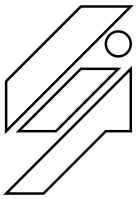


Laboratoire de l'Informatique du Parallélisme

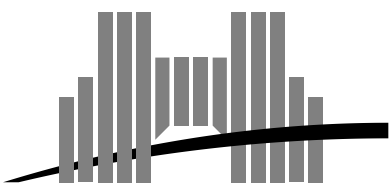
École Normale Supérieure de Lyon
Unité de recherche associée au CNRS n° 1398

Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs

Michel Cosnard
Emmanuel Jeannot
Laurence Rougeot

March 1998

Research Report N° 98-14



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

Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs

Michel Cosnard
Emmanuel Jeannot
Laurence Rougeot

March 1998

Abstract

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

Keywords: Control parallelism, coarse grain task graph, parameterized task graph, dynamic scheduling, memory optimization

Résumé

Ordonnancer de gros graphes de tches est un sujet important car il permet de traiter des problmes de grande taille. Dans ce rapport nous considrons le problme suivant : comment ordonnancer un graphe de tche quand celui-ci est trop gros pour tenir en mmoire? Notre rponse introduit le graphe de tche paramtr (GTP), qui est une reprsentation symbolique du graphe de tches. Nous proposons un algorithme d'ordonnancement qui prend le GTP en entre et qui permet de gnrer un programme gnrique. Nous tudions les performances de notre mthode ainsi que ses limitations. Nous montrons que notre algorithme trouve un bon ordonnancement pour les graphes de tches gros grain, qu'il est peu coteux en mmoire et qu'il a une bonne complexit temporelle. Quand le nombre moyen d'oprations de chaque tches est assez grand, nous prouvons que le surcot de l'ordonnancement est ngligeable par rapport au makespan. Nous tudions les aspects pratiques de notre approche sur plusieurs noyaux coteux en calcul que l'on trouve dans les applications scientifiques.

Mots-clés: Paralllisme de contrle, graphe de tches a1 gros grain, graphe de tches paramtr, ordonnancement dynamique, optimisation mmoire

Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs

Michel Cosnard
LORIA - INRIA Lorraine
615, rue du Jardin Botanique
BP 101
54602 Villiers Les Nancy
France

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

3rd March 1998

1 Introduction

As the computational power of distributed memory parallel computer increases the need to solve very large problems appears. In the control parallelism approach, computations are allocated onto processors in order to reduce communication costs between them. 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, 19], or not [27, 11, 17, 22]. In order to generate a parallel program, static schedulers, like Pyrros [26], need to have the complete task graph in memory. This is not a feasible solution when dealing with very large problems because the task graph is too large to fit into memory. Furthermore, a task graph is built when all the parameters have been instantiated. Thus, if the parameters change, the analysis of the sequential program has to be repeated and a new task graph is then, rebuilt. Hence, static task graph scheduling does not allow to build a generic program. In *Cilk* [3], the task graph is scheduled at run-time. Cilk can handle big size problems but, communication cost is not taken into consideration. The Cilk system performs very well for tree-like style computation (min-max search, backtrack exploration, etc...) but, it has not been designed for scientific loop-nest computations. In [2, 13] run time methods to schedule task graphs are described addressing the problem of processor memory requirement, but these works do not consider DAG memory requirement. In [1], a tool *CASCH*, is presented. It allows to generate a schedule and a parallel code for a sequentiel program. Nevertheless CASCH uses standard static scheduling algorithms and consequently, has the same drawbacks as Pyrros.

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

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

This report is organized as follows: Section 2, gives the definitions used along this report. Section 3 deals with the PlusPyr software. Section 4 presents some examples and their associated task graphs. Section 5, describes the dynamic approach and the PTGDS algorithm. In section 6 we study theoretical results concerning PTGDS behavior and our dynamic approach. Section 7 presents the experimental results we have obtained. Finally, in section 8 concluding remarks are given.

2 Definitions and models

All over this report, 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. We note v the number of nodes. In the following we will say node or task indifferently. E is the arc set. There is an arc 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 is the number of arcs. T is the task weight (or task duration) set , $\tau_i \in T$ will represent task i duration. C is the set of arc weights (or communication volumes), $c_{i,j} \in C$ represents the communication cost along the arc 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]:

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

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

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

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

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

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

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

3 PlusPyr the task graph builder

3.1 The parameterized task graph

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

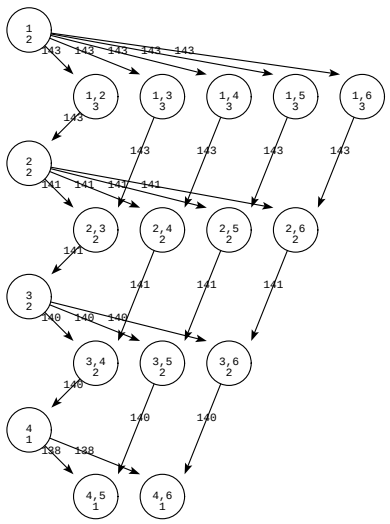


Figure 1: Gaussian Elimination and expanded task graph(n=5)

We have proposed the parameterized task graph [9] as a solution for automatically deriving task graphs from sequential programs. It uses parameters which have to be instantiated in order to build the expanded task graph. It is mainly composed of *generic tasks* and *communication rules*. A generic task is a set of instructions which have to be executed sequentially. These instructions are between two delimiters: *task* and *endtask*. The communication rules represent symbolic dependences between generic tasks. The Gaussian Elimination program, presented in Figure 1, has two generic tasks : one (*T1*) for the update of column k, the other one (*T2*) for the update of the sub-matrix.

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

Figure 2: Reception rules for the Gaussian Elimination

The communication rules have two forms. Reception rules define data received by generic tasks. Emission rules describe the data sent by a generic task. Figure 2 shows an example of the reception rules derived from the Gaussian Elimination program of Figure 1.

Let us take into consideration rule number 5:

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

This means that task $T2(k,j)$, for all $2 \leq k \leq n-1$ and all $k+1 \leq j \leq n+1$, will receive from $T1(k)$ the pivot column $A(i,k)$ for all $k+1 \leq i \leq n$. All these rules describe symbolically the communications handled by tasks *T1*, *T2*, during the program execution. They also give the symbolic value of the communication volume along the generic edge (for rule 5 it is $n-k$).

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

3.2 The input language

PlusPyr [18], a tool built by Michel Loi, is able, given a sequential program, to derive the parameterized task graph. PlusPyr is also able, once the values of the parameters are given, to construct the expanded task graph. There are some limitations concerning the PTG. In PlusPyr some static analysis are performed in order to build the parameterized task graph, for instance, to compute the symbolic values of the number of iterations of each loop. Therefore, all the array subscript expressions and FOR-loop bound expressions are integer-valued affine functions of program parameters, integer constants of enclosing loop indices. We shall not enter into many details about the constraints of the input language, for more details see [9]. The analysis performed by PlusPyr gives also the *symbolic computational cost* of each generic task. This analysis is based on the work done by Feautrier on integer parametric programming (see [12, 23] for an introduction). For instance PlusPyr computes that the cost of task $T1(k)$ (program Figure 1) is $2+2(n-k)$ (for more details see [9, 10]).

```
param n
real s, f
real a(n,n+1)
real x(n)
for i = 1 to n-1 do
  for j = i+1 to n do
    task                    /*T1(i,j)*/
      f = a(j,i) / a(i,i)
      for k = i+1 to n+1 do
        a(j,k) = a(j,k)-f*a(i,k)
      endfor
    endtask
  endfor
endfor

for i = 1 to n do
  task                      /*T2(i)*/
    s = 0.0
    for j = 1 to i-1 do
      s = s+a(n-i+1,n-j+1)*x(n-j+1)
    endfor
    x(n-i+1) = (a(n-i+1,n+1)-s)/a(n-i+1,n-i+1)
  endtask
endfor
```

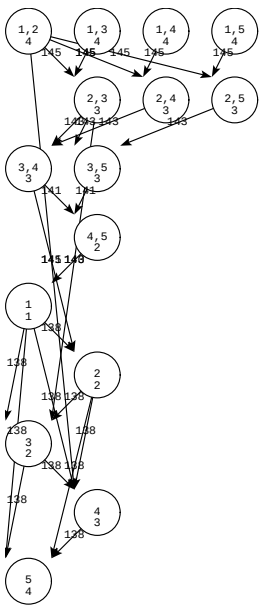


Figure 3: Gaussian Elimination, backward substitution and its task graph (n=5)

4 Some programs and their task graphs

```
param n
real a(n,n)
real c,s,d,a1,a2

builtin real r_sqrt()

for i = 1 to n do
  for j = i+1 to n do
    task                    /*T1(i,j)*/
      a1 = a(i,i)
      a2 = a(j,i)
      d = r_sqrt(a1*a1+a2*a2)
      c = a1/d
      s = a2/d
      for k = i to n do
        a1 = a(i,k)
        a2 = a(j,k)
        a(i,k) = c * a1 + s * a2
        a(j,k) = -s * a1 + c * a2
      endfor
    endtask
  endfor
endfor
```

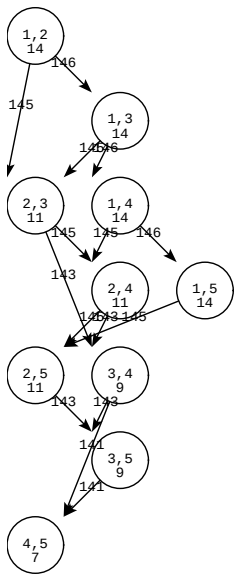


Figure 4: Givens algorithm and its task graph (n=5)

In this section we describe several programs and we show the task graph built by PlusPyr[1], for a small parameter value. Hence, we first demonstrate the feasibility of the PlusPyr software. Moreover, we have done extensive tests on the parallelization of these numerical kernels that will be presented in section 7.

Gaussian Elimination and backward substitution: the program in Figure 3 has two generic tasks: task T1 for the "ijk" Gaussian Elimination, and task T2, for solving the system by a backward substitution. This shows that we can easily analyze the concatenation of two programs.

The Givens algorithm: in Figure 4, we have a simple program, with only one generic task. It shows that complex computational costs can be handled.

Jordan Diagonalization Jordan method is decomposed into two generic tasks, one to diagonalize the matrix and the other one to solve the system. Note that in generic tasks $T1$ and $T2$, for-loops are sequentialized (see Figure 5).

Power of a matrix: in Figure 6, we compute the m^{th} power of matrix A (task T1), put the result in C (initialized by task T0), and we iterate a vector with each intermediate power (task T2). In this example, 3 parameters (n,m and l) are used. Hence, the computational cost and the granularity can be easily tuned.

[1] PlusPyr provides a graphical environment which allows to visualize the task graph

```
param n
real a(n,n)
real b(n)
assert n >= 3

for k = 1 to n do
  for j = k + 1 to  n do
C  update of col. j
    task                                /*T1(k,j)*/
      a(k,j) = a(k,j) / a(k,k)
      for i1 = 1 to k-1 do
        a(i1,j) = a(i1,j) - a(k,j) * a(i1,k)
      endfor
      for i2 = k + 1 to n do
        a(i2,j) = a(i2,j) - a(k,j) * a(i2,k)
      endfor
    endtask
  endfor
C  update of vector b
  task                                  /*T2(k)*/
    b(k) = b(k) / a(k,k)
    for i3 = 1 to k-1 do
      b(i3) = b(i3) - b(k) * a(i3,k)
    endfor
    for i4 = k + 1 to n do
      b(i4) = b(i4) - b(k) * a(i4,k)
    endfor
  endtask
endfor
```

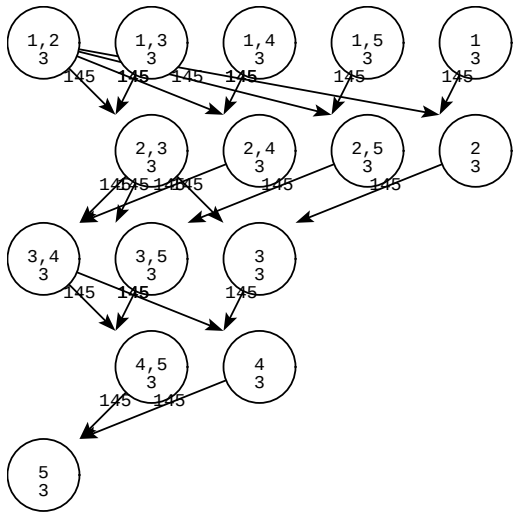


Figure 5: Jordan diagonalization and its task graph (n=5)

5 Parameterized task graph dynamic execution

In the literature, dynamic execution policy is mainly used in the following two cases : (1) dealing with nondeterministic programs [25, 3], (2) load balancing and process migration [24, 21]. Here we use dynamic execution for the following reasons:

1. to obtain a generic parallel program for each sequential program : we give the parameters value at run-time, and then schedule the tasks during the execution. We use the parameterized task graph because it is problem size independent,

2. to handle big size problems. It is impossible to have a very large task graph in memory. The parameterized task graph allows to build at run-time, the small part of the DAG needed to perform task mapping optimization.

5.1 PTGDS, the scheduling algorithm

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

- We use the code analysis performed by *PlusPyr* to determine the duration of T.

```
param m,n,l
assert m>=1
assert n>=1
assert l>=1
real a(n,n),b(n,n),c(n,n),v(l,n),w(l,n)

task                          /*T0*/
  for i=1 to n do
    for j=1 to n do
      c(i,j)=a(i,j)
    endfor
  endfor
endtask

for q=2 to m do
  for i=1 to n do
    task                      /*T1(q,i)*/
      for j=1 to n do
        b(i,j)=0
        for k= 1 to n do
          b(i,j)=b(i,j)+c(i,k)*a(k,j)
        endfor
      endfor
      for j=1 to n do
        c(i,j)=b(i,j)
      endfor
    endtask
  endfor
C Compute V<-C.V
  task                        /*t2(q)*/
    for i=1 to n do
      for j=1 to l do
        w(i,j)=0
        for k= 1 to n do
          w(i,j)=w(i,j)+b(i,k)*w(k,j)
        endfor
      endfor
      for j=1 to l do
        v(i,j)=w(i,j)
      endfor
    endfor
  endtask
endfor
```

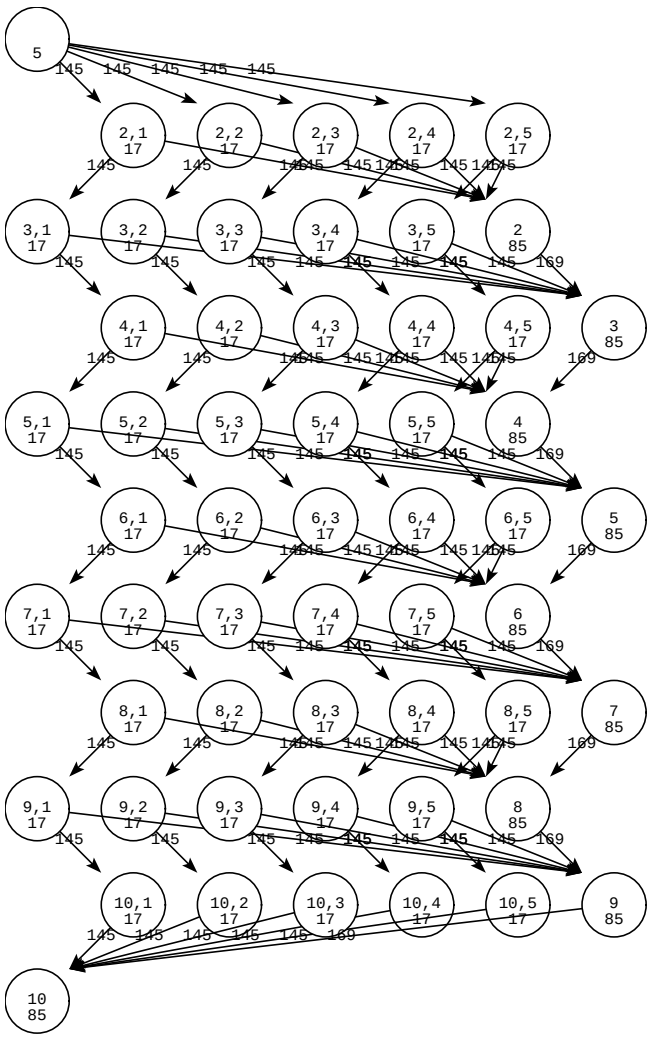


Figure 6: Power of a matrix and its task graph (m=10, n=5)

```
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 7: The PTGDS algorithm

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

Figure 7 gives the general scheme of PTGDS algorithm.

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

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

5.2 Combining scheduling and dynamic execution

We build an SPMD generic program which uses a supervisor/executor protocol. One supervisor executes PTGDS, and sends orders to the processors. There is one executor per processor. We assume that the supervisor is executed on the parallel machine host. Each executor receives messages from the supervisor. Those messages may to execute a task or to send data to an other processor. For instance, when a task T is assigned to processor P, the supervisor orders the executors to send to P the data needed for the computation of T. Receptions of messages are handled independently by each executor. The execution is dynamic, since the supervisor sends orders to the executors while scheduling the task graph. Such a complete dynamic execution of the parameterized task graph will be called PTGDE (Parameterized Task Graph Dynamic Execution). Figure 8 gives the code of the supervisor and one executor. All the executors have the same code: the sequential generic task, and the communication protocol.

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

```
1  schedule(task T){
2  for each task T' in father(T) do
3    if not(allocated(T')) then schedule(T');
4  endfor
5  send the order of executing T to the executor which,
according to PTGDS, minimizes its starting time;
6  send, to executors which execute the fathers of task
T, the order of transmitting data to the executor which
execute T;
7  for each task T' in father(T) do
8    if T is the last son of T' to be scheduled then
9      remove from memory all information on T'
10   endif
11 endfor
12 allocated(T)=true;
13 }
```

```
1  while true do{
2   case type_message of
3    execute :
4     if all the required data are in memory
5      then execute the task; send new computed data if
required (check send_list);
6           If possible execute a task with the new data
computed (check exec_list);
7       else store this message in exec_list;
8    send :
9     if all the required data are in memory
10     then send the data;
11    else store this message in send_list;
12    receive :
13     store the data in memory;
14     execute task if possible (check exec_list);
15  }
```

Figure 8: PTGDE code. Left: the supervisor; Right: an executor

or new data are computed. In *send_list* are stored the send messages that cannot be performed because the data have not already been computed by the processor. This list is updated when new data are computed.

6 Theoretical results

In this section we show that it is possible to derive a simple upper bound on the average number of operations of tasks in order to obtain an efficient execution on a parallel computer. Theorem 1 shows that the time taken by PTGDS to find a schedule is competitive in comparison to other static scheduling algorithms (an extensive study of scheduling algorithms is done in [17]). Theorem 2 shows that PTGDS produces good schedules when dealing with coarse grain task graphs on an unbounded number of processors. The main limitation of our approach is that dynamic scheduling is costly when dealing with fine grain task graphs. Theorem 3 gives a sufficient condition on the average number of operation of tasks such that, during the the parameterized task graph dynamic execution, the scheduling time is negligible with comparison to the parallel time.

6.1 PTGDS computational complexity

As shown in the following theorem PTGDS computational complexity is very competitive. Indeed, it is almost linear in the number of edges, which is the best possible complexity since all the edges should be traversed at least once in a scheduling algorithm.

Theorem 1 *PTGDS computational complexity 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: Complete proof can be found in [7, 8]. This proof is done in counting the number of time each line of the is executed, and to compute the cost of each line. It takes $O(e+v)$ to visit the whole graph. Each time we have to allocate a task it takes $O(p)$ to choose the right processors. Finally it takes at most $O(\log v)$ to find informations about a task in the AVL tree. ∎

6.2 Bound on the parallel time

We prove here that, for coarse grain task graphs ($g(G) \geq 1$), the parallel time produced by PTGDS is always within a bound of two of the optimal, for an unbounded number of processors.

Definition 1 *Let G be a task graph. We call $PT_{opt}(G)$ the optimal parallel execution time (optimal parallel makespan for G on an unbounded number of processors), and $PT_P(G)$ the execution time of G scheduled by PTGDS on an unbounded number of processors, i.e. the makespan of the parallel program taking into account only the executors.*

Theorem 2 *For any task graph G, the execution time of G scheduled by PTGDS is within a bound of $(1+\frac{1}{g(G)})$ of the optimal parallel execution time:*

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

Sketch of proof of theorem 2: Complete proof can be found in [8]. The main idea is that, for any task graph G, there exists a linear clustering (lc)such that $PT_P(G) \leq PT_{lc}(G)$. Furthermore, Gerasoulis and Yang in [15] proved that for any linear clustering, on an unbounded number of processors, we have: $PT_{opt}(G) \leq PT_{lc}(G) \leq (1 + \frac{1}{g(G)}) PT_{opt}(G)$. ∎

6.3 Minimizing the impact of the scheduling overhead

Definition 2 *Let us call A_{op}, the average number of operations of tasks in G, i.e. $A_{op} = \frac{\sum_{i \in V} \tau_i}{\omega v}$. Let $PTGDE(G,p)$ be the parallel program execution time, of the parameterized task graph G scheduled by PTGDS, on a p processors parallel computer. Let $T_{sup}(G,p)$ be the time taken by PTGDS on the supervisor to schedule G on p processors.*

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

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

In many cases (particularly, in all the section 4 examples), the number of edges of the task graph is on the same order as the number of nodes. In the following theorem we prove that if average number of operations of tasks is high then, the scheduling overhead is negligible in comparison to the makespan.

Theorem 3 *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 3 Let $T_{sup}(G,p)$ be the supervisor time.

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

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

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

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

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

Then, it exists some constant k such that:

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

We cannot have a superlinear speedup, thus:

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

A_{op} is the average number of tasks operation, then

$$A_{op} = \frac{T_{seq}}{v\omega}$$

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

$$\begin{aligned} pm = o(A_{op}) &\Rightarrow pm = o(\frac{T_{seq}}{v\omega}) \\ &\Rightarrow kpm = o(\frac{T_{seq}}{v\omega}) \\ &\Rightarrow kvm\omega = o(\frac{T_{seq}}{p}) \end{aligned}$$

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

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

■

7 Results and experiments

For all the examples described in section 4 we have performed some experiments to validate our approach. First, we have done speedup simulations to show the accuracy of the schedules found by PTGDS. Second, we have studied PTGDS memory cost: for each DAG, we have compared its number of tasks to the maximum number of tasks in memory during the execution of PTGDS. Third we have studied the time taken by PTGDS to execute, and compared it to the parallel and sequential time of the program.

As shown in section 3, PlusPyr gives, for each generic task, the symbolic value of computational cost and communication volume. Hence, once the parameters value are given, PTGDS knows the number of operations done by each task and the number of items sent between tasks.

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

7.1 Speedup simulation

In Figure 9, the simulations show that the speedup is at least 27 for all examples on 32 processors, when the matrix size is 1000. When the matrix size is small, PTGDS shows limitation in term of speedup. This is due to the difficulty to use all the processors when the number of tasks is small.

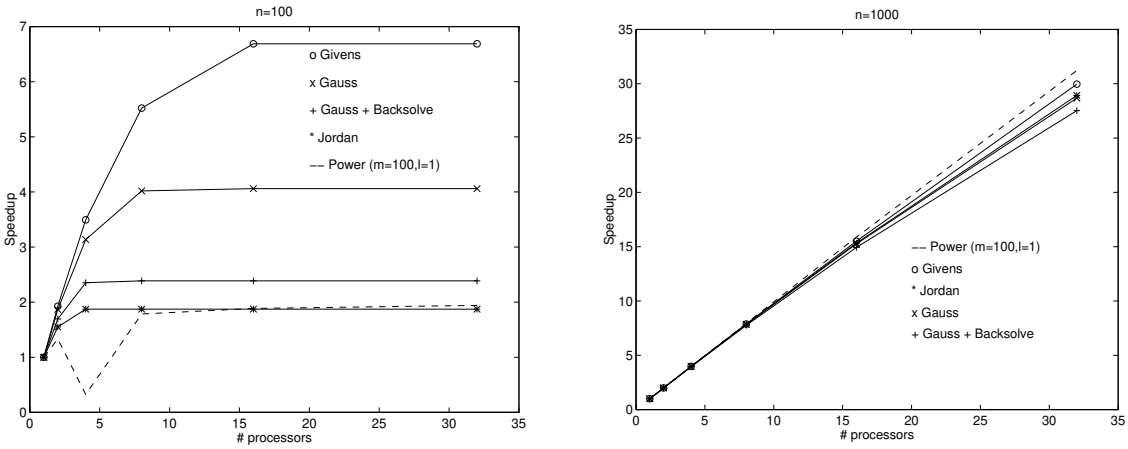


Figure 9: Speedup simulation vs. number of processors for several examples and several matrix sizes

7.2 Memory cost

Table 1: Memory cost of main data structures of PTGDS (n=100)

Program	# Tasks	Max # tasks	# Nodes	Max # nodes	# Edges	Max # edges
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

Table 2: Memory cost of main data structures of PTGDS (n=1000)

Program	# Tasks	Max # tasks	# Nodes	Max # nodes	# Edges	Max # edges
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

In Tables 1 and 2 we study the memory cost of PTGDS for several task graphs. We have instantiated parameters value and run PTGDS on the expanded graph. The column *# Tasks*, is the number of tasks in the DAG. PTGDS optimizes the number of tasks in memory. The list of tasks needed to allocate other tasks is handled by an AVL tree. Column *Max # tasks* is the maximum size of the AVL during the execution. The parameterized task graph is a description of the edges in the task graph. Most of the time an edge in the task graph is generated by several rules. For instance rules 1 and 2 in Figure 2 generate the same set of edges in the task graph. Hence, when we use the communications rule to explore the DAG, the program generates more edges and tasks than there are in the DAG. Our program ensures the schedule correctness by checking duplicated edges and tasks. However, it is necessary to use all the rules in order to count the amount of data sent between tasks. Column *# Nodes* gives the number of tasks generated by the recursive DAG exploration.

Column *Max # nodes* is the maximum number of tasks in memory during the execution. Column *# Edges* is the number of edges generated by the recursive DAG exploration. Column *Max # edges* is the maximum number of edges in memory during the execution. Edges and tasks are removed from memory when the corresponding recursive step is finished.

For example, let us consider the first line of Table 2. In the Gaussian Elimination DAG (n=1000) there is 501500 nodes/tasks, but our algorithm manipulates at the same time at most 2000 of them. The recursive exploration is done using the communication rules. Lines 1 and 2 of Figure 2 are two different rules that describe the same edges (the first one for the first element of column k, the second for remaining of column k). Since both rules are considered during the DAG exploration, these edges are generated twice, as well as the corresponding nodes. We check duplicates to ensure program correctness. By the way, there is 2005993 such nodes generate but only 1999 at the same time in memory, and 1503498 of such edges but only 3998 at the same time in memory. In general the number of nodes generate by the recursive exploration is about 4 time the real number of tasks because rules 1 and 2, and rules 3 and 4 generates duplicates of generic task $T2$.

These three data structures are the only one which vary with the parameters value.

Tables 1 and 2 show 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 Table 1 (matrices of size 100) and Table 2 (matrices of size 1000) show that required memory increases linearly, while (except for *Power*), the size of the DAG increases quadratically. Concerning the computation of the m^{th} power of a matrix of size n (Figure 6), further investigations have shown that required memory and the size of the DAG is proportional to mn.

7.3 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.

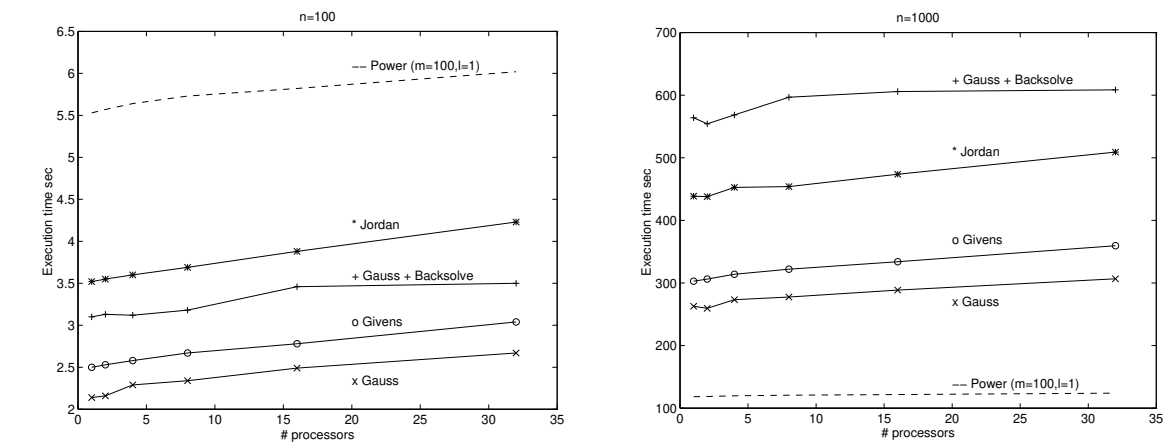


Figure 10: Execution time of PTGDS vs. number of processors for several matrix sizes

In Figure 10, 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.

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

7.4 Comparison between scheduling overhead and execution time

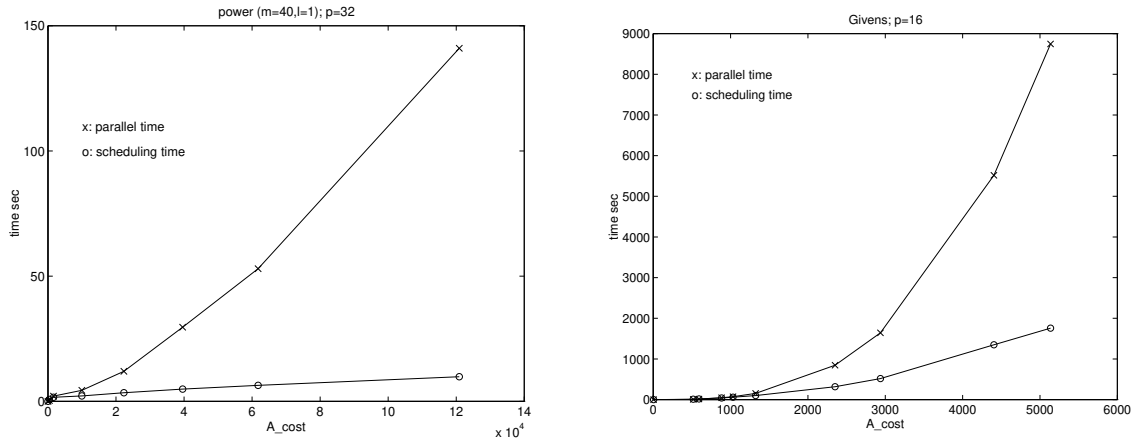


Figure 11: Execution time of PTGDS and execution time simulation vs. average number of operations of tasks

In this section we show that, according to theorem 3, the time taken by PTGDS to schedule the parameterized task graph is negligible when the average number of operations of tasks is big. In Figure 11 we have compared the time that would take *Power* and *Givens* to execute on a 32 processors machine having the characteristics given in this section, and the time taken to schedule the DAG, when the average number of operations of tasks increases. For *Power* we see that when $p^2 \approx 50.A_{op}$ then $PT(G) \approx 10.PTGDS(G)$. For the Givens algorithm we see that when $p^2 \approx 20.A_{op}$ then $PT(G) \approx 5.PTGDS(G)$.

8 Conclusion

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

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

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

9 Acknowledgments

This work is part of the European Community Eurêka EuroTOPS project. We thank Michel Loi for providing us with the PlusPyr software and Tao Yang of UCSB for valuable comments about this report .

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

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

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

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

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

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

[19] M.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.

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

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

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

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

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

[25] D. Towsley. Allocating Programs Containing Branches and Loop within a Multiple Processor System. *IEEE Transactions on Software Engineering*, 12(10):1018–1024, 1986.

[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.