Compact DAG Representation and its Symbolic Scheduling

Michel Cosnard
LORIA, INRIA Lorraine
France

Emmanuel Jeannot
LORIA
Université Nancy I Henri Poincaré
France

Tao Yang
CS. Dept. UCSB
USA

April 23, 2001

1 Introduction

New parallel computers are potentially able to solve larger and larger problems. In order to execute a program on a parallel computers one approach consists in modeling this program by a directed acyclic task graph (DAG). However, before executing a large problem on a parallel computer it is required to schedule the corresponding task graph. A lot of work has been done concerning scheduling task graphs on parallel machines [12, 19, 20, 23, 24, 30]. But, since the task graph has to be constructed statically and the number of processor of the machine must be given before scheduling, static scheduling algorithms present two major drawbacks:

- they are not *scalable*. The memory requirement of all static schedulers depends on the size of the task graph. Hence, very large task graphs cannot be scheduled and large applications cannot be executed,
- they are not *adaptive*. Each time the parameter values of the application change the task graph change and the schedule has to be recomputed. Moreover, each time the target machine changes the schedule is no longer valid.

In this article we present a scheduling algorithm for symbolically allocating tasks to processors. We use the parameterized task graph (PTG) model, which is a compact and problem size independent model of programs that can be found in scientific applications. A PTG uses parameters that can be instantiated at runtime. In our previous work [5, 6] we have presented a technique for scheduling at runtime the task graph using the PTG model. This scheme have low memory requirement but still depends on the parameter values. We present here an algorithm, called SLC (Symbolic Linear Clustering), that builds a linear clustering from a PTG. Parameters are never instantiated for finding the allocation. Hence, SLC is a scalable algorithm since the memory requirement and the time to compute the solution are constant. Moreover SLC is adaptive because this allocation is valid for all parameter values and any processor number.

The rest of this article is organized as follow: Section 2 gives the definitions used in this paper. Section 3 explains how to derive a parameterized task graph from a sequential program. Section 4 presents previous works on dynamic scheduling of parameterize task graphs. Section 5 is about our motivations and present an overview of SLC. Why rule merging is important and how we perform it is in Section 6. In section 7 we propose an algorithm for extracting bijection rules. How we explicitly build the linear clustering is provided section 8. Section 9 discusses how clusters can be numbered explicitly so that symbolic clusters can be evenly mapped to physical processors

Section 10 we discuss the issue of finding the starting and ending task of a program. How clusters are mapped to real processors is show section 11. In section 12 our results are presented. Section 13 presents related work on symbolic computation allocation. We propose an open problem section 14. Section 15 concludes the article.

2 Definitions

In this article we will use the following 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. In this paper we will say node or task indifferently. E is the edge set. There is an edge from task i to task j if there is a dependence between task i and task j (i.e. task j must be executed after the end of task i). T is the task weight (or task duration) set, $\tau_i \in T$ will represent task i duration. C is the set of edge weights (or communication volumes), $c_{i,j} \in C$ represents the communication cost along the edge from node i to node j. It becomes 0 if the two tasks are mapped on the same processor.

- We call $g(G)$ the *granularity* of the task graph G. We use the definition given by Gerasoulis and Yang in [17]:

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

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

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

- A cluster is called *linear* if all its tasks form a path in the task graph. A cluster is called *non-linear* if two independent tasks exist in this cluster.

- A *clustering* is a mapping of all the tasks of G into clusters. The clustering is called linear if all the clusters are linear otherwise it is called non-linear. Hence, a linear clustering is a set of disjoint paths in the task graph. Clustering algorithms have been first proposed by Sarkar in [26]. Gerasoulis and Yang have discussed the impact of granularity for clustering algorithms in [17].

- A *parameterized task graph* (PTG) is a compact and symbolic representation of some task graphs that can be found in scientific applications. It has first been proposed by Cosnard and Loi in [9] for automatically building task graphs. It uses parameters that can be instantiated for building the corresponding task graph.

  A PTG is defined by the tuple $\mathcal{G} = (\mathcal{T}, \mathcal{R}, \mathcal{C})$ where:

  - $\mathcal{T}$ is the set of generic tasks. A generic task is a set of instructions that have to be executed in sequential. A generic task is defined by its iteration vector (enclosing loop indices taken in lexicographic order) its first and its last instruction.
  - $\mathcal{R}$ is the set of communication rules. Each communication rule describes, in a symbolic manner, dependencies between generic tasks. More details on communication rules will be given later on.

- $\mathcal{C}$ is the cost function of each task. It computes the number of operations a task as to perform before completion once given the program parameter values and the iteration vector value.

- *Communication rules* are symbolic descriptions of task graph edges. There are two types of communication rules. The first type is emission rule that describes data sent by tasks. The second type is reception rule that describes input data of tasks. We can automatically transform a reception rule into an emission rule. Hence, we give only the formal definition of an emission rule. Let:

  - T_a and T_b two generic tasks with iteration vector $\vec{u}$ and $\vec{v}$.
  - D a data exchanged between T_a and T_b and $\vec{y}$ vector of same dimension than D.
  - P a parameterized polyhedron.

  An emission rule as the following form:

  $$T_a(\vec{u}) \rightarrow T_b(\vec{v}) : D(\vec{y})|P$$

  This rule is read: " for all $\vec{u}$ $\vec{v}$ and $\vec{y}$ in polyhedron P task $T_a(\vec{u})$ send data $D(\vec{y})$ to task $T_b(\vec{v})$.

- An *instantiated task graph* is the task graph corresponding to a given instance of a parameterized task graph.

3 From a Sequential Program to a Parameterized Task Graph

3.1 PlusPyr

PlusPyr is a tool developed by Loi [21] that build a PTG from an annotated sequential program. However, the input program model must have some limitations in order information about the program to be computed automatically. The program must have static control [13]:

- the program must be written in an FORTRAN-like imperative language,
- only `for` loops and `if-then-else` control structures are allowed,
- loop indices bound and array access functions must be integer-valued affine functions of enclosing loop indices and program parameters,
- `if-then-else` conditions must be comparison of integer-valued affine functions of enclosing loop indices and program parameters,
- the input language is augmented with two task delimiters `task` and `endtask`. These delimiters are used to grouped instructions into tasks.
- task cannot be nested,
- all assignments must be included into a task,
- loop indices and program parameters cannot be used as the left hand-side of an assignment.

```
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)
  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)
   endfor
  endtask
 endfor
endfor
```

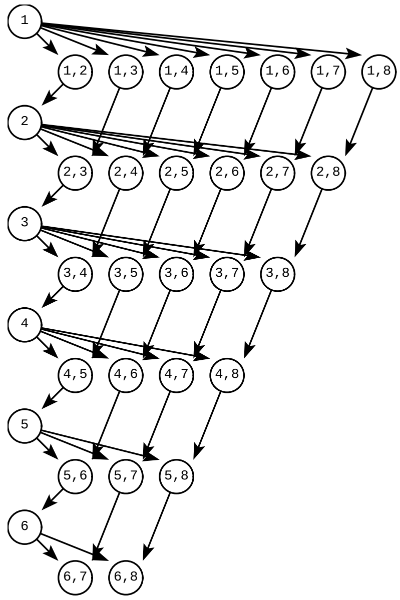


Figure 1: Gaussian Elimination and Instanciated Task Graph (n=7)

3.2 Example

Figure 1 is a straightforward implementation of the Gaussian Elimination algorithm. It uses one parameter: `n` and is composed of two generic tasks T_1 and T_2. When `n` $= 7$, the instantiated task graph is shown. Iteration vector of T_1 is (k) and iteration vector of T_2 is (k,j). The cost function of each task computed by PlusPyr is: $\mathcal{C}(T_1(k)) = 2(n-k-1)$ and $\mathcal{C}(T_2(k,j)) = 3(n-k-1)$. The emission rules set is shown Figure 2. Rule R_1 is then read as follows: "Task $T_1(k)$ sends to task

$$R_1 \ : T_1(k) \to T_2(k,j) : A(i,k) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$$

$$R_2 \ : T_2(k,j) \to T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+2 \leq i \leq n$$

$$R_3 \ : T_2(k,j) \to T_1(k+1) : A(k+1,k+1) | 1 \leq k \leq n-2, j = k+1$$

$$R_4 \ : T_2(k,j) \to T_2(k+1,j) : A(i,j) | 1 \leq k \leq n-2, k+1 \leq j \leq n+1, k+2 \leq i \leq n$$

$$R_5 \ : T_2(k,j) \to T_2(k+1,j) : A(k+1,j) | 1 \leq k \leq n-2, k+1 \leq j \leq n+1$$

Figure 2: Emission rules for the Gaussian Elimination

$T_2(k,j)$ a part of column k of matrix A, where $1 \leq k \leq n-1$, $k+1 \leq j \leq n+1$ and $k+1 \leq i \leq n$."

4 Dynamic Scheduling of Parameterized Task Graphs

The compactness of the PTG model make it very useful for scheduling large task graphs. In our previous work [5, 18], we have presented an algorithm called `PTGDS` (Parameterized task graph dynamic scheduler) for scheduling and executing parameterized task graphs at runtime. Compared to static scheduling algorithm this approach presents two major advantages:

1. it allows to schedule very large task graphs. Indeed, the task graph is never completely instantiated in memory. Hence, the memory complexity of the algorithm is very low,

2. the schedule can be computed and executed at runtime. This allows to build a generic program such that parameter values are instantiated just before execution.

In [27] Wu et al. present an algorithm for scheduling a parameterized task graph (called Compact DAG in the article) at runtime. As for PTGDS the task graph is instantiated incrementally, scheduled and executed on a parallel machine. The scheduling algorithm is parallelized to scale to large systems. This system seems to be able to achieve better performances than PTGDS. However this approach has the following deficiencies: (1) The algorithm schedule single-assigned programs. Hence the memory cost is greatly increased which limit its practical use. (2) The parameterized task graph is build from a single assigned program. Since there is no application written in a single-assigned manner, it is required to manually transform an application into a single assigned program. Hence the method is far to be fully automatic.

5 Symbolic Scheduling of Parameterized Task Graphs

5.1 Motivations

Dynamic scheduling allows to execute large task graphs and to build generic programs. However the time needed to schedule a task graph depends on the parameter values (the larger the parameter values, the larger the task graph and the longer the computation of the schedule). Moreover, the schedule has to be recomputed each time the parameter values change. Indeed, when the parameter values change the topology of the task graph and the valuation of the edges and nodes change.

We present here a new algorithm called SLC (Symbolic Linear Clustering), which finds, given a PTG, an allocation function for each generic task. This function depends only on the parameters of the program, the iteration vector of the generic task and the number of available porcessors for execution. Therefore our algorithms present the three following advantages upon all previous approaches:

1. the allocation found by SLC is a program parameter function. The allocation problem is virtually solved for all parameter values,

2. the function computed by SLC depends only of the parameterized task graph. Hence the time required to compute this function does not depend on the parameter values,

3. to evaluate the function it is only required to know the iteration vector value, the parameter values and the target machine. Hence this evaluation can be done in constant time and in a decentralized manner.

5.2 Overview of SLC

The goal of our algorithm is to built a linear clustering of any instantiated task graphs that can be built from a given parameterized task graph. In [17] Gerasoulis and Yang have proven that a linear clustering is within a factor of $1 + 1/g$ (if g is the granularity if the task graph) from the optimal schedule. Moreover in the same article the authors have shown that any non linear clustering can be transform into a linear clustering that has a parallel execution time lower than the parallel execution time of the non linear clustering. This proves that for any coarse grained graph it exist a linear clustering which is optimal. The good properties of a linear clustering comes from that, since it is composed of disjoint paths of the task graph, it suppresses unnecessary communications while preserving parallelism.

SLC is decomposed in four steps:

1. *merging and optimizing rules.* We need to merge rules in order to reduce their number. Indeed, the SLC execution time depends on the number of rules: the less rules the faster the algorithm. Moreover, the size and the speed of code generated from the PTG depend on the number of rule. Hence it is required to have as fewer rules as possible to generate a good parallel program,

2. *extracting bijection rules.* We analyze the communication rules in order to extract *bijection rules.* Bijection rules described point to point communications and are those that will be part of the linear clustering,

3. *building a linear clustering.* Given a set of bijection rules this step consists in selecting some of them in order to guaranty that, for all parameter values, the selected rules will always form a linear clustering,

4. *computing the symbolic allocation.* Once the linear clustering is build, we need to construct a function that, once parameter values are known, computes the cluster number of a given task according to the linear clustering found in the previous phase.

6 Merging and optimizing rules

It sometime appears that two rules describe the same set of edges and the data transmitted are contiguous. In that case it is possible to merge the two rules in one where the transmitted data is the union of the data transmitted by the two rules. Our rule merging algorithm is depicted Figure 4. This algorithm is able to merge rules R_2 and R_3 of the Gaussian Elimination rules set Figure 2 as well as rule R_4 and R_5. The three remaining communication rules are shown Figure 3.

$$R_1 : T_1(k) \rightarrow T_2(k,j) : A(i,k) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$$

$$R_2 : T_2(k,j) \rightarrow T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

$$R_3 : T_2(k,j) \rightarrow T_2(k+1,j) : A(i,j) | 1 \leq k \leq n-2, k+1 \leq j \leq n+1, k+1 \leq i \leq n$$

Figure 3: Merged emmission rules for the Gaussian Elimination

In algorithm described Figure 4, the `same_edges` function checks if rule R_1 and rule R_2 describe exactly the same set of communicating edges. The `adjacent` function checks if two domains are contiguous and if the union forms a convex polyhedron. So far, this function is simply implemented: it does some syntax analysis of the rules and is not able to find all the cases. However, this implementation is conservative (semantic of the PTG is preserved) and has been found effective for all cases encountered until now.

Entry: Two communication rules R_1 and R_2.
Output: One rule R, merging of R_1 and R_2 if possible.
`Merge`(R_1,R_2)
  if `same_edges`(R_1,R_2) **then**
    if transmitted data (D) is the same for R_1 and R_2 **then**
      S_1 = domain of D that is transmitted by R_1;
      S_2 = domain of D that is transmitted by R_2;
      if `adjacent`(S_1,S_2) **then**
        $R = R_1$ where D is transmitted on the $S_1 \cup S_2$ domain;

Figure 4: *Merging rules $R1$ and $R2$ describing the same set of edges*

7 Extracting bijection rules

A bijection rule is a communication rule that only described point to point communications. A bijection rule does not describe global communication such as broadcast or gather. We need to extract bijection rules because, as shown by theorem 2 only these rules can be part of a linear clustering.

Let R be an emission rule[1]:

$$R : T_a(\vec{u}) \to T_b(\vec{v})|P$$

R is a bijection rule if for any valid instance of $\vec{u}$ it exists one and only one instance of $\vec{v}$. Our method for finding if a given rule is a bijection rule or not is the following. Let $\vec{\imath}$ the vector made of the program parameters and the loop indices of the input program taken in lexicographic order. Therefore it exists two matrices D_1 and D_2 such that:

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

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

where $\vec{k_1}$ and $\vec{k_2}$ are two constant integer-valued vectors.

First we change equalities that appear for the variables in P by doing variable substitutions. Second matrices D_1 and D_2 are build. Then we find matrix E_1 and vector w_1 such that:

$$\vec{\imath} = E_1(\vec{u} - \vec{k_1}) + \vec{w_1}$$

It exists many E_1 and w_1 that verifies the previous equation. We impose that $\vec{w_1}$ is composed of variables that does not appear in $\vec{u}$. Formally this is written $w_{1j} = \delta_j i_j$ where w_{1j} (resp. i_j) is the j$^{\text{th}}$ element of $\vec{w_1}$ (resp. $\vec{\imath}$) ; $\delta_j = 0$ if i_j appears in $\vec{u}$ and $\delta_j = 1$ otherwise. This restriction guarantees that E_1 and w_1 are unique.

Therefore, we can express $\vec{v}$ as

$$\vec{v} = D_2E_1(\vec{u} - \vec{k_1}) + D_2\vec{w_1} + \vec{k_2} \tag{1}$$

[1] In the following we do not consider the communication part of the rule, because this does not interfere with the fact that a rule is a bijection rule or not.

Similarly we can build E_2 and w_2 such that:

$$\vec{u} = D_1 E_2(\vec{v} - \vec{k_2}) + D_1 \vec{w_2} + \vec{k_1} \tag{2}$$

We have the following theorem:

Theorem 1 *R is a bijection rules if and only if $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant (we consider program parameters as constant).*

Proof of theorem 1: Equation 1 (resp. equation 2) expresses $\vec{v}$ as a function of $\vec{u}$ (resp. $\vec{u}$ as a function of $\vec{v}$). Hence if $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant for each valid instance of $\vec{u}$ there is exactly one instance of $\vec{v}$ and vice a versa: the rule is a bijection rule.

Let suppose that R is a bijection rule. Hence for any valid instance of $\vec{u}$ there is only one valid instance of $\vec{v}$. After variable substitutions due to equalities all the variables of $\vec{v}$ must appear in $\vec{u}$ (if it is not the case then for different values of the $\vec{v}$'s variables that does not appear in $\vec{u}$ we have only one instance of $\vec{u}$ hence the rule is not a bijection rule). By definition $\vec{w_1}$ is composed of variables and parameters that does not appear in $\vec{u}$. Therefore $\vec{w_1}$ is composed of variables that does not appear in $\vec{v}$. Hence $D_2\vec{w_1}$ is constant. A symmetric reasoning leads to $D_1\vec{w_2}$ is constant too.

∎

Example Let consider two emission rules of the Gaussian Elimination of Figure 3

$$R_1 : T_1(k) \rightarrow T2(k,j) : A(i,k) | 1 \leq k \leq n-1, k+1 \leq j \leq n+1, k+1 \leq i \leq n$$

$$R_2 : T_2(k,j) \rightarrow T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

Rule R_1 is not a bijection rule. Indeed, $\vec{u} = (k)$ and $\vec{v} = \begin{pmatrix} k \\ j \end{pmatrix}$. The parameters and loop indices vector is: $\vec{\imath} = \begin{pmatrix} n \\ k \\ l \\ j \\ i \end{pmatrix}$. Hence, $D_1 = (0,1,0,0,0)$ and $D_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$. We find that $E_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ and $\vec{w_1} = \begin{pmatrix} n \\ 0 \\ l \\ j \\ i \end{pmatrix}$ such that $\vec{\imath} = E_1\vec{u} + \vec{w_1}$. Hence, $D_2\vec{w_1} = \begin{pmatrix} 0 \\ j \end{pmatrix}$ is not constant.

However, rule R_2 is a bijection rule. Indeed, $\vec{u} = \begin{pmatrix} k \\ j \end{pmatrix}$ and $\vec{v} = (k+1)$. Since $j = k+1$ we substitute j by $k+1$. We obtain: $\vec{u} = \begin{pmatrix} k \\ k+1 \end{pmatrix}$. Moreover $D_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ and $D_2 = (0,1,0,0,0)$. We find that: $\vec{w_1} = \vec{w_2} = \begin{pmatrix} n \\ 0 \\ l \\ j \\ i \end{pmatrix}$. Hence $D_2\vec{w_1}$ et $D_1\vec{w_2}$ are both constant.

8 Building a linear clustering

Our algorithm for building a linear clustering is based on the rule zeroing algorithm depicted Figure 5. Zeroing a rule consists in putting each sending task and receiving task of each instance of the rule onto the same cluster (if two communicating task are put onto the same cluster then the communication cost between these two tasks becomes *zero*). Formally zeroing a rule is defined as follows. Let κ the mapping function of tasks: $\kappa(T_a(\vec{u})$ is the cluster number of task $T_a(\vec{u})$. Let $R : T_a(\vec{u}) \to T_b(\vec{v}) : D(\vec{y})|P$ a rule. R is zeroed if for each instance of $\vec{u}$ and $\vec{v}$ $\kappa(T_a(\vec{u})) = \kappa(T_b(\vec{v}))$. As shown by theorem 2, our algorithm selects a set of rules (called the $\mathcal{Z}$-set such that once zeroed they describe a linear clustering.

Entry: A PTG=$(\mathcal{T},\mathcal{R},\mathcal{C})$.
Output: a set $\mathcal{Z}$ of selected rules.
1 `merge_rule`$(\mathcal{R})$;
2 $\mathcal{B}$ = `extract_bijection_rules`$(\mathcal{R})$;
3 $\mathcal{Z}$ =`maximal_non_conflicting_rule_set`$(\mathcal{B})$

Figure 5: *Rule Selection Algorithm*

We have already described the merging rule step as well as the bijection rule extraction step. The last step consists in determining if two rules are in conflict and then extracting a maximal set of rules that are not in conflict.

8.1 Determining Conflicting Rules

Two rules are in conflict if once zeroed the produced clustering is not linear. Let R_1 and R_2 two bijection rules.

$$R_1 : T_a(\vec{u}) \to T_b(\vec{v})|P_1$$

and

$$R_2 : T_c(\vec{w}) \to T_d(\vec{z})|P_2$$

In order to know if R_1 and R_2 are in concflict we consider two cases

- We can have a *fork* conflict, if for a valid instance of R_1 and R_2, the same sending task send data to two different receiving tasks. Formally:
    1. $T_a = T_c$.
    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 $\vec{w_1}$, $\vec{w_1} \in P_2$ succ that $\vec{u_1} = \vec{w_1}$.
    3. Let $\vec{v_1}$ the image[2] of $\vec{u_1}$ according to ruleR_1 and $\vec{z_1}$ the image of $\vec{w_1}$ according to rule R_2 such that $T_b(\vec{v_1}) \neq T_d(\vec{z_1})$.
- We can have a *join* conflict, if for a valid instance of R_1 and R_2 two different sending tasks send data to the same receiving task. Formally:

[2]This image exists and is unique because R_1 is a bijection rule.

1. $T_b = T_d$
2. It exists an instance of $\vec{v}$ called $\vec{v_1}$, $\vec{v_1} \in P_1$ and an instance of $\vec{z}$ called $\vec{z_1}$, $\vec{z_1} \in P_2$ such that $\vec{v_1} = \vec{z_1}$.
3. Let $\vec{u_1}$ the source of $\vec{v_1}$ according to rule R_1 and $\vec{w_1}$ the source of $\vec{z_1}$ according to rule R_2 such that $T_a(\vec{u_1}) = T_c(\vec{w_1})$.

In order to determine automatically if two rules are in join conflict (the fork conflict case is symmetric) we proceed as follow. Concerning step 2 we compute the set I which is the intersection between $P_1 \backslash \vec{u}$ (P_1 restraint to $\vec{u}$) and $P_2 \backslash \vec{w}$. If I is empty there cannot be any fork conflict. Otherwise if $T_b = T_d$ and $R_1 \backslash I$ (R_1 restricted to the I set) is different from $R_2 \backslash I$ there is a fork conflict. I $R_1 \backslash I$ and $R_2 \backslash I$ can be symbolically computed with the omega calculator from the university of Maryland [25].

Example Let consider the two following rules:

$$R_1 : T_1(k,j) \to T_2(j) | k = 1, 1 \leq j \leq n$$

and

$$R_2 : T_1(k,j+1) \to T_2(j) | k = 1, n \leq j \leq 2n$$

R_1 and R_2 are in *join* conflict for $j = n$ (see Figure 6). On the contrary the following rules:

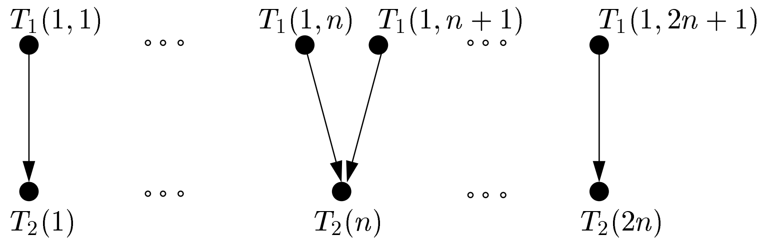


Figure 6: Example of join conflict: for $T_2(n)$ has two predecessors

$$R_1 : T_1(k,j) \to T_2(j) | k = 1, 1 \leq j \leq n$$

and

$$R_2 : T_1(k,j) \to T_2(j) | k = 1, n \leq j \leq 2n$$

are not in conflict, because for $j = n$, both rules describe the same edge. Let us come back to our Gaussian Elimination example. We have two bijection rules:

$$R_1 : T_2(k,j) \to T_1(k+1) : A(i,k+1) | 1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$$

and

$$R_2 : T_2(k,j) \to T_2(k+1,j) : A(i,j) | 1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n$$

R_1 and R_2 can be in fork conflict. However R_1 is valid only when $j = k+1$ whereas R_2 is valid for $k+2 \leq j \leq n+1$. Therefore no instance of $T_2(k,j)$ has two sons. R_1 and R_2 are not in conflict and can be both zeroed for describing a linear clustering.

8.2 Selecting Bijection Rules

Given a set of bijection rules, we need to select the $\mathcal{Z}$-set, a subset of non-conflicting bijection rules that once zeroed will lead to a linear clustering. Since when a rule is zeroed the corresponding communications are suppress we need to zero rules that imply large are communication. We propose here two complementary approaches.

Greedy selection In our previous work [1, 7, 8] we have proposed a greedy method that is composed of two steps:

1. sorting the rules such that most communicating rules will be consider first,
2. initially the zeroed rule set is empty. This set is augmented with a new rule if this rule is not in conflict with previous selected rules.

This method is fast and can be used for very large parameterized task graph. However, since it is a greedy algorithm, it may lead to suboptimal rule selection. Hence, we propose a new method based on the topology of the parameterized task graph.

Maximal independent set search Given a graph $G = (V, E)$ an *independent set* is a subset of the nodes $V' \subset V$ such that there is no edge between any node of V'.

For selecting bijection rules we build the *conflict graph*. Each node is a bijection rules and there is an edge between two nodes if the two corresponding rules are in conflict. Finding an independent set of this graph leads to build a subset of bijection rules that are not in conflict.

Moreover, we want to zero rules that imply large communication and we want to zero as much rule as possible. Zeroing as much rule as possible can be done in selecting the maximal independent set of the conflict graph. Zeroing large communication is done by labeling the nodes with weights corresponding to the dimension of the transmitted data. If the transmitted data is a matrix we label the node with 10000 if the transmitted data is a vector we label the node with 100 and if the transmitted data is a scalar we label the node with 1. Hence we find an independent set that maximizes the sum of the weight of the selected nodes. This method has the advantage to be more general than the greedy method appears presented above and is now the default method of our algorithm. However finding a maximum independent set is an NP-complete problem [16]. Hence our program checks all possibilities. We have done some optimizations such as decomposing the graph in cliques: at most one element of each clique is in the independent set. It appears that even for very large PTGs this program is very fast (finding an independent set for a PTG with 40 rules takes a few seconds on a Pentium III 500 Mhz). Therefore the maximum independent set search method is used for PTGs with less than 40 rules, for PTGs with more than 40 rules we use the greedy method described above.

8.3 Proof of the zeroing algorithm

We need to prove that our method is correct: the selected rules once zeroed form a linear clustering.

Theorem 2 *Let $\mathcal{G}$ a PTG. For all instantiated task graphs G of $\mathcal{G}$, zeroing all the communications described by the $\mathcal{Z}$-set leads to a linear clustering.*

Proof of theorem 2 Each rule in $\mathcal{Z}$ is a bijection rule. Zeroing a bijection rules consists in grouping tasks, in the instantiated task graph, along paths (linear cluster).

Zeroing all rules in the $\mathcal{Z}$ set leads, in the instantiated task graph, to merge some linear clusters. Since no rule in $\mathcal{Z}$ are in conflict there will be no independent task in the resulting clusters.

9 Computing the symbolic allocation

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

9.1 General Case

Let us consider the following rule

$$R : T_a(\vec{u}) \to T_b(\vec{v})|P$$

If this rule is zeroed then:

$$\kappa(T_a, \vec{u}) = \kappa(T_b, \vec{v}) \tag{3}$$

Our method is based on Feautrier algorithm [14]. As Feautrier we assume that our allocation function is affine with respect to the program parameters and iteration vector. We choose the affine form because it is general and leads to a system of equations we always know how to solve when a solution exists.

Therefore we must have:

$$\kappa(T_a, \vec{u}) = \vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p}$$

where $\vec{p}$ is the program parameter vector, $\vec{\alpha_a}$ and $\vec{\gamma_a}$ are unknown vectors to be determined and β_a is constant scalar to be determined.

We also have $\kappa(T_b, \vec{v}) = \vec{\alpha_b}\vec{u} + \vec{\beta_b} + \vec{\gamma_b}\vec{p}$. Therefore solving equation 3 is the same as solving

$$\vec{\alpha_a}\vec{u} + \vec{\beta_a} + \vec{\gamma_a}\vec{p} = \vec{\alpha_b}\vec{v} + \vec{\beta_b} + \vec{\gamma_b}\vec{p} \tag{4}$$

We repeat this process for all rule in $\mathcal{Z}$ and we obtain a system S such that

$$S : C\Omega = 0$$

where C is obtained by algebraic manipulation of equations like 4 and Ω is a vector composed of the $\vec{\alpha}$'s the β's and the $\vec{\gamma}$'s. For solving this system we apply the Gaussian elimination algorithm on integers for matrix C. Then, all non eliminated variables are set to 1. Finally we do a triangular solve of the system. If we find $\vec{\alpha} \neq \vec{0}$, then a last step consists in subtracting to all the β's and to all γ_i's the smallest absolute value of these figures. For example if $\beta_a = -1$ and $\beta_b = 1$ then the simplified solution is $\beta_a = 0$ and $\beta_b = 2$. If $\gamma_{a,1} = \gamma_{b,1} = 1$ then the simplified solution is $\gamma_{a,1} = \gamma_{b,1} = 0$. We will consider the case where the unique solution of the system is $\vec{\alpha} = \vec{0}$ section 9.3.

9.2 Example

Let us come back to our Gaussian Elimination example rule R_2 is:

$T_2(k, j) \to T_1(k+1) : A(i, k+1)|1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$

Hence we have:

$\kappa(T_2, (k, j)) = \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n$ and $\kappa(T_1, (k+1)) = \alpha_{1,1}(k+1) + \beta_1 + \gamma_{1,1}n$

This leads to:

$$\begin{cases} \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n = \alpha_{1,1}(k+1) + \beta_1 + \gamma_{1,1}n \\ j = k+1 \end{cases} \tag{5}$$

Equation 5 is satisfied for: $\alpha_{2,1} + \alpha_{2,2} = \alpha_{1,1}$, $\gamma_{1,1} = \gamma_{2,1}$ and $\beta_2 + \alpha_{2,2} = \beta_1 + \alpha_{1,1}$

Concerning rule R_3 ($T_2(k,j) \rightarrow T_2(k+1,j) : \{A(i,j)|1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n\}$) we have:

$$\alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n = \alpha_{2,1}(k+1) + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n \tag{6}$$

Equation 6 is satisfied for $\alpha_{2,1} = 0$.

Hence, we have the following system:

$$\begin{cases} \alpha_{2,1} + \alpha_{2,2} - \alpha_{1,1} = 0 \\ \beta_2 + \alpha_{2,2} - \beta_1 - \alpha_{1,1} = 0 \\ \gamma_{1,1} = \gamma_{2,1} \\ \alpha_{2,1} = 0 \end{cases} \tag{7}$$

System 7 has an infinite number of solution, in particular:

$$\begin{cases} \alpha_{2,2} = \alpha_{1,1} = 1 \\ \beta_1 = \beta_2 = \alpha_{2,1} = \gamma_{1,1} = \gamma_{2,1} = 0 \end{cases} \tag{8}$$

Hence a possible κ function for the Gaussian elimination is:

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

and

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

Figure 7 shows in shaded boxes the clustering for the Gaussian Elimination when n is 6.

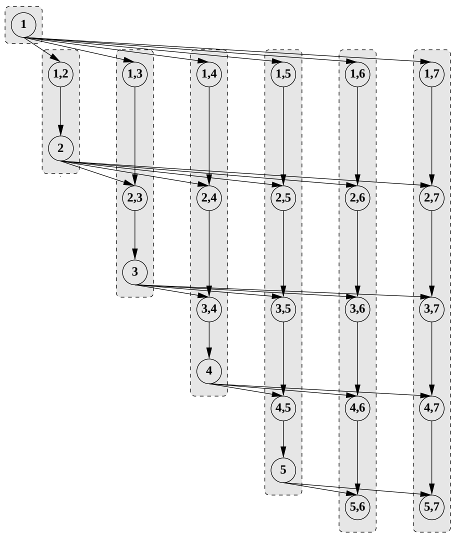


Figure 7: A clustering for an instantiated Gaussian Elimination graph with $n = 6$.

9.3 Case $\vec{\alpha} = \vec{0}$

It may happen that two rules may lead to contradictory conditions on the α's. For instance, let us consider the two following rules:

$$R_1 : T_1(k,j) \rightarrow T_1(k+1,j) | 1 \leq k \leq n, 1 \leq j \leq k$$

and

$$R_2 : T_1(k,j) \rightarrow T_1(k,j+1) | 1 \leq k \leq n, k+1 \leq j \leq n$$

Rule R_1 leads to $\alpha_{1,1} = 0$ and rule R_2 leads to $\alpha_{1,2} = 0$. We do not want a solution where $\vec{\alpha} = \vec{0}$ because no parallelism is expressed. However, we know that these two rules are not in conflict. Two functions can then be applied to two disjoint polyhedrons:

$$\kappa(T1,(k,j)) = \begin{cases} j & \text{if } 1 \leq k \leq n, 1 \leq j \leq k \\ k & \text{otherwise} \end{cases}$$

In the general case we proceed has follow. We build a graph G from the PTG where each vertex is a generic task and each edge is a bijection rule taken from the $\mathcal{Z}$ set.

When all the $\vec{\alpha}$'s are zeroed the obtained allocation function has few or no parallelism: we reject it. In such a case, we split the graph G in subgraphs G_1, $G_2, \ldots$ G_i, then we apply our identification method for the subgraph. The final solution for G is no longer an affine function but a *piecewise affine* function. During execution, in order to determine which function is to be applied for a given task $T_b(\vec{v})$, we go from rule to rule into the graph up to generic task T_a where the graph has been splitted. Then we determine the instance of $T_a(\vec{u})$ that leads to $T_b(\vec{v})$ (if $T_a = T_b$ then $\vec{u} = \vec{v}$). Finally the value of $\vec{u}$ determine the sub-graph and the identification function is to be applied.

When a PTG has to be splitted we first suppress multiple edges as shown in Figure 8.

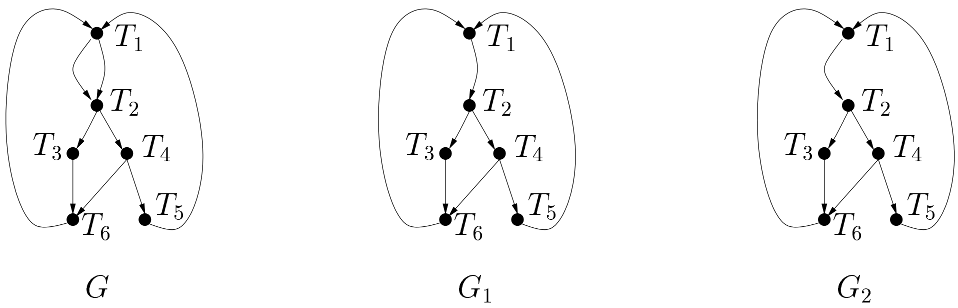


Figure 8: PTG G is splitted in two subgraphs G_1 and G_2 such that multiple-edge (T_1,T_2) is suppressed

If there are no multiple edges we split the PTG at a node without degree greater than 1 as in Figure 9.

In the worst case it may happen that we obtain a graph that cannot be splitted and such that all the α's are zero. We propose three solutions for handling this case:

- if G is composed of only one rule and one generic task, we accept the solution with $\alpha = 0$. Indeed that means that G describes a path in all instantiated graphs hence the mapping function must not depends on the iteration vector.

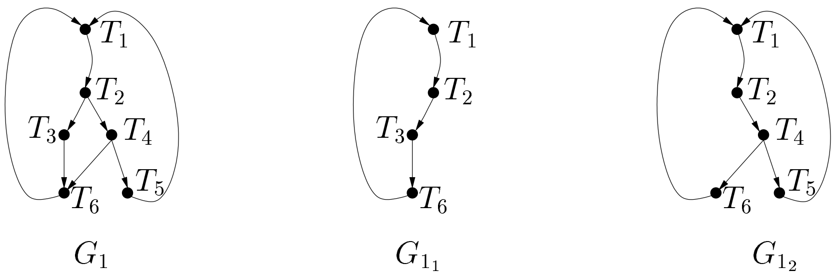


Figure 9: PTG G_1 is splitted in two subgraphs G_{1_1} and G_{1_2} at node T_2 (note that G_{1_2} can be splitted at node T_4)

- We propose a runtime solution that consists in mapping the first task in a cluster which number is the value of the first component of its iteration vector. Then, we execute all the tasks of this sub-graph on this processor.
- Finally we propose a compile time solution that consists in suppressing rules from the $\mathcal{Z}$-set and to recompute a solution until a non-trivial solution is found.

The second solution allows to execute the linear clustering but requires computations during execution. The third solution is simpler but the found solution implies more communication since fewer rules are zeroed.

10 Finding starting tasks and ending tasks

In order to execute the found allocation it is required to know the set of starting tasks (tasks that have no predecessor) as well as the set of ending tasks (tasks that have no successor).

Let T_a a generic task. We compute the polyhedron $P_{\text{start}}(T_a)$ that describes iteration vector values for which T_a has no predecessor and $P_{\text{end}}(T_a)$ the polyhedron that describes iteration vector values for which T_a has no successor. Once we have computed these two sets for all generic tasks of a PTG we know the starting and ending tasks.

In order to compute $P_{\text{start}}(T_a)$ we shall proceed as follow. Let $\text{emit}(T_a)$ the set of T_a instances that send data. It is the union of all the values of $\vec{u}$ for all emission rules such that T_a is the emission task ($T_a(\vec{u}) \rightarrow \ldots$) and $\text{recep}(T_a)$ the set of T_a instances that receive data. In that case it is the union of all values of $\vec{u}$ for all reception rules such that T_a is the receiving task ($T_a(\vec{u}) \leftarrow \ldots$). Let $\text{valid}(T_a)$ the set of all valid instances of T_a: it is the union of $\text{emit}(T_a)$ with $\text{recep}(T_a)$. Therefore $P_{\text{start}}(T_a) = \text{valid}(T_a) - \text{recep}(T_a)$ is the set of valid instances of T_a that do not send data and thus has no successor. $P_{\text{end}}(T_a) = \text{valid}(T_a) - \text{emit}(T_a)$ is the set of valid instance of T_a that do not receive data and thus has no predecessor.

Let us consider task T_1 of the Gaussian Elimination:

$$\text{emit}(T_1) = \{k | 1 \leq k \leq n-1\}$$

and

$$\text{recep}(T_1) = \{k | 2 \leq k \leq n-1\}$$

Therefore

$$\text{valid}(T_1) = \{k | 1 \leq k \leq n-1\}$$

hence

$$P_{\text{start}}(T_1) = \{k|1 \le k \le n-1\} - \{k|2 \le k \le n-1\} = \{1\}$$

et

$$P_{\text{end}}(T_1) = \{k|1 \le k \le n-1\} - \{k|1 \le k \le n-1\} = \emptyset$$

For task T_2 of the Gaussian Elimination we have:

$$\text{emit}(T_2) = \{(k,j)|1 \le k \le n-2, k+1 \le j \le n+1\}$$

and

$$\text{recep}(T_2) = \{(k,j)|1 \le k \le n-1, k+1 \le j \le n+1\}$$

therefore

$$\text{valid}(T_1) = \{(k,j)|1 \le k \le n-1, k+1 \le j \le n+1\}$$

hence

$$\begin{aligned} P_{\text{start}}(T_2) &= \{(k,j)|1 \le k \le n-1, k+1 \le j \le n+1\} \\ &\quad -\{(k,j)|1 \le k \le n-1, k+1 \le j \le n+1\} \\ &= \emptyset \end{aligned}$$

and

$$\begin{aligned} P_{\text{end}}(T_2) &= \{(k,j)|1 \le k \le n-1, k+1 \le j \le n+1\} \\ &\quad -\{(k,j)|1 \le k \le n-2, k+1 \le j \le n+1\} \\ &= \{(k,j)|k = n-1, k+1 \le j \le n+1\} \end{aligned}$$

Therefore the set of starting task for the Gaussian Elimination is: $\{T_1(1)\}$ and the set of ending task is: $\{T_2(n-1,j)|n \le j \le n+1\}$.

11 Mapping clusters to processors

Once the clustering function is found. We can execute any instantiated task graph on a parallel machine. However, for most of the instances of a PTG the number of clusters is larger than the number of available processors. Hence, we need to map clusters to processor. In this section we first show how to symbolically evaluate the number of clusters. Second, we describe the different mapping function we propose.

11.1 Evaluating the number of clusters

For each generic task T we compute the set $E_T = \{\kappa|\exists \vec{u} \in \text{valid}(T), \vec{\alpha}\vec{u} + \beta + \vec{\gamma}\vec{p} = \kappa\}$. E_T is the set of valid cluster numbers. The set $E = \cup_T E_T$ is the union, for each generic tasks, of valid cluster numbers. We can symbolically compute this set with the omega calculator from the university of Maryland [25]. The number of element of E is the number of clusters. We can build a function that enumerate each part of E that is a convex polytope. We use the *polylib* from Université de Strasbourg [4], to build such a polynomial. Due to internal algorithms of the omega calculator, some parts of E may contain existential quantifiers. In that case we generate a loop nest that scan E: the number of valid iterations is the number of points of E.

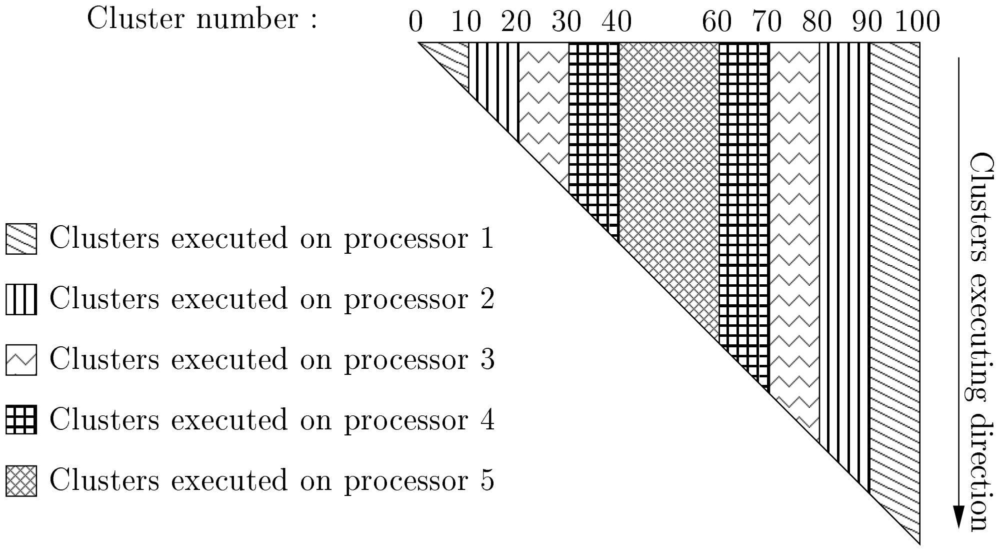


Figure 10: Allocating 100 clusters on a 5 processors parallel machine using the reflect mapping, for a triangular task graph

11.2 Mapping functions

We propose four mapping functions. Let P the number of available processors. Let c the cluster number to be mapped on a processor. p the processor number where c is mapped.

1. *Cyclic.* This is the default option. We uniformly map clusters to processors:

   $$p = c \bmod P$$

   . Such a function has good load balance properties.

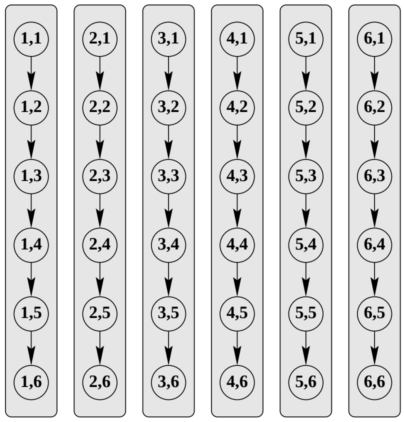
2. *Cyclic*(n). We can n grouped contiguous clusters to the same processors. In that case

   $$p = (c \div n) \bmod P$$

   The two previous functions does not require to know the number of clusters.

3. *Bloc.* If we know that we have N clusters, we can suppress residual communications. In order to do that we perform a Cyclic$(N \div P)$ mapping. In general this allocation does not have good load balancing properties.

4. *Reflect.* We expect this mapping function to have better load balancing properties. Let $n = N \div (2P)$. If $c \geq N/2$ then we pose $c = N - c$. The mapping function is then $p = (c \div n) \bmod P$. This is the same as creating $2P$ blocs of size $N/(2P)$ and allocating these blocs by folding one half on the other. An example of such a mapping is shown Figure 10 for a triangular task graph.



$$\kappa(T_1, (i, k)) = i$$

Figure 11: SLC for Matrix Multiplication ($n = 6$)

12 Results

We have tested SLC on several compute intensive kernels that can be found in scientific applications. First allocation function as well as clustering are shown. Second we compare our algorithm with static scheduling algorithm: DSC by Yang and Gerasoulis [30] and CASS-II by Liou and Palis [20]. Since SLC is symbolic it can schedule very large task graphs in a time independent from the parameter values. However, this algorithm loses its interest if the found solution are far from those that can be found by non symbolic algorithm. Thus is why we have decided to compare SLC to algorithms designed for instantiated task graphs.

12.1 SLC for compute intensive kernels

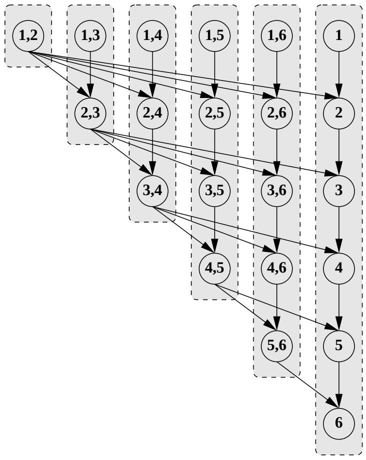
Figures 11, 12, 13, 14 gives the allocation function and the clustering for instantiated task graphs for matrix multiplication, Jordan Diagonalization, Givens Algorithm, and Cholesky algorithm. Gaussian Elimination is not shown because it has been fully detailed in the previous section. Figure 15 shows the Gaussian elimination with backward substitution. In that case allocation function required parameters. Otherwise rules that describe communications from tasks $T_1(i, j)$ to tasks $T_1(i + 1, j)$ and from tasks $T_1(j - 1, j)$ to tasks $T_2(n - j + 1)$ cannot be zeroed.

12.2 Comparison with static scheduling algorithms

We compare here the performance of the SLC algorithm for an unbounded number of processors with DSC and CASS-II. These two algorithms required an instantiated task graph. Hence they cannot handle large task graphs (It takes hours to schedule one this kernels for parameter values 400 and the memory is not sufficient to schedule these kernels with parameter values larger than 1000). We have used the code granted by the authors of these algorithms.

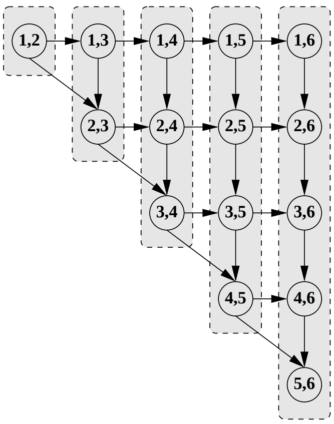
12.2.1 Comparison for a bounded number of processors

Table 1 show comparisons between the clustering found by SLC and the clustering found by DSC and CASS-II for parameter values of 100, 200 and 400. For the Cholesky algorithm it has not been possible to perform the test for matrix size larger than 200. It has not been possible to perform test



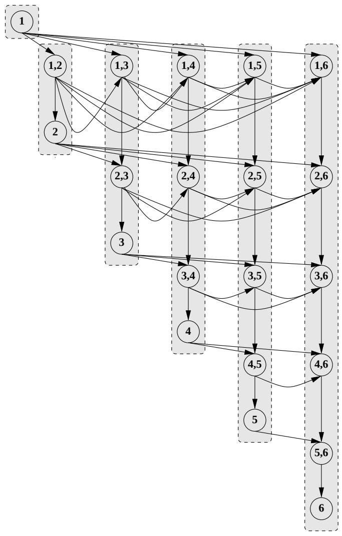
$$\kappa(T_1, (k, j)) = j$$
$$\kappa(T_2, (k)) = k$$

Figure 12: SLC for Jordan Diagonalization ($n = 6$)



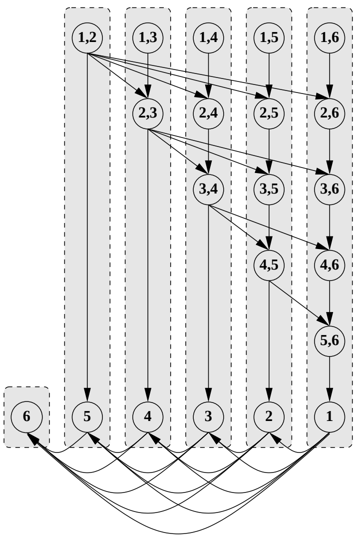
$$\kappa(T_1, (i, j)) = j$$

Figure 13: SLC for Givens Algorithm ($n = 6$)



$\kappa(T_1, (k)) = k$
$\kappa(T_2, (k, i)) = i$

Figure 14: SLC for Cholesky Algorithm($n = 6$)



$\kappa(T_1, (i, j)) = n - i + 1$
$\kappa(T_2, (i)) = i$

Figure 15: SLC for Gaussian Elimination and Backward Substitution($n = 6$)

Graph	n	Seq. time	# clusters DSC	Schd. length DSC	# clusters CASSII	Schd. length CASSII	#clusters SLC	Schd. length SLC	R_{DSC}	R_{CASSII}
Gauss	100	1009800	101	25093	101	29797	102	30001	0.84	0.99
Gauss	200	8039600	201	100193	201	119597	202	120001	0.83	1.00
Gauss	400	64159200	401	400393	401	479197	402	480001	0.83	1.00
Jordan	100	1509950	101	40002	101	40202	101	40202	1.00	1.00
Jordan	200	12039900	201	160002	201	160402	201	160402	1.00	1.00
Jordan	400	96159800	401	640002	401	640802	401	640802	1.00	1.00
Givens	100	3720750	98	166393	99	170841	101	175906	0.95	0.97
Givens	200	39214956	318	671163	297	688567	201	755580	0.89	0.91
Givens	400	257418656	496	2668550	497	2737167	401	2844880	0.94	0.96
Gauss + back. sub.	100	1025250	99	36038	100	36038	101	36335	0.99	0.99
Gauss + back. sub.	200	8100500	199	142088	200	142088	201	142685	1.00	1.00
Gauss + back. sub.	400	64401000	399	564188	400	564188	401	565385	1.00	1.00
Cholesky	100	510150	100	32156			101	30100	1.07	
Cholesky	200	4040300	239	121759			201	120200	1.01	

Table 1: Comparison of SLC with DSC and CASS-II on an unbounded number of processors

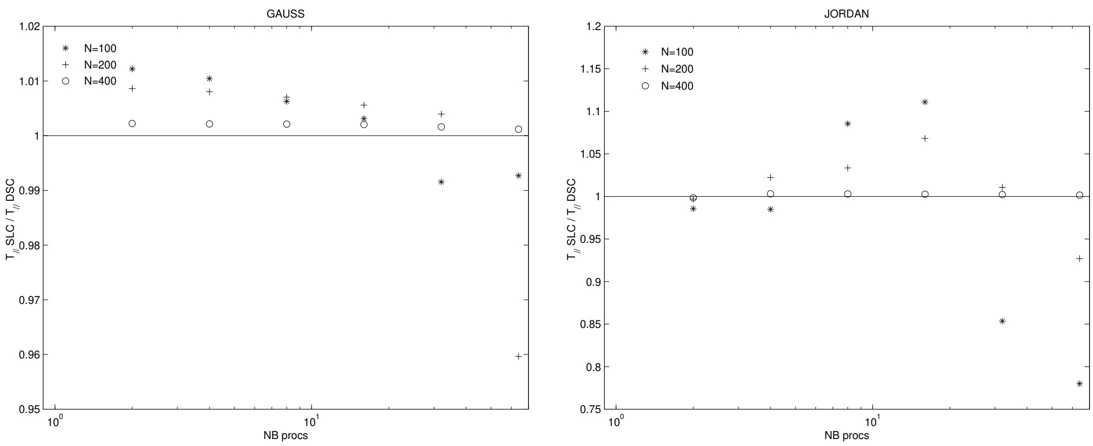


Figure 16: Scheduling ratio between DSC and SLC for 2 to 64 processors for the Gaussian Elimination and the Jordan Diagonalization.

for the Cholesky algorithm with CASS-II even for small matrices (DSC is less memory consuming than CASS-II). This is because Cholesky algorithm task graph is very large (for $n = 400$ there is 80 200 nodes and 10 746 600 vertices, the space needed to store such a task graph is 117 Mo). Results for matrix multiplication is not shown because SLC, DSC and CASS-II all find the optimal schedule which yields no difference.

Column *sequential time* is the sum of all tasks duration. Columns *# clusters* is the number of clusters found by the algorithm. Columns *Schd. length* is the length of the schedule found by the corresponding algorithm. Column R_{DSC} is the ratio between DSC schedule length and SLC schedule length. Column R_{CASSII} is the ratio between CASS-II schedule length and SLC schedule length. If R is greater than 1, this means that SLC outperform the static scheduling algorithm.

Results show that R is never smaller than 0.83. Furthermore in several cases SLC outperform the two other algorithms. Hence despite the fact that SLC computes the schedule for all task graph instance and DSC and CASS-II compute a schedule only for one instance, schedules found by SLC are highly competitive to the DSC or CASS-II schedule. We recall here that SLC schedules are computed in a much shorter time (some seconds for SLC versus many minutes to hours for CASS-II and DSC) with constant memory requirement.

12.2.2 Comparison for an unbounded number of processors

We have use the Pyrros [28] scheduling tool for simulating scheduling execution found by SLC and DSC for an unbounded number of processors. Cluster mapping is done by the RCP* [29] algorithm for DSC and by cyclic mapping for SLC. Figures 16, 17 and 18 show the ration between DSC and SLC solution. When the ratio is greater than 1 SLC outperform DSC. Results show that this ration is between 0.75 and 1.4. Thus scheduling quality of SLC is also good on a bounded number of processors: most of the time we find equivalent schedules.

13 Related works

There are a number of papers addressing computation mapping to processors. For loop programs, the SUIF compiler [2] uses a mapping algorithm [3] which maximizes parallelism and minimizes

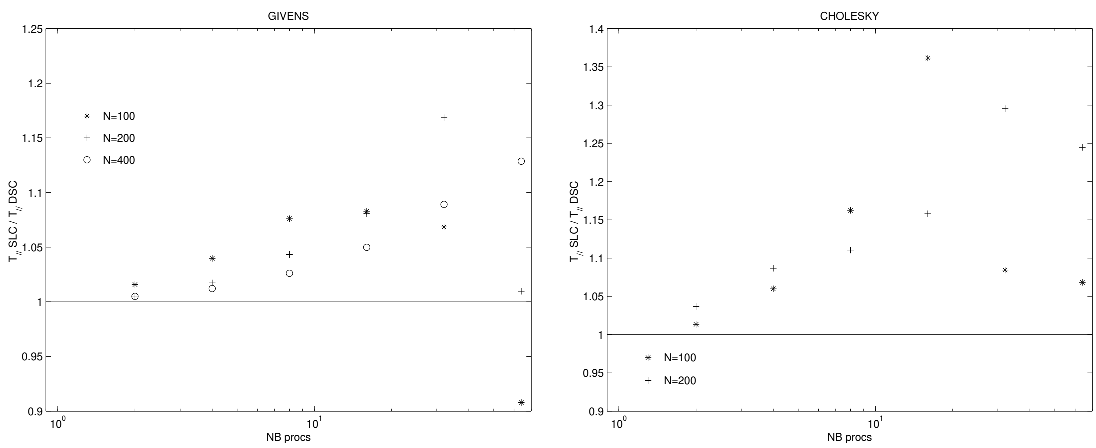


Figure 17: Scheduling ratio between DSC and SLC for 2 to 64 processors for Givens and Cholesky algorithms.

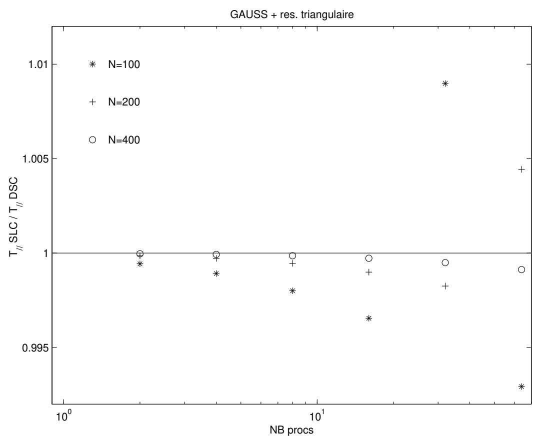


Figure 18: Scheduling ratio between DSC and SLC for 2 to 64 processors for Gaussian elimination with backward substitution.

communication for `doall` or `forall` parallelism. Both data mapping and computation allocation functions are affine.

In [10], Darte and Robert have introduced the "*communication graph*" model, which describes read/write data dependence in the case of affine loop nests. For the case of one uniform loop nest, they prove that the problem of aligning computation with data in order to minimize communication is NP-Complete. They propose a heuristic for mapping loop nests for aligning computation with data. This work is further extended by Dion and Robert in [11]. They introduce the "*access graph*" model which is a communication graph whose edges are labeled with the access matrix of data arrays. They prove that aligning computation with data in order to minimize the residual communication is an NP-Complete problem for the case of one uniform loop nest.

Feautrier in [14] proposes an algorithm for "*cutting*" edges in a data flow graph. Edges are selected in a greedy manner and a Gaussian elimination is performed to find the allocation function. Data are then mapped according to the owner compute rule. In [15], Feautrier extended his previous results in order to map data more efficiently and the owner-compute rule is not always respected.

In [22], Mongenet addresses the problem of data allocation and instruction scheduling when programs are modeled as a system of recursive affine functions. Dependences are sorted according to the dimension of transmitted data. She gives a condition on dependences such that they do not imply communication in parallel execution. She shows how point-to-point communication and broadcast can be determined. Once data are mapped, the owner compute rule is applied for allocating computation.

All these works share a common characteristic: allocation/mapping functions are affine. None of these works can address affine piecewise functions. Also these work addresses either fine-grain parallelism or DOALL parallelism. Our setting is more complicated and deals with symbolic coarse-grain DAG parallelism.

14 An Open Problem: The Rule Fusion Problem

Merging rules is an important problem. It reduces the number of rules while preserving the semantic of the parameterized task graph. Hence SLC is faster and there will be fewer messages when executing the schedule. In figure 4 we have presented an algorithm that is able to suppress some multiple-edges. However it may happen that some rules does not describes multiple-edges but can be merged. This is the case for the three following rules:

$$T_1(k) \rightarrow T_1(k+1) : b(i) | 1 \leq k \leq n-1, k+1 \leq i \leq n$$

$$T_1(k) \rightarrow T_1(k+1) : b(k) | 1 \leq k \leq n$$

$$T_1(k) \rightarrow T_1(k+1) : b(i) | 2 \leq k \leq n, 1 \leq i \leq k-1$$

That can be merged into

$$T_1(k) \rightarrow T_1(k+1) : b(i) | 1 \leq k \leq n, 1 \leq i \leq n$$

The rule merging algorithm we have designed works only for rules transmitting contiguous data of a given edge. An algorithm able to merge rules in a more general case would be very welcome.

15 Conclusion

In this article we have presented in detail SLC an algorithm for symbolic allocation of parameterized task graphs. The advantages of such an approach is that the found allocation is independent from the parameter values and its time and memory cost is constant. Our contribution is the following:

- we have presented a rule merging algorithm that permit to reduce the number of rules of a PTG while preserving its semantic,
- we have shown how, given a PTG, extracting rules, which once zeroed leads to a linear clustering. In order to achieve this goal we have introduce the notion of bijection rule and rule conflict. The rule selection can be made using a greedy algorithm for large PTGs or after a maximal independent set search for standard and small PTGs,
- concerning cluster identification, we have augmented Feautrier's algorithm with graph splitting in order to derive a piecewise linear functions for allocating clusters to processors,
- we have compared SLC to two of best and fastest static scheduling algorithms. Results show that, despite the fact that our techniques derives a solution for all graph instances, our symbolic algorithm is very competitive with these two static algorithm.

References

[1] *Nonlinear Assignment Problems: Algorithms and Applications*, chapter Symbolic Scheduling of Parameterized Task Graphs on Parallel Machines. Kluwer books, 2000.

[2] S.P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[3] 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.

[4] 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.*

[5] M. Cosnard and E. Jeannot. Compact DAG Representation and Its Dynamic Scheduling. *Journal of Parallel and Distributed Computing*, 58(3):487–514, September 1999.

[6] M. Cosnard, E. Jeannot, and L. Rougeot. Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs. In *IEEE International Parallel Processing Symposium (IPPS'98)*, Orlando, Florida, April 1998. IEEE.

[7] M. Cosnard, E. Jeannot, and T. Yang. Symbolic Partitionning and Scheduling of Parameterized Task Graphs. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS'98)*, Tainan, Taiwan, December 1998.

[8] M. Cosnard, E. Jeannot, and T. Yang. SLC: Symbolic Scheduling for Executing Parameterized Task Graphs on Multiprocessors. In *International Conference on Parallel Processing (ICPP'99)*, Aizu Wakamatsu, Japan, September 1999.

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

[10] A. Darte and Y. Robert. On the Allignment Problem. *Parallel Processing Letters*, 4(3):259–270, 1993.

[11] M. Dion and Y. Robert. Mapping Affine Loop Nests : New Results. In *Int. Conf. on High Performance Computing and Networking, HPCN'95*, pages 184–189, May 1995.

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

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

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

[15] P. Feautrier. Distribution automatique des données et des calculs. *T.S.I.*, 15(5):529–557, 1996.

[16] M.R. Garey and D.S. Johnson. *A Guide to the Theory of NP-Completeness.* W.H. Freeman and company, New-York, 1979.

[17] 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.

[18] Emmanuel Jeannot. *Allocation de graphes de tâches paramétrés et génération de code.* PhD thesis, École Normale Supérieure de Lyon, France, October 1999. `ftp://ftp.ens-Lyon.fr/pub/LIP/Rapports/PhD/PhD1999/PhD1999-08.ps.Z`.

[19] 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.

[20] J.-C. Liou and M. A. Palis. A New Heuristic for Scheduling Parallel Programs on Multiprocessor. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 358–365, Paris, October 1998.

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

[22] C. Mongenet. Affine Dependence Classification for Communications Minimization. *IJPP*, 25(6), December 1997.

[23] 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.

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

[25] W. Pugh. The Omega Test a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, August 1992. (website: `http://www.cs.umd.edu/projects/omega`).

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

[27] M. Wu, W. Shu, and Y. Chen. Runtime Parallel Incremental Scheduling of DAGs. In *International Conference on Parallel Processing 2000 (ICPP'2000)*, Toronto, August 2000.

[28] 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.

[29] T. Yang and A. Gerasoulis. List scheduling with and without communication delay. *Parallel Computing*, 19:1321–1344, 1993.

[30] 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.