Laboratoire de l'Informatique du Parallélisme

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

Symbolic Partitioning and Scheduling of Parameterized Task Graphs

Michel Cosnard
Emmanuel Jeannot
Tao Yang

September 1998

Research Report N° 1998-41

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`

Symbolic Partitioning and Scheduling of Parameterized Task Graphs

Michel Cosnard
Emmanuel Jeannot
Tao Yang

September 1998

Abstract

A parameterized task graph is a model of computation which is small and problem size independent (i.e. it requires the same amount of memory, whatever the size of the program parameters is). Here, we discuss a method that allows to statically (at compile time) schedule such a parameterized task graph. Therefore, we find an allocation algorithm which memory complexity and computational complexity are $O(1)$. Once the symbolic allocation is computed we describe a distributed method to execute the program in a multithreaded environment and we show preliminary result on the efficiency of a parallel program built using the framework described in this paper.

Keywords: Static scheduling, parameterized task graph, thread, control parallelism, control static program

Résumé

Un graphe de tâches paramétré est un modèle de calcul qui est petit et indépendant de la taille des données (i.e. il requière la même quatité de mémoire quel que soit la taille des paramètres du programme). Nous présentons ici une methode qui permet d'ordonnancer statiquement (à la compilation) de tels graphes de tâches. Nous trouvons ainsi un algorithme d'allocation qui a une complexité mémoire et temporelle en $O(1)$. Nous décrivons une méthode distribuée qui, une fois cette allocation symbolique calculée, permet d'exécuter le programme dans un environnement multithreadé. Nous montrons des premiers résultats sur l'efficacité du programme parallèle construit suivant le schéma décrit dans ce papier.

Mots-clés: Ordonnancement statique, graphe de tâches paramétré, processus léger, parallélisme de contrôle, programme à contrôle statique

Symbolic Partitioning and Scheduling of Parameterized Task Graphs

Michel Cosnard
LORIA INRIA Lorraine
Michel.Cosnard@loria.fr

Emmanuel Jeannot
LIP ENS-Lyon
Emmanuel.Jeannot@ens-lyon.fr

Tao Yang
UCSB
Tao.Yang@cs.ucsb.edu

16th September 1998

1 Introduction

In order to have good performances a parallel program must have a suitable granularity. Fine grain applications cannot achieve good performances because the execution overhead is too big. On the other hand when the grain is very large all the parallelism is lost. In order to have a better granularity three approaches have been addressed in the literature:

1. **Data aggregation**. The data are grouped (by block, column, row,...) on a same processor in order to reduce communications. Computations are then executed where are the data using the *owner compute rule*. This is the approach taken, for instance, in HPF [3]
2. **Tiling**. In this approach loop nests are modified in order to have the parallelism in the outermost loops, and then to execute a set of operations on the same processor, gathering the communication into one when necessary. For an introduction see [11, 18].
3. **Task parallelism**. The two former approaches fit into the data-parallelism approach (one control flow). In the task parallelism approach, first a sequential program is decomposed into tasks, second dependences between the tasks are found, third a schedule of the tasks onto the processors is computed, finally each task is executed on its processor.

In this paper we will address the task parallelism approach for compute intensive kernel programs.

A lot of work has been done, in the literature, in order to schedule task graphs[4], either allowing the duplication of tasks on the processors [15, 16], or not [7, 12]. All the algorithms have a memory complexity and a computational complexity that depend on the size of the task graph. Therefore, the need to treat very large problem (very large task graph), makes the use of standard task graph scheduling algorithm critical. As a matter of fact, a big task graph may not fit into memory and is very time costly to schedule. In this paper we use the parameterized task graph (PTG) instead of the expanded task graph. The PTG is a symbolic representation of the task graphs. We present a method which allows to schedule the parameterized task graph. We obtain a symbolic allocation in constant time and memory cost, whatever the program parameter value is. We describe a distributed method to execute the symbolic allocation in a multithreaded environment and we show some preliminary results on a real program.

2 Background and Scope of this Paper

As a frontend of our method we use the PlusPyr tool built by Michel Loi [13].

PlusPyr is able, given an annotated sequential program, to derive the parameterized task graph (PTG). The PTG is a symbolic representation of the task graph, and has a size which is constant, whatever the sequential program parameter value is. PlusPyr is also able, once the values of the parameters are given, to construct the expanded task graph. The analysis performed by PlusPyr also gives the *symbolic computational cost* of each generic task. This analysis is based on the work done by Feautrier on integer parametric programming (see [8, 19] for an introduction) and [5, 6], for more details.

PlusPyr is a suitable tool for analyzing compute-intensive kernels that can be found in scientific programs. In this paper we target programs on dense matrices with static control (loop bounds and array references are affine function of loop indices), which are, in many applications, the most time-consuming part. The programs are annotated by the keyword *task* and endtask in order to delimit the parts of the program that have to be executed in sequential.

In this paper we will use the program of Figure 1 as a running example.

```
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
```

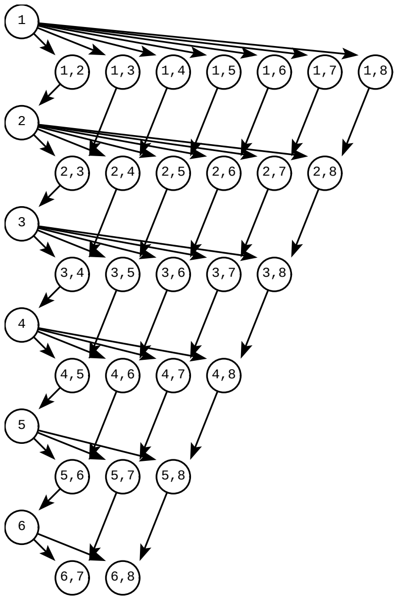

Figure 1: Gaussian Elimination and expanded task graph(n=7)

This program, a straightforward implementation of the Gaussian Elimination, has static control and has two generic tasks $T1(k)$ and $T2(k, j)$. Its parameterized task graph, computed by our tool PlusPyr, is shown in Figure 2

3 Definitions and Notations

We call parameterized task graph the following tuple $PTG = \{T, R, D\}$, where:

- T is the set of generic tasks
- R is the set of communication rules,

1. $T1(k) \longrightarrow T2(k,j) : A(i,k) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$
2. $T2(k,j) \longrightarrow T1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$
3. $T2(k,j) \longrightarrow T2(k+1,j) : A(i,j) | 1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n$

Figure 2: Emission rules for the Gaussian Elimination

- C is the cost function of each task.

A generic task is defined by its name, its iteration vector, and its code.

R is composed of the reception rules and the emission rules. Reception and emission rules are dual forms of each other. Reception rules describe the set of fathers of a given task. Emission rules describe the set of sons of a given task. Most of the time we will deal with emission rules, but keep in mind that for each emission rule we also have the reception rule.

A emission rule has the form:

$$Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y}) | P$$

This means that if predicate P is true, task $Ta(\vec{u})$ sends data $D(\vec{y})$ to task $Tb(\vec{v})$. With each rule is associated polyhedron P which described the valid values of each iteration vector: it is a predicate on the variables $\vec{u}$, $\vec{v}$ and $\vec{y}$.

Let $\vec{\imath}$ be the vector composed of all the parameters and the loop indices taken in lexicographic order.

A rule is said a **bijection rule** if, for each instance of iteration vector $\vec{u}$ it exists exactly one instance of iteration vector $\vec{v}$, associated to $\vec{u}$ by the emission rule.

Tasks are nested into loops, their iteration vectors are affine functions of the program parameters and loop indices. Thus, it exists two matrices $D1$ and $D2$ such that:]

$$\vec{u} = D_1\vec{\imath} + \vec{k_1}$$

and

$$\vec{v} = D_2\vec{\imath} + \vec{k_2}$$

Where $\vec{k_1}$ and $\vec{k_2}$ are constant vectors.

To find a bijection rule we shall proceed as follow : build D_1 and D_2. Then find E_1 and $\vec{w_1}$ such that $\vec{\imath} = E_1(\vec{u} - \vec{k_1}) + \vec{w_1}$. It exists many E_1 and $\vec{w_1}$ that verifies the former equation. We impose that $\vec{w_1}$ is composed of the indices that does not appear in $\vec{u}$. Formally, $w_{1_j} = \delta_j i_j$, where the w_{1_j} (resp. i_j) is jth element of $\vec{w_1}$ (resp. $\vec{\imath}$); $\delta_j = 0$ if i_j appears in $\vec{u}$, and $\delta_j = 1$ if i_j does not appear in $\vec{u}$. Hence, we have: $\vec{v} = D_2E_1(\vec{u} - \vec{k_1}) + D_2\vec{w_1} + \vec{k_2}$. On the other hand we can build E_2 and $\vec{w_2}$ such that : $\vec{u} = D_1E_2(\vec{v} - \vec{k_2}) + D_1\vec{w_2} + \vec{k_1}$. The rule is a bijection rule if for each instance of $\vec{u}$ there is only one instance of $\vec{v}$ and for each instance of $\vec{v}$ there is only one instance of $\vec{u}$. This is true if $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant.

In Figure 2, rule number 1 is a not a bijection rule. Indeed, $\vec{u} = (k)$ and $\vec{v} = \begin{pmatrix} k \\ j \end{pmatrix}$. The vector of all the loops indices is $\vec{\imath} = \begin{pmatrix} k \\ l \\ j \\ i \end{pmatrix}$. Therefore $D_1 = (1,0,0,0)$ and $D_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$.

We deduce that: $E_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ and $\vec{w_1} = \begin{pmatrix} 0 \\ l \\ j \\ i \end{pmatrix}$ which are such that $\vec{\imath} = E_1\vec{u} + \vec{w_1}$. Thus, $D_2\vec{w_1} = \begin{pmatrix} 0 \\ j \end{pmatrix}$ is not constant.

On the other hand, rule number 2 is a bijection rule. Indeed, $\vec{u} = \begin{pmatrix} k \\ j \end{pmatrix}$ and $\vec{v} = (k+1)$. Since $j = k+1$, we have $\vec{u} = \begin{pmatrix} k \\ k+1 \end{pmatrix}$. Thus, $D_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ and $D_2 = (1,0,0,0)$. We deduce that: $\vec{w_1} = \vec{w_2} = \begin{pmatrix} 0 \\ l \\ j \\ i \end{pmatrix}$. Therefore, $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant.

It is also easy to check that rule 3 is a bijection rule.

We define a **cluster** as a set of tasks which have to be executed on a same processor. We use the following notation: $\kappa(Ta, \vec{v})$, is a *function*, that gives for all the vectors $\vec{v}$ in polyhedron P, the cluster number of task Ta.

A cluster is said **linear** if there is no independent tasks in it.

The **zeroing process** is defined as follows. The rule $Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y})|P$ is said *zeroed* if and only if $\kappa(Ta, \vec{u}) = \kappa(Tb, \vec{v})$ for all the valid instances of $\vec{u}$ and $\vec{v}$ in P. Intuitively, when a rule is zeroed every couple ($Ta(\vec{u})$,$Tb(\vec{v})$) will be put on the same processor leading to a *zero* cost communication between them.

4 Overview of the Method

We describe here the steps needed to compute the scheduling of a parameterized task graph:

1. We first look at all the bijection rules in order to derive a symbolic linear clustering. Symbolic clusters are built using the zeroing process. The symbolic linear clustering zeroes rules in such a way that, the instantiated clusters will always be linear (whatever the parameters value).

2. When the symbolic linear clustering is done, the second step is the identification of each cluster. The identification consists in finding a function such that, when the parameters value is known, it gives a cluster number to each task according to the linear clustering computed in the previous step.

3. The third step is to derive the mapping and packaging of the data. This is done using the communication rules. When a rule is not zeroed this means that data will be sent from a processor to an other. The rules describe which kind of data (scalar, vector, matrix,...), is sent. We use these informations for generating the massage packaging code.

4. The final step deals with the execution of the program. Our parallel program is a multi-threaded SPMD one. We cyclically map the clusters among the processors. Concerning the tasks, each node of the parallel machine manages two queues. The waiting queue, for the tasks waiting for data and the ready queue for tasks ready for execution. Each node executes n threads (n is fixed at the beginning of the execution). Each thread scans the ready queue, executes the ready task and sends data to the processors.

```
Input: a parameterized task graph PTG={T,R,D}
Output: The set zeroed_rule_set of the rule that have to be zeroed.
zeroed_rule_set:=∅
bijection_rule_set:=compute_bijection_rule(R)
R:=sort_bijection_rule(R)
foreach r in bijection_rule_set do
  if not_in_conflict(r,zeroed_rule_set) then
    zeroed_rule_set+=r
  endif
enddo
```

Figure 3: The Zeroing Algorithm

5 Linear Clustering

Yang and Gerasoulis have proven that a linear clustering provides good schedules on an unbounded number of processors. If g is the granularity task graph G, PT_{lc} is the parallel time any linear clustering of G and PT_{opt} is the optimal parallel time of G, then on an unbounded number of processors [10]:

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

We wish to build a symbolic linear clustering of the parameterized task graph. This means that we are going to zero some bijection rules. This is done by the algorithm of Figure 3.

The zeroing algorithm proceeds as follow. It considers all the bijection rules. It sort them such that the rules implying more communication will be zeroed first. The ordering of the rules is done by taking into consideration the dimension of the sent data (i.e. a scalar, a row or a column, a block of matrix). Then it checks if a given rule is not in conflict with a rule already zeroed. Two rules are in conflict if, once zeroed, the produced clustering is not linear. If the rule is not in conflict with any other zeroed rule then the algorithm adds this rule to the set of zeroed rules. The algorithm repeats this process until no more rule can be zeroed.

In the two following sections we discuss the conflict problem and the identification of a rule.

5.1 The Conflict Problem

Let us consider $R1$ and $R2$, the two following emission rules:

$$R1 : Ta(\vec{u}) \longrightarrow Tb(\vec{v})|P_1$$

$$R2 : Tc(\vec{w}) \longrightarrow Td(\vec{z})|P_2$$

Suppose R1 is already zeroed. We want to check if R2 is not in conflict with R1. There is two kind of conflict: the fork conflict and the join conflict:

- We have a *fork conflict* if:
  1. $Ta = Tc$
  2. it exists an instance of $\vec{u}$ called $\vec{u_1}$, $\vec{u_1} \in P_1$ and an instance of $\vec{w}$, called $\vec{w_1}$, $\vec{w_1} \in P_2$ such that $Ta(\vec{u_1}) = Tc(\vec{w_1})$,

3. Let $\vec{v_1}$ the image of $\vec{u_1}$ according to rule R1 and $\vec{z_1}$ the image of $\vec{w_1}$ according to rule R2. such that $Tb(\vec{v_1}) \neq Td(\vec{z_1})$

- We have a *join conflict* if:

  1. $Tb = Td$
  2. it exists an instance of $\vec{v}$, called $\vec{v_1}$, $v_1 \in P_1$ and an instance of $\vec{z}$ called $\vec{z_1}$, $\vec{z_1} \in P_3$ such that $Tb(\vec{v_1}) = Td(\vec{z_1})$
  3. Let $\vec{u_1}$ the source of $\vec{v_1}$ according to rule R1 and $\vec{w_1}$ the source of $\vec{z_1}$ according to rule R2. such that $Ta(\vec{u_1}) \neq Tc(\vec{w_1})$

In order to automatically compute if two rules are in *fork conflict* (the *join conflict* case is symetric) we proceed as follows. Concerning step 2, we compute the set I, which is the intersection of all the tasks $Ta(\vec{u}), \vec{u} \in P_1$ with all the tasks $Tc(\vec{w}), \vec{w} \in P_2$. If I is not empty then step 2 is verified. Concerning step 3, we need $Tb = Td$ and we compute if $R1$ restricted to I is equal to $R2$ restricted to I, if not then the $R1$ and $R2$ are in *fork conflict*. We can compute I, the restriction of $R1$ and $R2$ to I symbolically using the *omega test* program of the university of Maryland[1].

Let come back to our example. Rule 2 and rule 3 can be in fork conflict. But rule 2 is valid if $j = k+1$ and rule 3 is valid if $k+2 \leq j \leq n+1$. Hence there is no instance of $T2(k,j)$ which have two sons: both $T2(k+1,j)$ and $T1(k+1)$. So rule 2 and 3 are not in conflict and can be can be both zeroed.

The zeroing algorithms build a set of bijection rules which are never in conflict.

Hence, we have the following theorem:

Theorem 1 *Let $G = (T, R, D)$ a PTG where R is composed of bijection rule and there is no rule in conflict then zeroing all the rules build linear clustering of G.*

Proof of theorem 1 Each rule is a bijection rule. Therefore, zeroing a rule leads, in the instantiated task graph, to zero exactly one edge for each instance of the rule.

No rules are in conflict. Therefore, zeroing all the rules leads, in the instantiated task graph, to a clustering with no independent tasks in the same cluster. ∎

5.2 Identification

In this section we show how to build a function $\kappa(Ta, \vec{u})$ that, for any generic task Ta and any valid instance $\vec{u}$ gives a cluster number, such that all the tasks in the same cluster will have the same number.

Let us consider rule:

$$R1 : Ta(\vec{u}) \longrightarrow Tb(\vec{v}) | P_1$$

If this rule is zeroed then, in P_1, we must have:

$$\kappa(Ta, \vec{u}) = \kappa(Tb, \vec{v}) \tag{1}$$

The basis of our method is the same as the one used by Feautrier in [9]. As Feautrier we will suppose that $\kappa(Ta, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a}$, where $\vec{\alpha_a}$ is a vector of unknowns which have to be found. We also have $\kappa(Tb, \vec{v}) = \vec{\alpha_b}\vec{u} + \vec{\beta_b}$ Then solving equation 1 leads to $\vec{\alpha_a}\vec{u} + \vec{\beta_a} = \vec{\alpha_b}\vec{v} + \vec{\beta_b}$.

[1] website:`http://www.cs.umd.edu/projects/omega`

Let us return to our running example. Rule 2 is : $T2(k,j) \longrightarrow T1(k+1) : A(i,k+1)|1 \leq k \leq n-2, j=k+1, k+1 \leq i \leq n$. Hence we have $\kappa(T2,(k,j)) = \alpha_{2,1}k + \alpha_{2,2}j + \beta_2$ and $\kappa(T1,(k+1)) = \alpha_{1,1}(k+1) + \beta_1$. This leads to

$$\left\{ \begin{array}{l} \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 = \alpha_{1,1}(k+1) + \beta_1 \\ j = k+1 \end{array} \right. \tag{2}$$

Equation 2 is satisfied for: $\alpha_{2,1} + \alpha_{2,2} = \alpha_{1,1}$ and $\beta_2 + \alpha_{2,2} = \beta_1 + \alpha_{1,1}$

As far as rule 3 ($T2(k,j) \longrightarrow T2(k+1,j) : \{A(i,j)|1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n\}$) is concerned we have:

$$\alpha_{2,1}k + \alpha_{2,2}j + \beta_2 = \alpha_{2,1}(k+1) + \alpha_{2,2}j + \beta_2 \tag{3}$$

Equation 3 is satisfied for $\alpha_{2,1} = 0$.

This leads to the following system:

$$\left\{ \begin{array}{l} \alpha_{2,1} + \alpha_{2,2} - \alpha_{1,1} = 0 \\ \beta_2 + \alpha_{2,2} - \beta_1 - \alpha_{1,1} = 0 \\ \alpha_{2,1} = 0 \end{array} \right. \tag{4}$$

System 4 has many solutions. In particular this one:

$$\left\{ \begin{array}{l} \alpha_{2,2} = \alpha_{1,1} = 1 \\ \beta_1 = \beta_2 = \alpha_{2,1} = 0 \end{array} \right. \tag{5}$$

Thus, one clustering function κ is:

$$\kappa(T2,(k,j)) = j$$

and

$$\kappa(T1,(k)) = k$$

However, this might not be always that simple. Indeed, two rules may lead to a contradictory condition on the $\vec{\alpha}$'s and the $\vec{\beta}$'s. Let us look, for instance, at the 2 following rules:

$$R1 : T1(k,j) \longrightarrow T1(k+1,j)|1 \leq k \leq n, 1 \leq j \leq k$$

$$R2 : T1(k,j) \longrightarrow T1(k,j+1)|1 \leq k \leq n, k+1 \leq j \leq n$$

Rule R1 leads to $\alpha_{1,1} = 0$ but rule R2 leads to $\alpha_{1,2} = 0$.Since we do not want the solution $\vec{\alpha_a} = 0$ for any task Ta, this solution is not feasible. However, we know that these two rules are not in conflict, the two functions can be applied to the two disjointed polyhedra, removing the incompatibility:

$$\kappa(T1,(k,j)) = \left\{ \begin{array}{ll} j & \text{if } 1 \leq k \leq n, 1 \leq j \leq k \\ k & \text{otherwise} \end{array} \right.$$

In the general case we will proceed as follows: We first build a PTG composed with the rules selected by the zeroing algorithm. Since there is no conflict between the tasks all the edges can be zeroed. However, as in the previous example, multiple edges may induce difficulties (an edge is said multiple between generic tasks Ta and Tb if there is at least one other edge between Ta and Tb). We compute the symbolic clustering with the following algorithm:

1. Let $G = (T, R, D)$ be the PTG to be clustered.

2. Compute the system $\mathcal{S}$:

$$C\alpha_G = 0$$

Where α_G is the vector whose components are all the unknowns $\vec{\alpha_a}$ and $\vec{\beta_a}$ for tasks $Ta \in T$. C is obtained by algebraic manipulations of the equations of the form $\vec{\alpha_a}\vec{u} + \vec{\beta_a} = \vec{\alpha_b}\vec{v} + \vec{\beta_b}$ for any rule $R1 : Ta(\vec{u}) \longrightarrow Tb(\vec{v})|P_1$, $R_1 \in R$.

3. If system $\mathcal{S}$ admits a non trivial solution $\alpha_G \neq 0$ then the clustering function associated to G is: $\kappa_G(Ta, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a}$.

If the only solution of $\mathcal{S}$ is $\alpha_G = 0$ and at least one node in G is of outdegree at least two then we split G. The splitting is done as follows:

(a) If it exists n multiple rules $(n \geq 2)$ $M_1, \ldots, M_n$ such that

$$\begin{aligned} M_1 &: Ta(\vec{u_1}) \longrightarrow Tb(\vec{v_1})|P_1 \\ &\vdots \\ M_n &: Ta(\vec{u_n}) \longrightarrow Tb(\vec{v_n})|P_n \end{aligned}$$

We note $M = \{M_1, \ldots, M_n\}$. We build n PTGs $G_1 \ldots G_n$ such that G_i is composed of all the rules $R \setminus M$ of G and rule M_i $(1 \leq i \leq n)$. Then, we apply recursively this algorithm for computing the solution κ_{G_i} for all the graphs G_i. Hence, we can compute $\kappa_G(Tc, \vec{v})$ with the procedure of Figure 4

i. We use the emission rules to determine which was the previous instance $\vec{u}$ of task Ta that leads to $Tc(\vec{v})$.

ii. Then the function κ_G is:

$$\begin{aligned}\kappa_G(Tc, \vec{v}) = \text{if } \vec{u} \in P_1 \text{ then } &\kappa_{G_1}(Tc, \vec{v}) \\ &\text{else if } \vec{u} \in P_2 \text{ then } \kappa_{G_2}(Tc, \vec{v}) \\ &\qquad \text{else } \ldots \\ &\qquad\quad \ldots \text{if } \vec{u} \in P_{n-1} \text{ then } \kappa_{G_{n-1}}(Tc, \vec{v}) \\ &\qquad\qquad \text{else } \kappa_{G_n}(Tc, \vec{v})\end{aligned}$$

Figure 4: determining $\kappa_G(Tc, \vec{v})$ after spliting G at task Ta

(b) If it exists a node $Ta \in T$ with outdegree n, $n \geq 2$, the n associated rules with Ta have the form:

$$\begin{aligned} M_1 &: Ta(\vec{u_1}) \longrightarrow Tb_1(\vec{v_1})|P_1 \\ &\vdots \\ M_n &: Ta(\vec{u_n}) \longrightarrow Tb_n(\vec{v_n})|P_n \end{aligned}$$

And $Tb_i \neq Tb_j$ for $i \neq j$.

We build n PTGs $G_1 \ldots G_n$ such that G_i has rule M_i but no rule M_j $(j \neq i)$, and no successors of the Tb_j unless they are also successors of Tb_i (see Figure 5 for an example). Compute recursively the solution κ_{G_i} for all the graph G_i. Then, the algorithm to compute κ_G is the same as in case (a) and is given in Figure 4.

4. if the outdegree off all the nodes is less than or equal to one and the only solution of system $\mathcal{S}$ is $\alpha_G = 0$, then it is not possible to find a clustering function which is a linear combination of the elements of the iteration vector by splitting G. Remark that this cannot occur if G

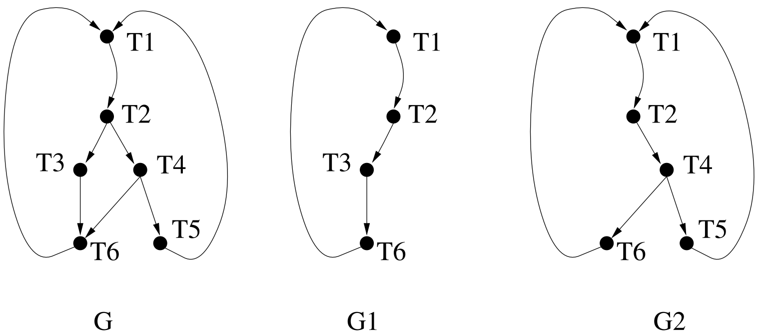

Figure 5: Spliting of Graph G into G_1 and G_2 at node $T2$ (note that G_2 may be splited again at node $T4$)

is a chain. Thus, G is necessary an elementary cycle. Then, we need to find a non linear function that allow to zero all the edges.

For a node Ta with iteration vector $\vec{u}$ ($\vec{u} \in P$), its next iteration vector (after one lap in the cycle) is $\vec{u}'$ and $\vec{u}' = A\vec{u} + \vec{b}$ for some A and $\vec{b}$.

Let f_{Ta} be the following function:

$$\begin{array}{rrcl} f_{Ta}: & P & \rightarrow & P \\ & \vec{u} & \mapsto & A\vec{u} + \vec{b} \end{array}$$

Let $\mathcal{U} \subset P$ be the set of iteration vectors $\vec{u}$ which have no predecessor by f_{Ta}:

$$\vec{u} \in \mathcal{U} \iff \nexists \vec{w} \in P \text{ such that } \vec{u} = f_{Ta}(\vec{w})$$

Hence, for any $\vec{v} \in P$ it exists $\vec{u} \in \mathcal{U}$ such that $\vec{v} = f^n_{Ta}(\vec{u})$, $n \geq 1$. $\vec{u}$ is called the generator of $\vec{v}$, and is denoted by $\text{gen}(\vec{v})$.

We build our clustering function this way: Let $Tb(\vec{z})$ a task ($Tb \in T$). In order to find $\kappa_G(Tb, \vec{z})$, we apply the emission rule of R in order to find the first instance, $\vec{u}''$ of Ta following $Tb(\vec{z})$ in the cycle G (if $Tb = Ta$ then $\vec{u}'' = \vec{z}$). We compute $\vec{u} = \text{gen}(\vec{u}'')$, then $\kappa_G(Tb, \vec{z}) = \sum u_i$ (u_i the ith component of vector u).

5.3 Finding the Starting Task(s)

In order to initiate the computation of the program, we need to know the set of starting tasks (i.e. the set of tasks that have no father).

For any task $Ta(\vec{u})$, using the emission rules, we can compute a polyhedron that describes all the instances of Ta that send data, namely $send(Ta, \vec{u})$. On the other hand, using the reception rules, we can compute a polyhedron that describes all the instances of Ta that receive data, namely $recv(Ta, \vec{u})$. The union of $send(Ta, \vec{u})$ and $recv(Ta, \vec{u})$ describes all the valid instances of Ta namely $valid(Ta, \vec{u})$. The difference $valid(Ta, \vec{u}) - recv(Ta, \vec{u})$ is a polyhedron that describes the set of tasks with no father, i.e. the starting tasks. These tasks can start immediately at the beginning of the program.

For example, concerning task $T1$ in the Gaussian Elimination, its receiving polyhedron is $recv(T1(k+1)) = \{1 \leq k \leq n-2\}$, renaming the variable $k+1$ in k leads to:

$$recv(T1(k)) = \{2 \leq k \leq n-1\}$$

We also have

$$send(T1(k)) = \{1 \leq k \leq n-1\}$$

Thus,

$$valid(T1(k)) = \{1 \leq k \leq n-1\}$$

Computing the difference of the valid and receive polyhedrons is the same as computing the following conjunction:

$$1 \leq k \wedge k \leq n-1 \wedge (k < 2 \vee k > n-1)$$

This can be solved by the *omega library* of the University of Maryland, leading to $\{k = 1\}$. Applying the same algorithm with task $T2$ leads to an empty polyhedron. Hence, T1(1) is the only task that has no father for the Gaussian Elimination.

6 Mapping and Execution of Clusters

All the methods and algorithms described above deal with symbolic parameters and ,hence, can be done at compile time.

Let us assume now that the clustering phase has been done and let us study the execution of the clusters with a parallel machine.

Assume that we have one process per node of the parallel machine and that each process is composed of t threads. The tasks are executed by the threads. Thus, at most t tasks can be executed at the same time on a processor.

Concerning the execution it is possible to reduce some remaining communications using a bloc mapping of each cluster c with bloc of size B. Call p the processor number, hence we have:

$$p = c \div B$$

.

However, in order to get good load balancing properties we will map the clusters cyclically among the P processors:

$$p = c \bmod P$$

We have shown how to build parameterized polyhedrons that described the set of starting tasks. At run time, once the parameter values are given, each processor p scans the set of starting tasks. For each starting task it computes its processor p'. If $p = p'$ then it executes the task.

For each generic task we can build a polynomial that computes its number of fathers. This polynomial is computed from the reception rules, and can be used for any valid instance of the iteration vector and any value of the parameters. We propose the following protocol for task ordering: we maintain a ready queue for the tasks ready to be executed, and a waiting queue for the tasks that have already received data and are waiting for other data.

- When a task receives a data :
  1. If this task is not in the waiting queue then we compute its number of fathers and decrement it. This number is called the *number of remaining fathers*. If its current number of remaining fathers is 0, we put this task in the ready queue, otherwise we put it in the waiting queue.
  2. If this task is in the waiting queue then we decrement its number of remaining fathers and, if this number reaches 0, we put it in the ready queue.
- When a thread is idle, it picks up a task in the ready queue (if not empty).

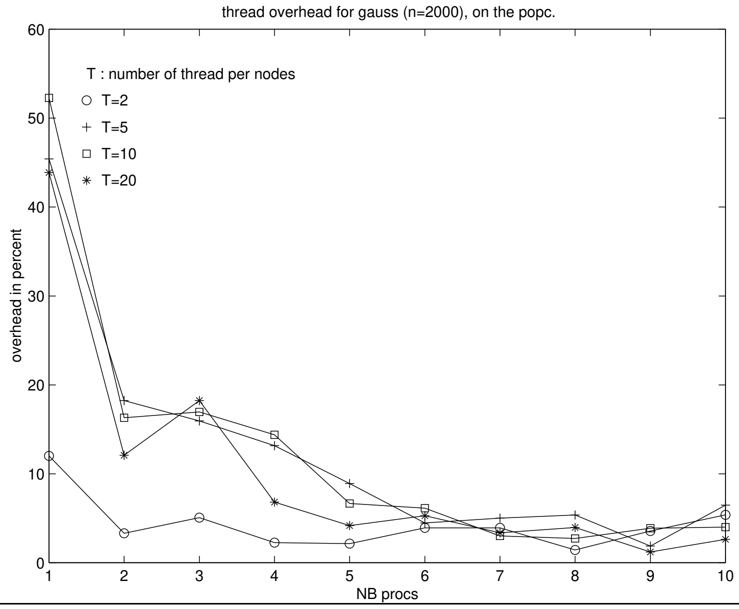

Figure 6: Overhead of the Use of Many Threads on One Node

The execution of the tasks are done asynchronously which, contrary to the data parallel approach, does not involve any synchronization barrier. Therefore, the processors are never forced to be idle by the execution system.

The data to be sent are described by the emission rules. Hence, the message generation is a straightforward implementation of the data part of the emission rules.

7 Results

We have built a multithreaded parallel program that executes the symbolic static allocation founded in the previous section. We have run it on the *popc* machine of LHPC. The popc (pile of PC) is a high performance cluster based on a modular architecture featuring Intel Pentium Pro processors (200 Mhz, 256 KB of cache L2 memory) interconnected by a fast MYRINET network. We use *BIP* as communication protocol. BIP [17](Basic Interface Protocol), allows the network to reach a 10 μs latency and a 100 Mbytes/s bandwidth. The multithreaded API we use is called *PM2* [14]. PM2 (Parallel Multithreaded Machine), allows the user to create threads remotely on a distance node, it can handle a lot of threads on a same node

without high overhead (see figure 6), and finally it can migrate threads easily (which is of no

Size of the matrix	1000	2000	2500
Number of tasks $(n^2/2+3n/2-2)$	501 498	2 002 998	3 128 748
Number of edges (n^2+n-4)	1 000 996	4 001 996	6 252 496
Number of float op $((n^3+n^2-2)/10^6)$	1 001	8 004	15 631
Size in MB for Pyrros	18	78	127

Table 1: Characteristics of the Gaussian Elimination DAG for Various Matrix Size

use so far). We have run our generic program for various size of the program parameters and various number of processors (see Figure 7). We can only use eleven nodes because node 0 is used as the host of the machine. Table 1 gives the characteristics of the DAG with respect to the size of the matrix.

Figure 6 shows the overhead percentage of execution time when using more than one thread per nodes. This overhead is important when using only one processor, because there is no communication, but it decreases with the number of processors. It lowers between 6.5% and 2.5% with ten processors. We believe this is mainly due to the overlapping of communication, and a good scheduling of the threads.

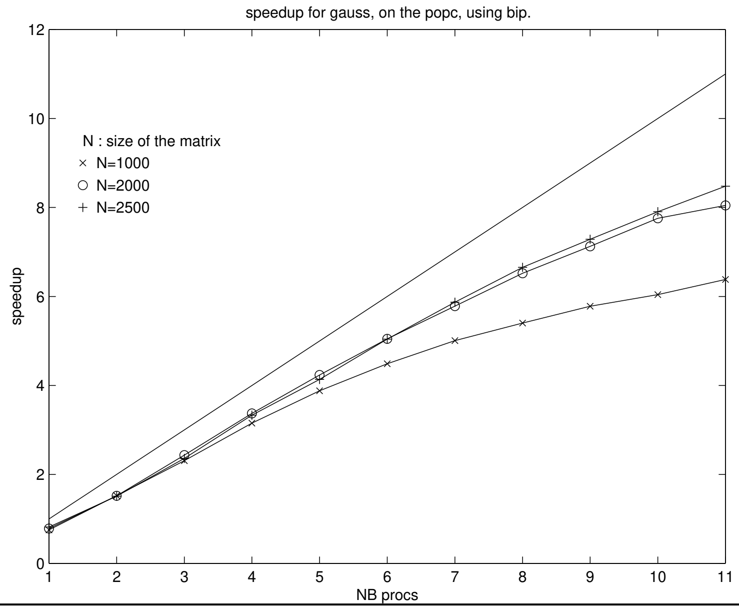

Figure 7: Speedup for Gaussian Elimination for Various Matrix Size

Figure 7 shows the speedup of our program for various size of matrix. We see the high scalability of our method and environment. We have a speedup of 8 for 11 processors when matrix size is 2500.

8 Related Works

It exists tools that schedule and do code generation for static task graphs (CASCH [1] of HKUST, Pyrros [20] of UCSB/Rutgers,...). But those tools require the expanded task graph to be known at compile time.

The problem of symbolically allocating statement of a program as been addressed by many authors. Feautrier in [9] described an algorithm that cut edges of the dataflow graph. This algorithm greatly inspired our work. However, we enhanced it so that conditional can be computed for the allocation.

The SUIF compiler also uses algorithms that symbolically allocate data and computation to the processors [2]. But, this approach compiles programs with *forall* which implies synchronization.

9 Conclusion

In this paper we have presented a method to statically and symbolically schedule a parameterized task graph. This approach is useful when dealing with coarse grain applications and when the expanded task graph is too large to fit into memory.

Our contribution is twofold. First, we have modified Feautrier's algorithm for distributing computations among processors in order to take into account conditionals that arise with the PTG model. Our algorithm is then more general and well suited for our problem. Second, we have described a distributed and asynchronous method to schedule and execute our symbolic allocation using a multithreaded environment.

Our first experiments are very promising and, in our future works, we will realize a code generator that transforms an annotated sequential program into a generic parallel one and executes the symbolic allocation on a distributed memory parallel computer.

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] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[3] T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K.M. Decker, R.M. Rehmann, editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96, Birkhäuser, April 1994.

[4] 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.

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

[6] 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.

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

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

[9] P. Feautrier. Toward Automatic Distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[10] 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.

[11] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988. ACM SIGACT-SIGPLAN.

[12] 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.

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

[14] R. Namyst and J.-F. Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo'95)*, pages 279–285. Elsevier Science Publishers, Sep 1995.

[15] 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.

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

[17] L. Prylli and B. Tourancheau. Bip: a New Protocol Designed for High Performance Networking on Myrinet. In *Workshop PC-NOW IPPS/SPDP'98*, Orlando, USA, April 1998.

[18] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Systems*, 2(4):472–482, 1991.

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

[20] 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.