AUTOMATIC COARSE-GRAINED PARALLELIZATION TECHNIQUES

M. COSNARD AND E. JEANNOT
Laboratoire de l'Informatique du Parallélisme, CNRS,
Ecole Normale Supérieure de Lyon, 69364 Lyon FRANCE

Abstract. We present a complete automatic coarse-grained parallelization tool allowing the execution of sequential programs on distributed memory parallel computers. It is based on the PlusPyr parameterized task graph builder and a new dynamic scheduling algorithm whose computational complexity and experimental performances are analyzed.

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

1. Introduction

Today, it is possible to automatically generate efficient scheduled code for message passing machines if dependence information is available [26]. Currently available tools require that the source program is expressed as static directed acyclic task graph. In such systems, it is the user's responsibility to construct the task graph by analyzing his program. He has to define tasks with sufficient granularity, compute the associated computational load, find dependences between tasks and determine the associated communication volume. After that, if experiments show that the initial task partitioning is not adequate, the user will have to iterate the whole process. This may be tedious and error prone for real life applications. [1]

In this paper, we present a complete automatic coarse-grained parallelization tool for most of the compute-intensive kernels found in numerical scientific applications. Most of these modules have a static control flow

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

with only restricted forms of DO loops [11] and IF-THEN-ELSE statements. This enables us to use compile-time techniques in order to collect the informations needed by the task graph construction. The techniques involved are, among others, exact data-flow analysis [7, 8, 11, 19, 20] and iteration counting.

In our system the user annotates his application source code with simple task definition directives. Then, the system automatically derives all the informations needed by scheduled code generators: source code and computational load associated with each task types, dependences and the associated communication volume between tasks instances. All these informations are computed in a symbolic form: the problem size parameters are variables and informations specific to some task instances are given as decision trees. Hence, the program analysis has not to be done for each possible values of the problem parameters. If the scheduled code generator requires that the task graph should be explicitly constructed then, given the exact problem size parameter values, the system is able to construct it. We are currently developing a back-end able to schedule and generate code by only exploiting the symbolic knowledge provided by our front-end.

2. Previous results

A lot of work is being done for compiling parallel programs, mainly around the HPF language [13, 4]. We shall not go into this direction in this paper and HPF will not be taken as the target language for our automatic parallelization tools. Related papers in this domain are [24, 3]. Our idealistic aim is to take as input a program and to execute it on distributed memory parallel computer with no human interaction. In order to automatically parallelize a sequential program, a set of operations should be performed on the input program. In fact, the program itself should satisfy strong requirements. The bulk of the work has been done for automating the parallelization of nested loops, where loop bounds and array indices are affine expressions of the loop indices and parameters. For fine grain parallelization techniques, the first step is the dependence analysis, where the dependences between loop statements are analyzed. This results in the reduced dependence graph. Few work has been done on models that may be used as an intermediate representation for parallelization tools on distributed memory systems. One of the most used in the area of automatic parallelization is the Data Flow Graph (DFG) [12]. The second phase is the scheduling and mapping of these statements on an unbounded set of virtual processors. The partitioning techniques allow the assignment of the statements to a finite number of processors and hence to obtain SPMD type code generation. Rajopadhye [21] has designed the LACS language for describing affine communications. Again, this model is fine grained but it has many similarities with our model as it uses polyhedra and affine transformations for specifying the communication volumes.

Yang and Gerasoulis have designed a graph description language for their mapping and scheduling tool Pyrros [26]. In this system it is the user's responsibility to construct the task graph by analyzing his program. Lo et al. [17] have proposed Temporal Communication Graphs (TGC) as a model for mapping and scheduling. It integrates the DAG and the static process graph models. The LaRCS language is used to describe TGCs graphs. This model has been used in the OREGAMI programming environment.

Most of the mathematical machinery deals with integer programming. For an introduction see [23]. Feautrier has proposed an algorithm for solving parametric integer programs [10]. This method has been implemented into the PIP software. Libraries for manipulating polyhedra are also available [25].

A lot of work has been done in the dependence analysis area. For an introduction, see [1]. We are only interested by exact dependences. Feautrier was a pioneer in this field [11]. Recent results have shown that it is possible to build fast dependence analyzers[20].

The parameterized task graph constructed by our method results in efficient task graphs only if some optimizations are done in order to enable the parallelism hidden by output and anti dependences or by an inadequate loop nest structure. A solution to the first problem is array privatization [9]. The second problem may be solved by finding an appropriate schedule of the operations [12]. The last problem is that the tasks should have sufficient granularity. As far as we know, only partial solutions exist, mainly by using data partitioning or tiling techniques [2, 15].

Code and data structure generation techniques are also closely related to our system. Rajopadhye and Wilde [22] have studied the problem of allocating polytopes in memory in an efficient way with respect to memory space usage and access time. Chamski has studied the derivation of optimized data structures for single assignment programs [5].

3. Parameterized task graph

The parameterized task graph has been proposed by Cosnard and Loi, [7] as a solution for automatically deriving task graphs from sequential programs involving the use of parameters, 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 [18].

```
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
```

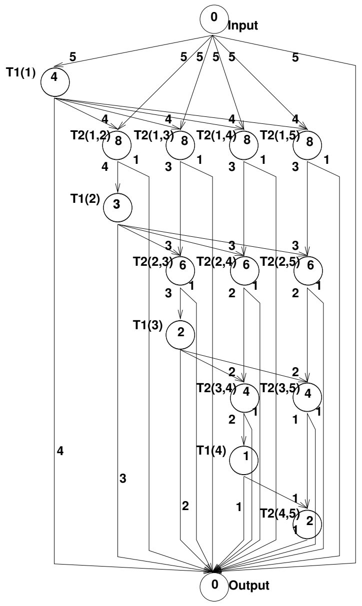

Figure 1. Gaussian Elimination explicit task graph

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

$\ll$ will denote the *non-strict textual order* in which statements appear in the program text. A task T is defined by its task name T, its iteration vector denoted by $\text{itv}(T)$, and its two delimiting statements $\text{first}(T)$ and $\text{last}(T)$. first(T) denotes the first statement of a task T (with respect to the order $\ll$) and last(T) denotes the last one. A statement S is included in task T iff $\text{first}(T) \ll S \ll \text{last}(T)$ and it will be denoted $S \in T$. Simple conditions ensure that task opening and closing and loop opening and closing are correctly nested. In a valid program, each statement S is included in a task $T = \text{task(S)}$. Thus, each operation $S(s)$ is included in a task instance $\text{task}\{S(s)\} = T(s_{1..n})$. $S_1 \prec S_2$ will denote the fact that operation S_1 executes before operation S_2, with respect to the sequential execution order of the program. $T_1 \prec T_2$ will denote the fact that task instance T_1 executes before task instance T_2, with respect to the partial order of execution induced by the program 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 which will be described later.

The specific execution of a statement will be called a *statement instance* or an *operation*. Generally, a statement enclosed in some loop will be executed several times, giving rise to many statement instances. In our input language, the only repetitive construct is the FOR-loop. Assume that a statement S has n surrounding loops $L_1, ..., L_n$ and that the loop counter of L_i is x_i. Hence, an instance of statement S is uniquely defined by $S(x)$, where x is the value of the column vector $(x_1, ..., x_n)^T$ for that operation. We call x the *iteration vector* of the statement S. itv(S) denotes the iteration vector of S. For each S the function domain(S) returns the set of all the possible values of the iteration vector x of S. This set will be called the *iteration domain* of S.

A n dimensional integer polyhedron P is the set $\{x \mid x \in N^n, Ax \geq b\}$, where A and b are respectively a matrix and a vector with integer coefficients. All iteration domains are bounded polyhedra. $\mid P \mid$ denotes the dimension of the polyhedron P. proj(P,m,n) denotes a projection of the polyhedron P such that $\text{proj(P,m,n)} = \{x_{m..n} \mid x \in P\}$.

We extend the notion of iteration vectors and iteration domain to tasks and use the term *task instance* to designate a specific task execution $T(t)$. Note that a task instance is nothing else than a set of operations. task(S) denotes the enclosing task of a statement S. vcalc{T(t)} denotes the arithmetic cost of task instance $T(t)$. We will assume that for any statement S, $\text{vcalc(S)} > 0$. $\text{vcomm}(T_1(t_1), T_2(t_2))$ denotes the number of data items sent by task instance $T_1(t_1)$ to task instance $T_2(t_2)$.

Optimizing the constructed parameterized task graph is beyond the scope of this paper. Hence, we make two hypotheses. First, we do not address computation replication. See Kruatrachue and El-Rewini [16] or Colin and Chretienne [6] for an attempt to lift this restriction. Second, the memory requirement of the parameterized task graph execution should be of the same order as the one of the sequential execution. In particular, we do not perform array privatization [9].

4. Construction of parameterized task graphs

In this section we propose techniques automating the construction of the tasks and the communication rule set, starting from an annotated sequential program.

4.1. FLOW, ANTI AND OUTPUT DEPENDENCES

There is a dependence between operations $R(r)$ and $S(s)$ only if both operations reference variable M, if at least one of the references is a write, if $R(r) \prec S(s)$ and if M is not written between the execution of $R(r)$ and $S(s)$. There are three kinds of dependences based on the type of the reference. It is a *flow dependence* if R writes M and S reads it, an *anti dependence* if R reads M and S writes it and it is an *output dependence* if both R and S write M. Each dependence is characterized by an integer, the dependence depth d, defined as follow: if $\mathrm{itv(R)} \cap \mathrm{itv(S)} = \emptyset$ then $d = 0$, else d is the maximum integer such that $r_{1..d} = s_{1..d}$. Moreover, if $\mid \mathrm{itv(R)} \cap \mathrm{itv(S)} \mid > d$ then $r_{d+1} > s_{d+1}$.

Such dependences are generally computed by a two step method [11]. First, one has to compute the so-called direct dependences: for a fixed S and a candidate R at depth d express that $R(r)$ and $S(s)$ both reference variable M and that $R(r) \prec S(s)$ at depth d as a parametric linear integer program where the components of itv(R) are parameters and the components of itv(S) are unknowns. Then s is the lexicographic maximum of the set of feasible points, that is the last operation satisfying the constraints. This linear program is solved by using parametric linear programming techniques, see [10]. Then it is necessary to combine the direct dependences together because some of them may hide others, at least partially.

In this paper, we will only assume that the result of dependence analysis may be represented as degenerate communication rules denoted as $\{R(r) \mid r \in P_R\} \leftarrow \{S(f_S(r))\} : \{M(f_D(r))\}$. Note that for such rules each task is a statement, and thus each task instance is an operation. Moreover, for a particular operation $R(r)$, for each RHS reference $M(f_D(r))$ there

$\{S_1(k,l) \mid 2 \leq k \leq n-1, k+1 \leq l \leq n\} \leftarrow \{S_2(k-1,k,k)\}$
$: \{A(k,k)\}$
$\{S_1(k,l) \mid k = 1, k+1 \leq l \leq n\} \leftarrow \{\textit{S-INPUT-A(k,k)}\}$:
$\{A(k,k)\}$
$\{S_1(k,l) \mid 2 \leq k \leq n-1, k+1 \leq l \leq n\} \leftarrow \{S_2(k-1,k,l)\}$
$: \{A(l,k)\}$
$\{S_1(k,l) \mid k = 1, k+1 \leq l \leq n\} \leftarrow \{\textit{S-INPUT-A(l,k)}\}$
$: \{A(l,k)\}$
$\{S_2(k,j,i) \mid 1 \leq k \leq n-1, k+1 \leq j < n+1, k+1 \leq i \leq n\}$
$\leftarrow \{S_1(k,i)\} : \{A(i,k)\}$
$\{S_2(k,j,i) \mid 2 \leq k \leq n-1, k+1 \leq j < n+1, k+1 \leq i \leq n\}$
$\leftarrow \{S_2(k-1,j,k)\} : \{A(k,j)\}$
$\{S_2(k,j,i) \mid k = 1, k+1 \leq j < n+1, k+1 \leq i \leq n\}$
$\leftarrow \{\textit{S-INPUT-A(k,j)}\} : \{A(k,j)\}$
$\{S_2(k,j,i) \mid 2 \leq k \leq n-1, k+1 \leq j < n+1, k+1 \leq i \leq n\}$
$\leftarrow \{S_2(k-1,j,i)\} : \{A(i,j)\}$
$\{S_2(k,j,i) \mid k = 1, k+1 \leq j < n+1, k+1 \leq i \leq n\}$
$\leftarrow \{\textit{S-INPUT-A(i,j)}\} : \{A(i,j)\}$
$\{\textit{S-OUTPUT(i,j)} \mid i = 1, 1 \leq j \leq n+1\} \leftarrow \{\textit{S-INPUT-A}\}$
$: \{A(i,j)\}$
$\{\textit{S-OUTPUT(i,j)} \mid 2 \leq i \leq n, 1 \leq j \leq i-1\}$
$\leftarrow \{S_1(j,i)\} : \{A(i,j)\}$
$\{\textit{S-OUTPUT(i,j)} \mid 2 \leq i \leq n, i \leq j \leq n+1\}$
$\leftarrow \{S_2(i-1,j,i)\} : \{A(i,j)\}$

Figure 2. Result of the dependence analysis for the Gaussian Elimination

is an unique operation $S(f_S(r))$ that writes the corresponding value. Such rules are called dependence rules, or simply dependences.

Figure 2 shows the result of the dependence analysis for the Gaussian Elimination.

4.2. FROM DEPENDENCES TO COMMUNICATION RULES

For transforming dependences into communication rules, we should gather dependences involving statements belonging to the same tasks. In other words, we should go from a fine grain analysis to coarse grain communication rules. The dependence analysis specifies for each communication rule i a positive integer depth(f_S^i), the depth of the dependence. If $\text{itv}(R) \cap \text{itv}(S) = \emptyset$ then $\text{depth}(f_S) = 0$ and f_S is an arbitrary affine function. Else, let $n = \text{depth}(f_S)$. Then $\forall r \in P_R$, $r_{1..n} = f_S(r)_{1..n}$. Moreover if $\mid \text{itv}(R) \cap \text{itv}(S) \mid > n$ then $r_{n+1} > f_S(r)_{n+1}$.

There are only two types of dependences: intra-task dependences are such that $\forall r \in P_R, \text{task}(R(r)) = \text{task}(S(f_S(r)))$, and intertask dependences are such that $\forall r \in P_R, \text{task}(R(r)) \neq \text{task}(S(f_S(r)))$. Intertask dependences involve data transfer operations between task instances, it is not the case for intra-task dependences.

– If $\text{task}(R) \neq \text{task}(S)$, then $\forall r \in P_R$, $\text{task}(R(r)) \neq \text{task}(S(f_S(r)))$ (intertask),

– else $\text{task}(R) = \text{task}(S) = T$. Let $n = \text{depth}(f_S)$ and $m = \mid \text{itv}\{T\} \mid$. Then, either

- $m = 0$ and then there is only one task instance for task T and thus $\text{task}\{R(r)\} = \text{task}\{S(f_S(r))\}$ (intra-task),
- or $n < m$ and then $\forall r \in P_R, r_{1..m} \neq f_S(r)_{1..m}$, and thus $\text{task}\{R(r)\} \neq \text{task}\{S(f_S(r))\}$ (intertask),
- or $n \geq m$ and then $\forall r \in P_R, r_{1..m} = f_S(r)_{1..m}$ and thus $\text{task}\{R(r)\} = \text{task}\{S(f_S(r))\}$ (intra-task).

Thus, rules corresponding to intra-task dependences may be safely ignored when building the communication rules. Transforming a dependence such as $\{R'(r) \mid r \in P'_R\} \leftarrow \{S'(f'_S(r))\} : \{M(f'_D(r))\}$ into a communication rule: $\{R(r) \mid r \in P_R\} \leftarrow \{S(f_S(s)) \mid s \in P_S(r)\} : \{M(f_D(d)) \mid d \in P_D(s)\}$ with $R = \text{task}\{R'\}$ and $S = \text{task}\{S'\}$ is done as follows.

Let $m = \mid \text{itv}\{R\} \mid$ and $n = \mid \text{itv}\{S\} \mid$. Then $P_R = \text{proj}(P'_R, 1, m)$ and $P_S(r) = \{x \mid x \in P'_R, x_{1..m} = r\}$. If $f'_S(r) = C' \times r + d'$ and $f_S(s) = C \times s + d$ then C is equal to the n first rows of C' and d is equal to the n first rows of d' . Last, $P_D(s) = \{s\}$ and $f_D = f'_D$.

Note that the constructed graph has no cycle. Let $T(t)$ be a task instance. The task definition ensures that if $S_1, S_2 \in T(t)$ and $S_1 \prec S_3 \prec S_2$ then $S_3 \in T(t)$. This property and the dependence definitions ensure that a path into the constructed graph from $T(t)$ to $T'(t')$ may exist only if $\text{first}\{T(t)\} \prec \text{first}\{T'(t')\}$. Hence, by contradiction, the constructed graph is acyclic.

4.3. DETERMINING THE COMPUTATIONAL LOAD AND THE COMMUNICATION VOLUME

The restricted form of our source language ensures that each simple statement S has a constant computational load denoted by $\text{vcalc}(S)$. PlusPyr is able to compute this load simply by summing up an estimated elemental arithmetic cost of each operator involved in the right hand side of S. Hence, each statement instance $S(s)$ for any valid iteration vector s has the same execution cost. Hence the computational load of task $T(t)$ is $\text{vcalc}\{T(t)\} = \sum_{S \in T} \text{vcalc}(S) * V(\{x \mid x_{1..|\text{itv}\{T\}|} = t, x \in \text{domain}\{S\}\})$.

For a given communication rule $\{R(r) \mid r \in P_R\} \leftarrow \{S(f_S(s)) \mid s \in P_S(r)\} : \{M(f_D(d)) \mid d \in P_D(s)\}$ and for a given value of $r \in P_R$ and $s \in P_S(r)$ the communication volume associated to the edge $(S(f_S(s)), R(r))$ is simply $V(P_D(s))$.

5. Pyrros

Once the parameterized task graph (PTG) is built, it is very easy, by giving values to the parameters, to derive the direct acyclic task graph of the application [18]. That representation of the source code can now be used, as an entry of *Pyrros* [26], to schedule and to map the tasks, and then to produce the parallel code.

5.1. ARCHITECTURE OF PYRROS

As shown in figure 3, the Pyrros software is composed of 3 parts:

1. The first part checks if the input graph is correct, generates a correct DAG and computes the real communication and computation costs. In order to compute such costs the user has to give the following target machine parameters : α, the startup time, β the transmission rate, for the communication cost, ω the time taken for one elementary operation, for the computation cost.
2. The second part schedules and maps each task of the DAG. Pyrros uses a two-step scheduling method: clustering on an unbounded number of processor, then merging the clusters to obtained p virtual processors. The next step of this part is mapping the virtual processors onto real processors. The mapping heuristic used by Pyrros takes into account the distance between real processors.
3. Once each processor is assigned to a set of tasks, Pyrros generates the parallel code for the target machine. Pyrros makes some optimizations to improve code performance. Those optimizations address memory utilization and communication.

5.2. DSC, THE SCHEDULING ALGORITHM

One of the most interesting part of Pyrros, is its static scheduling algorithm: Dominant Sequence Clustering (DSC [27]). The *Dominant Sequence* (DS), is the longest path in the scheduled graph. Initially each task is a cluster. In order to decrease the parallel time DSC tries to zero an edge in the DS. Each time an edge is zeroed, DSC merges the two clusters on the extremities of that edge. Then DSC recomputes the new DS until all edges of the DAG have been examined. The computational complexity of this algorithm is $O(e+v)\log v$ where v is the number of vertices of the DAG and e is the number of edges. Furthermore, DSC has been shown to behave very well, in terms of parallel time, for real life application, as for random graphs. DSC can be considered as one of the best scheduling algorithms. Since the whole DAG has to be in memory during the execution of DSC, the space complexity of that algorithm is $O(e+v)$.

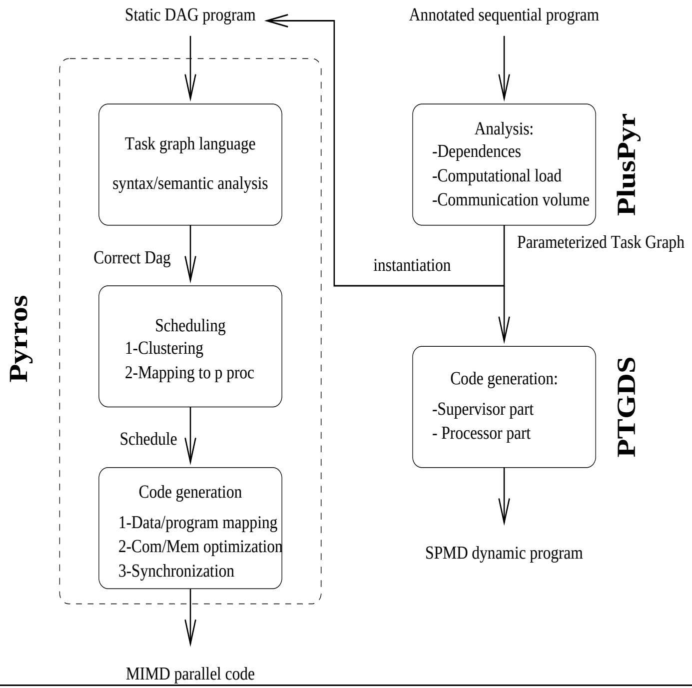

Figure 3. Pyrros and PlusPyr schemes

6. Dynamic scheduling of parameterized task graphs

We present a dynamic algorithm that schedules PTG. This work originates from two remarks:

1. DSC, in Pyrros, requires that the program 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 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 scheduling algorithm, called PTGDS, 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.

6.1. GENERAL OVERVIEW

PTGDS has the following characteristics:

- the entry of the algorithm is a PTG, i.e. the communication rules, the code of each generic task, and the parameters
- the PTG is executed by an SPMD program, composed of two parts:
    1. the first part is executed by a supervisor. The supervisor 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 task, 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.

6.2. THE SUPERVISOR ALGORITHM

This part of the algorithm is executed by a dedicated processor. The supervisor schedules the tasks: it determines on which processor and when each task has to be executed. For each task T, after instantiation of the parameters, the analysis of the code performed by *PlusPyr*, and the parameterized task graph allow to know the duration of T, all the sons of T, all the fathers of T, and the volume of communications between fathers and sons.

Figure 4, gives the general scheme of the supervisor 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, let *T-OUTPUT* be the last task to be executed. So, the call *schedule(T-OUTPUT)* 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 [14]), 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.

```
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 minimize its starting time;
6 for each task T' in father(T) do
7   if T is the last son of T' it has to be scheduled then
8     remove from memory all information on T'
9  endif
10 endfor
11 allocated(T)=true;
12 }
```

Figure 4. The supervisor algorithm

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.

6.3. THE PROCESSOR ALGORITHM

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. Thus PTGDS ensures that each processor executes tasks in the order given by the supervisor. 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.

6.4. 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. Informations stored on a node of an AVL 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 $O(\log s)$.

7. Experiments and performances

In this section we study the general performances of PTGDS. The goal of PTGDS is to minimize the memory required for scheduling the DAG, in order to be able to deal with large graphs. First we present a theoretical result concerning its computational complexity, second we present results concerning real parameterized task graphs, and then, since scheduling is an NP-complete problem, we study the memory required by PTGDS for random graph.

7.1. COMPUTATIONAL COMPLEXITY

We show here that PTGDS is, in term of computational complexity, comparable to DSC.

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

Proof: 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. 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 $O(e \log s)$.

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 $O(\log s)$. With the same argument than for the line 3 test, we see that 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 $O((e+v) \log s)$. To prove the theorem, we upper-bound [2] the size of the AVL tree by v , the total number of tasks in the DAG. ■

7.2. EXPERIMENTS ON REAL DAG

Figure 5 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 supervisor part of PTGDS. The communication and computation costs are computed with the same model as in the section 5.1. This figure shows that for 32 processors the speedup is around 26.

Figure 6 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 two parts of PTGDS. The execution of that program has been performed on a cluster of SPARC 5 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.

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 supervisor part we have recorded these values and compared them to the total number of edges and nodes of the DAG.

Figure 7 shows that the total number of edges and nodes increases quadratically with respect to the matrix order. Figure 8 shows that the maximum number of edges and nodes in memory increases linearly with

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

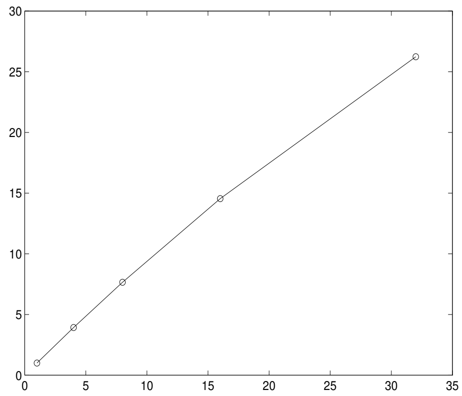

Figure 5. Speedup versus number of processors (Gauss $1000 * 1000$)

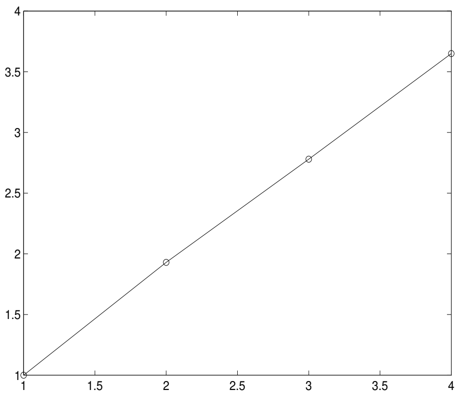

Figure 6. Speedup versus number of processors (Mandelbrot)

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

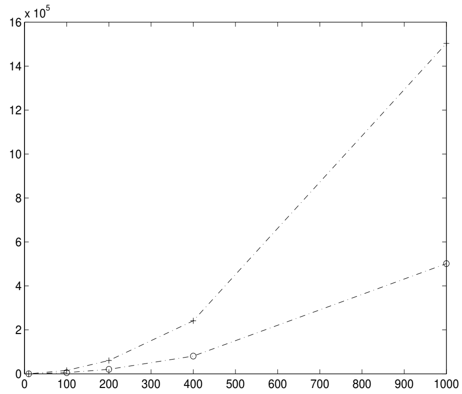

Figure 7. Total number of edges (+) and nodes (o) in the DAG, for Gaussian elimination versus matrix order

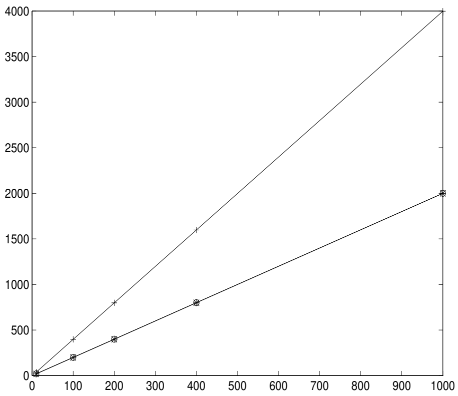

Figure 8. Maximum number of edges (+) nodes (o), in memory, and maximum size of the AVL (*)

7.3. EXPERIMENTS ON RANDOM GRAPHS

Table 1 shows the results of experiments we have done on random graphs. Each graphs has an average number of layers (column *layers*), an average number of tasks per layers (column *tasks*) and an average number of sons per tasks. 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 average value of the number of nodes (column *# nodes*)), tasks (column *# edges*), the average maximum number of nodes and edges in memory (column *max nodes* and *max edges*), and finally the average maximum number of nodes in the AVL.

TABLE 1. Experiments on weakly linked random graphs

layers	tasks	# nodes	max nodes	# edges	max edges	max nodes AVL
100	40	3819.9	255.3	12686.2	1309.6	43.2
100	80	6438.3	334.2	21725.4	2488.7	32.6
200	40	4884.4	117.4	17069.7	2462.7	56.3
200	80	11824.4	408.2	40540.0	5156.2	46.0
400	40	16977.4	468.2	56650.1	5823.9	76.2
400	80	38606.0	866.3	128621.8	12914.0	93.6

In the weakly linked random graphs, each task has an average number of son equal to 1.5 .

Whatever the type of the graph is, there is always less nodes or edges in memory that the respectively total number of nodes and edges, in the DAG. We see that the size of the AVL is always very small with regards to the number of nodes. Nevertheless, it is not obvious that the memory required to scheduled such graphs increases linearly when the number of task increases quadratically.

8. Concluding remarks

In this paper we have presented two tools allowing the automatic coarse grain parallelization of sequential programs and their execution on distributed memory parallel computers. Many constraints have to be imposed on the input programs. These tools are independent and hence can be used in relation with other tools. PlusPyr, a parameterized task graph builder, can serve as front-end program analyzer for scheduling algorithms. PT-GDS is dynamic scheduling algorithm taking as input a parameterized task graph.

Our current investigations are directed towards the generalization of the use of PlusPyr and also in deriving a symbolic static scheduling algorithm with performance of the same order as PTGDS.

Finally we would like to thank Michel Loi for providing us with the PlusPyr software and for many fruitful discussions.

References

1. U. Banerjee, *An Introduction to a Formal Theory of Dependence Analysis*, The Journal of Supercomputing 2,2, 1988, pp. 133–149
2. P. Boulet et al., *(Pen)-ultimate tiling ?*, Research Report LIP-CNRS-ENS LYON 93-36, 1993
3. 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.
4. Th. Brandes and F. Zimmermann, *ADAPTOR - A Transformation Tool for HPF Programs, In K.M. Decker and R.M. Rehmann, editors*, Programming Environments for Massively Parallel Distributed Systems, pages 91–96. Birkhäuser, April 1994.
5. Z. Chamski, *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*, Ph. D. Thesis Université Rennes I, Rennes France, 1993
6. J.Y. Colin and P. Chretienne, *Grain Size and Communication Channels on Distributed Memory Multiprocessors*, ParCo93, 1993, pp. 281–286
7. M. Cosnard and M. Loi, *Automatic task graph generation techniques*, Parallel Processing Letters, **5,4** (1995), 527-538.
8. M. Cosnard and M. Loi, *A Simple Algorithm for the Generation of Efficient Loop Structures.* International Journal of Parallel Processing, to appear.
9. R. Cytron, M. Hind and W. Hsieh, *Automatic generation of DAG Parallelism*, Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, 1989
10. P. Feautrier, *Parametric Integer Programming*, RAIRO Recherche Opérationelle 22, 1988, pp. 243–268
11. P. Feautrier, *Dataflow analysis of array and scalar references*, Int. Journal of Parallel Programming **20** (1991), no. 1, 23–53.
12. P. Feautrier, *Some Efficient Solutions to the Affine Scheduling Scheduling Problem, Part I, One Dimensional Time*, Int. Journal of Parallel Programming **21** (1992)
13. High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0.α.3*, Department of Computer Science, Rice University, August 1996.
14. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of data structures in C*, W.H. Freeman and company, New-York, 1993.
15. F. Irigoin and R. Triolet, *Supernode Partitioning*, ACM SIGACT-SIGPLAN SPPL 88, 1988, pp. 319–329
16. B. Kruatrachue and T. Lewis, *Grain Size Determination for Parallel Processing*, IEEESOFT 5, 1, 1988, pp. 23–32
17. V. M. Lo and al., *OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures*, Parallel Computing, 1992, pp. 237
18. M. Loi, *Construction et exécution de graphes de tâches acycliques à gros grains*, PhD thesis, ENS Lyon, France, 1996.
19. V. Maslov, *Lazy array data-flow dependences analysis*, Conference Record of the 21st ACM SIGPLAN Symposium on Principles of Programming Languages, January 1994, pp. 311–325.

20. W. Pugh and D. Wonnacott, *An evaluation of exact methods for analysis of value-based array data dependences*, Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing (Portland, Oregon), August 1993.
21. S. Rajopadhye, *LACS: A language for affine communication structures*, Research Report IRISA PI-712, 1993
22. S. Rajopadhye and D. K. Wilde, *Allocating memory arrays for polyhedra*, Research Report IRISA PI-749, 1993
23. A. Schrijver *Theory of linear and integer programming*, John Wiley & sons, 1986.
24. 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.
25. D. K. Wilde, *A library for doing polyhedral operations*, Research Report IRISA PI-785, 1993
26. 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.
27. 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.