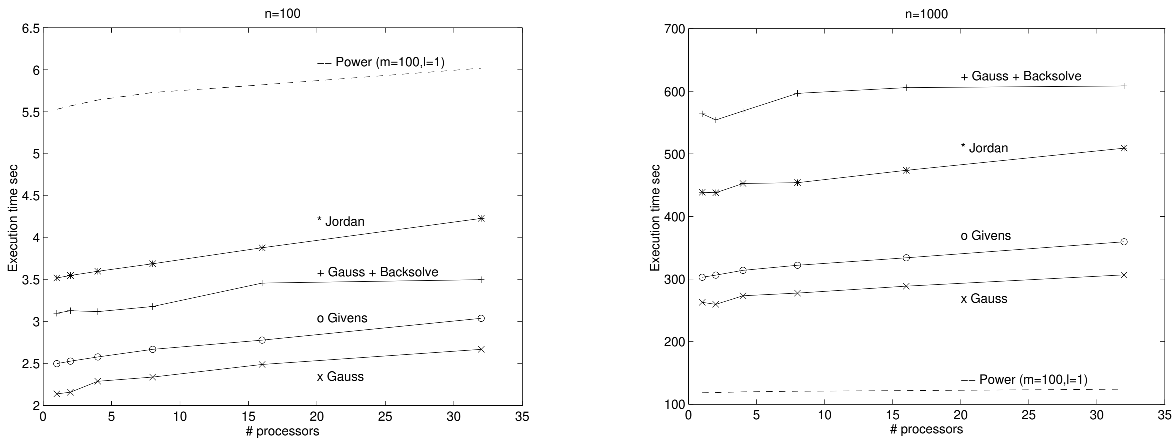

Figure 9: Execution Time of PTGDS vs. Number of Processors for Several Matrix Sizes

In Figure 9, we study the evolution of PTGDS execution time for different matrix sizes (n=100 and n=1000), with respect to the number of processors. It appears first, that the time taken to schedule a task graph slightly increases with the number of processors.

For large size problem, scheduling the computation of the power of a matrix is very competitive, since the sequential time is 300 times larger for *Power* than for *Gauss*[2]. This is due to the fact that the number of operations for each task in the program *Power* is very large.

7.5 Comparison Between Scheduling Overhead and Execution Time

In this section we show that, according to theorem 5, the amount of time taken by PTGDS to schedule the parameterized task graph is negligible when the average number of task operations of is high. In Figure 10, the simulated execution time of *Power* and *Givens* on a 32-processors machine (such as defined in this section), has been compared to 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 $PT(G) \approx 10.PTGDS(G)$.

[2] *Gauss* for n=1000 requires 1 billion flops, *Power* for m=100, n=1000, l=1 requires 298 billions flops

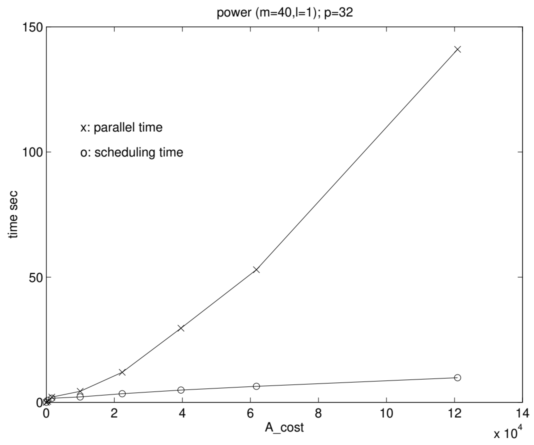

Figure 10: Execution Time of PTGDS and Execution Time Simulation vs. Average Number of Task Operations

For the Givens algorithm we see that when $p^2 \approx 20.A_{op}$ then $PT(G) \approx 5.PTGDS(G)$.

8 Concluding Remarks

In this paper we have presented a scheme for a complete line of automatic parallelization for some kind of coarse grain applications. This work is based on the PlusPyr tool, which builds the parameterized task graph. This leads to the construction of a dynamic generic SPMD program that needs the parameters values at runtime, and is supported by PTGDS, a scheduling algorithm. We have proved here that PTGDS has a competitive complexity of $O((e+v)(\log v + p))$, is within a factor of two from the optimal for coarse grain task graphs on an unbounded number of processors. We have found a bound of the memory cost. We have presented a link between the number of processors of the parallel machine, the number of tasks of the graph and the average number of operations a task should perform, such that the schedule overhead is 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.

Our current and future works are directed towards 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

This work is part of the European Community Eurêka EuroTOPS project. We would like to thank Michel Loi for providing us with the PlusPyr software andTao Yang of UCSB for very helpful discussions about this paper

References

[1] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu. Automatic Parallelization and Scheduling on Multiprocessors using CASCH. In *ICPP'97*, August 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] Ph. 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 M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, 1995.

[8] 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.

[9] M. Doreille, F. Galilée, and J.-L. Roch. Athapascan 1b : Présentation. Technical report, LMC IMAG, Grenoble, France, projet Apache, 1996. http://navajo.imag.fr/ath1/.

[10] H. El-Rewini and T.G. Lewis. Scheduling Parallel Programs onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, June 1990.

[11] H. El-Rewini, T.G. Lewis, and H.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. Le 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] J. Jiao. *Software Support For Parallel Processing of Irregular and Dynamic Computations.* PhD thesis, Rutgers University, New-Jersey, USA, 1996.

[18] 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.

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

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

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

[22] 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, April 1997.

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

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

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

[26] 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.

[27] 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.