Compact DAG Representation and its Symbolic Scheduling

Michel Cosnard
INRIA Sophia Antipolis
France

Emmanuel Jeannot
LORIA
Université Nancy I Henri Poincaré
France

Tao Yang
CS. Dept. UCSB
USA

April 10, 2002

Abstract

Task graph scheduling has been found effective in performance prediction and optimization of parallel applications. A number of static scheduling algorithms have been proposed for task graph execution on parallel machines. Such an approach cannot be adapted to changes in values of program parameters and the number of processors and it also cannot handle large task graphs. In this paper, we model parallel computation using parameterized task graphs which represent coarse-grain parallelism independent of the problem size. We present a symbolic scheduling algorithm for a parameterized task graph which first derives linear clusters and then assigns task clusters to processors. The runtime system executes clusters on each processor in a multi-threaded fashion. We evaluate our method using various compute-intensive kernels that can be found in scientific applications.

1 Introduction

Directed acyclic task graphs (DAGs) have been used in modeling parallel applications and performing performance prediction and optimization [3, 10, 15, 17, 28]. There are a number of algorithms which have been proposed to perform static task graph mapping on distributed memory machines [12, 20, 21, 24, 25, 32]. Because a task graph is obtained statically and the number of processors must be given in advance before scheduling, these methods present two major drawbacks:

- Static scheduling is not *adaptive*. Each time the problem parameter values or the number of available processors change, a scheduling solution has to be recomputed.
- Static scheduling is not *scalable*. For large problems, the corresponding task graph may contain a large number of tasks and dependence edges, and a scheduler that examines the entire task graph might fail due to memory constraints.

Our previous work [5, 19] addresses the above issues by maintaining a small portion of the task graph in memory and using dynamic scheduling. In [29] Wu et al. present an algorithm which also incrementally expands a task graph and uses a parallel dynamic scheduling algorithm. These approaches still suffer from runtime overhead for scheduling. Also, the schedule has to be recomputed each time the parameters of the graph change.

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 model for representing coarse-grain parallel computation. A PTG models a DAG with parameters that can be instantiated at runtime. Thus such a graph is symbolic and its size does not vary if program the parameters change. We present here an algorithm, called SLC (Symbolic Linear Clustering), that builds a linear clustering from a PTG. A task graph is not instantiated for finding an allocation and hence, SLC manipulates a task graph symbolically, independently of the graph parameter values. Thus an SLC schedule is adaptive because this allocation is valid for all parameter values and any number of processors.

The rest of this article is organized as follows. 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 an overview of SLC. Section 5 presents rule merging for reducing the algorithm complexity. Section 6 describes our method for extracting bijection rules that are used for linear clustering. Section 7 presents a symbolic clustering algorithm. Section 8 discusses how clusters can be numbered explicitly so that symbolic clusters can be evenly mapped to physical processors. Section 9 describes a mapping from clusters to a given number of processors and addresses the issue of finding the starting and ending task of each symbolic cluster for the program execution. Section 10 presents our experimental results. Section 11 presents related work on symbolic computation allocation. Section 12 discusses an open problem. Section 13 concludes the paper.

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 use 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, $\tau_i \in T$ represents 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 [18] where $pred(i)$ is the set of the predecessors of task i and $succ(i)$ is the set of the successors of task i:

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

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

- A *clustering* is a mapping of all the tasks of G into clusters. A group of tasks that are in the same cluster will be executed on the same processor. 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. Hence, a linear clustering is a set of disjoint paths in the task graph.

- A *parameterized task graph* (PTG) is a compact and symbolic representation of some task graphs that can be found in scientific applications. It was first proposed by Cosnard and Loi in [8] 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 sequentially. 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 has to perform before completion, 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, emission and reception rules, which model how tasks send and receive data. Reception and emission rules are dual forms of each other. Reception rules describe a set of parents that a task depends on. Emission rules describe a set of children that a task needs to send data. An emission rule R (a reception rule) has the form:

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

where $\vec{u}$ and $\vec{v}$ are the iteration vectors of tasks T_a and T_b. Rule R means that if predicate P is true, task $T_a(\vec{u})$ sends data $D(\vec{y})$ to task $T_b(\vec{v})$. Vector $\vec{y}$ describes which part of the data D is sent to task T_b. P is a polyhedron which describes valid values of vectors $\vec{u}$, $\vec{v}$, and $\vec{y}$.

- 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

We have used the PlusPyr software tool [22] in order to construct a PTG from a sequential program. PlusPyr is able to derive a parameterized task graph for a sequential program if a user also provides an annotation describing how this program is partitioned into tasks.

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. The instantiated task graph is shown for `n` $= 7$. The teration vector of T_1 is (k) and the 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 $T_2(k,j)$ 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 Overview of the SLC scheduling

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 processors for execution. The optimization goal of our symbolic clustering algorithm is the same as that of static scheduling algorithms [28, 32]: eliminate unnecessary communication and preserve data locality, map symbolic clusters to processors evenly to achieve load balance, and execute clusters within each processor using the multithreading technique to overlap computation with communication. The mapping and scheduling process is symbolic in the sense that the change in the problem size and the number of processors does not affect the solution derived by our algorithm.

```
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
```

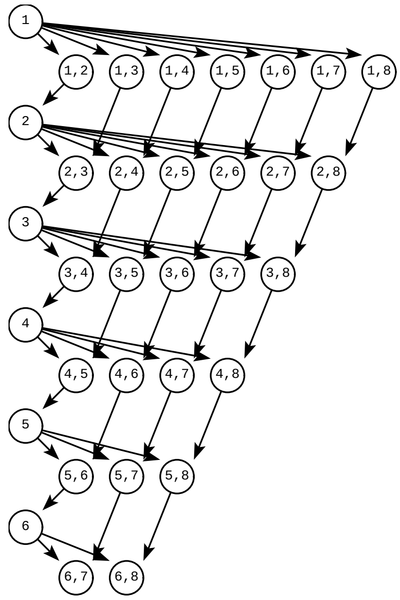


Figure 1: Gaussian Elimination and Instantiated Task Graph (n=7)

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

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

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

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

Figure 2: Emission rules for the Gaussian Elimination

The main steps involved in our method for scheduling and executing a parameterized task graph are summarized as follows:

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: fewer rules, the faster the algorithm. Moreover, the size and the speed of the code generated from the PTG depends on the number of rules. Hence it is important to have as few 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 they 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 guarantee that, for all parameter values, the selected rules will always

form a linear clustering,

4. *computing the symbolic allocation.* Once the linear clustering is built, 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.

5 Merging and optimizing rules

We first discuss a method to merge rules when feasible, to reduce the complexity of scheduling. Merging is possible when two rules describe the same set of edges and the data transmitted are contiguous. In this 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 rule set of 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 emission rules for Gaussian Elimination

Entry: Two communication rules R_1 and R_2.
Output: One rule R, merging of R_1 and R_2 if possible.

```
Merge(R1,R2)
    if same_edges(R1,R2) then
        if transmitted data (D) is the same for R1 and R2 then
            S1 = domain of D that is transmitted by R1;
            S2 = domain of D that is transmitted by R2;
            if adjacent(S1,S2) then
                R = R1 where D is transmitted on the S1 ∪ S2 domain;
```

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

In the 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.

6 Extracting bijection rules for linear clustering

A bijection rule is a communication rule that describes 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]:

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

R is a bijection rule if for any valid instance of $\vec{u}$ there 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 $\vec{\imath}$ be the vector made of the program parameters and the loop indices of the input program taken in lexicographic order. Therefore there 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, the matrices D_1 and D_2 are built. Then we find matrix E_1 and vector w_1 such that:

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

There exists many E_1 and w_1 which verifies the previous equation. We impose that $\vec{w_1}$ is composed of variables which do not appear in $\vec{u}$. Formally this is written $w_{1j} = \delta_j i_j$ where w_{1j} (resp. i_j) is the j[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

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

Similarly we can build E_2 and w_2 such that:

$$\vec{u} = D_1E_2(\vec{v} - \vec{k_2}) + D_1\vec{w_2} + \vec{k_1}. \tag{2}$$

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 versa: the rule is a bijection rule.

Let us 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 which do 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 which do not appear in $\vec{u}$. Therefore $\vec{w_1}$ is composed of variables which do 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.

∎

[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.

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

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

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

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.

7 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 tasks 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 be the mapping function of tasks: $\kappa(T_a(\vec{u}))$ is the cluster number of the task $T_a(\vec{u})$. Let $R : T_a(\vec{u}) \to T_b(\vec{v}) : D(\vec{y}) | P$ be 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.

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.

7.1 Determining Conflicting Rules

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

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

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*

and

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

In order to know if R_1 and R_2 are in concflict we consider the following 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. There 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 $\vec{u_1} = \vec{w_1}$.
  3. Let $\vec{v_1}$ be the image[2] of $\vec{u_1}$ according to rule R_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:
  1. $T_b = T_d$.
  2. There 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 $\vec{u_1}$ be 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 follows. 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. The sets I, $R_1\backslash I$, and $R_2\backslash I$ can be symbolically computed with the omega calculator from the University of Maryland [27].

Example Let consider the two following rules:

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

and

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

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

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

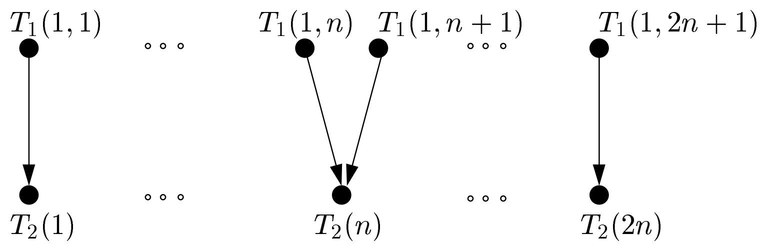


Figure 6: Example of join conflict: for $T_2(n)$ has two predecessors

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

and

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

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:

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

and

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

where R_1 and R_2 could 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 both be zeroed for describing a linear clustering.

7.2 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 zeroing a rule corresponds to suppressing communications, we need to zero rules that have large communication cost. We propose here two complementary approaches.

Greedy selection In our previous work [6, 7, 26] we have proposed a greedy method that is composed of two steps:

1. sorting the rules such that most the most costly (in terms of communications) will be considered 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*. Here, each node is a bijection rule and there is an edge between two nodes if the two corresponding rules are in conflict. Finding an independent set of this graph means building a subset of bijection rules that are not in conflict.

Moreover, we want to zero rules that imply large communication costs and we want to zero as many rules as possible. Zeroing as many rules 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 weights of the selected nodes. This method has the advantage of being more general than the greedy method 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 into 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.

7.3 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 $\mathcal{G}$ bea 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 rule 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 merging some linear clusters. Since no rules in $\mathcal{Z}$ are in conflict, there will be no independent task in the resulting clusters.

8 Cluster Identification

The previous algorithm zeros a number of rules and as a result it zeros a number of edges in the instantiated task graph for a given set of program parameter values. However, there is no cluster ID given for each cluster. We need such an ID number explicitly so we can assign clusters evenly to physical processors using a symbolic mapping function. In static scheduling research [28, 32], producing such a number is easy since clusters can be explicitly numbered.

Given a rule zeroing of a PTG, we show how to explicitly build a symbolic function $\kappa(T_a, \vec{u})$ such that for any generic task T_a and any valid instance $\vec{u}$, this function gives a cluster number for this task instance. We call this procedure *cluster identification* or *cluster numbering*.

Notice that all the tasks in the same cluster will have the same cluster number and tasks from different clusters should not have the same cluster number.

8.1 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 the Feautrier algorithm [13]. As Feautrier, we assume that our allocation function is affine with respect to the program parameters and the 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 rules 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. To solve 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 of subtracting from all the β's and from all γ_i's the smallest 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 8.3.

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

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

and

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

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

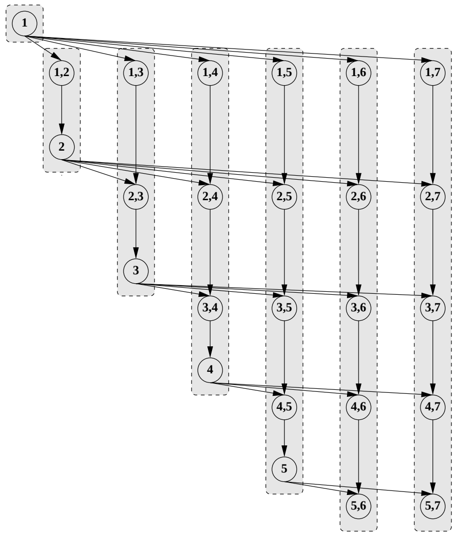


Figure 7: A clustering for an instantiated Gaussian Elimination graph with $n = 6$.

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

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

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

and

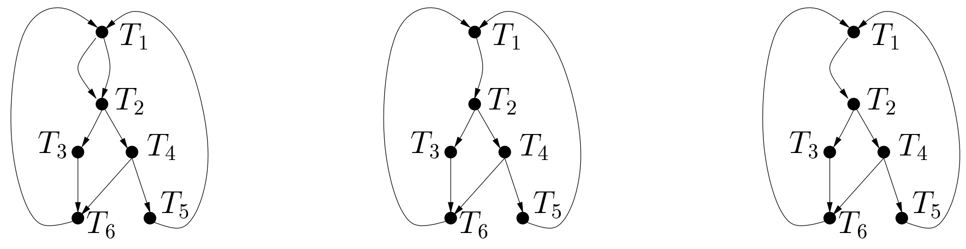
$$R_2 : T_1(k, j) \rightarrow T_1(k, j+1) | 1 \leq k \leq n, k+1 \leq j \leq n.$$

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:

$$\kappa(T_1, (k, j)) = \begin{cases} j & \text{if } 1 \leq k \leq n, 1 \leq j \leq k \\ k & \text{otherwise} \end{cases}$$

In the general case we proceed as follows. 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 little 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 each 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 in the graph up to generic task T_a where the graph has been split. 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 $\vec{u}$ determines the subgraph and the identification function which is to be applied.

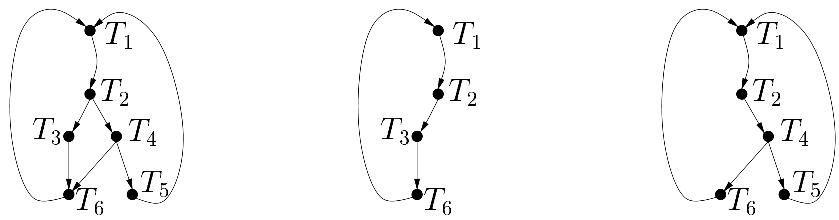
When a PTG has to be split we first suppress multiple edges as shown in Figure 8.



G G_1 G_2

Figure 8: PTG G is split 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 with output degree greater than 1 as in Figure 9.



G_1 G_{1_1} G_{1_2}

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

In the worst case it may happen that we obtain a graph that cannot be split 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 this means that G describes a path in all instantiated graphs hence the mapping function must not depend on the iteration vector.
- We propose a runtime solution that consists of mapping the first task in a cluster whose 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 of suppressing rules from the $\mathcal{Z}$-set and recomputing 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.

9 Mapping clusters to processors

Next, we need to map clusters to processors. In this section we discuss how to find the starting and ending tasks of clusters for runtime cluster execution within each processor, describe the different mapping functions, and then we show how to evaluate symbolically the number of clusters.

9.1 Finding starting tasks and ending tasks

In order to execute the derived allocation we need 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 be a generic task. We compute the polyhedron $P_{\text{start}}(T_a)$ that describes the iteration vector values for which T_a has no predecessor and $P_{\text{end}}(T_a)$ the polyhedron that describes the 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 follows. Let $\text{emit}(T_a)$ be 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 $\text{recep}(T_a)$ be the set of T_a instances that receive data. In this 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 $\text{valid}(T_a)$ be the set of all valid instances of T_a: it is the union of $\text{emit}(T_a)$ and $\text{recep}(T_a)$. 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 have no successor. $P_{\text{end}}(T_a) = \text{valid}(T_a) - \text{emit}(T_a)$ is the set of valid instancees of T_a that do not receive data and thus have 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

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

and

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

For task T_2 of the Gaussian Elimination we have:

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

and

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

Therefore

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

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

9.2 Mapping functions

We use four mapping functions. Let P be the number of available processors. Let c be the ID of a cluster to be mapped on a processor. Let p be the processor ID where cluster 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 balancing properties.

2. *Cyclic*(n). We can group contiguous clusters to the same processor. In this case

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

   The previous two functions do not require knowing the number of clusters.

3. *Block.* 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 $n = N \div (2P)$. We pose

   $$c' = \begin{cases} c \text{ if } c < N/2 \\ N - c \text{ otherwise.} \end{cases}$$

   The mapping function is then $p = (c' \div n) \bmod P$. This is the same as creating $2P$ blocks of size $N/(2P)$ and allocating these blocks by folding one half on the other. An example of such a mapping is shown Figure 10 for a triangular task graph.

9.3 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 of valid cluster numbers. We can symbolically compute this set with the omega calculator from the university of Maryland [27]. The number of elements in E is the number of clusters. We can build a function that enumerates each part of E that is a convex polytope. We use the *polylib* library from Université de Strasbourg [4], to build such a polynomial. Due to some internal algorithms of the omega calculator, some parts of E may contain existential quantifiers. In this case we generate a loop nest that scans E: the number of valid iterations is the number of points of E.

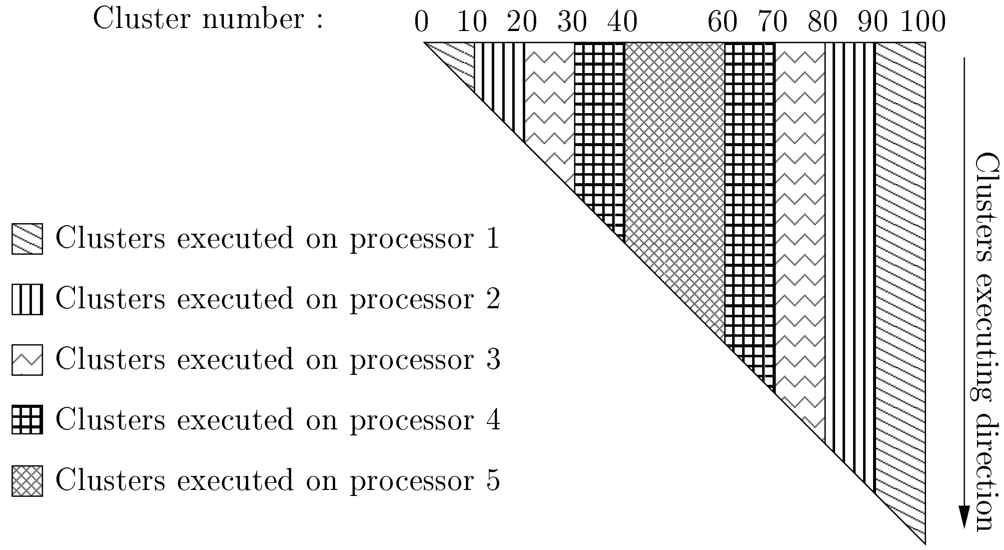


Figure 10: Allocating 100 clusters on a 5 processors parallel machine using the reflect mapping, for a triangular task graph

10 Results

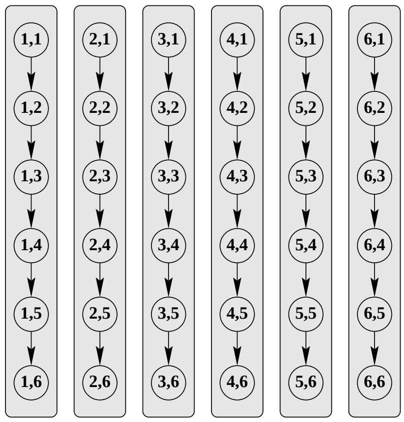
We have tested SLC on several compute intensive kernels that can be found in scientific applications. First, we present the allocation function as well as the clustering. Second we compare our algorithm with two static scheduling algorithms: DSC by Yang and Gerasoulis [32] and CASS-II by Liou and Palis [21]. Since SLC is symbolic it can schedule very large task graphs in a time independent manner. However, this algorithm loses its interest if the resulting solutions are far from those that can be found by non-symbolic algorithms. This is why we have decided to compare SLC to algorithms designed for instantiated task graphs.

10.1 SLC for compute intensive kernels

Figures 11, 12, 13, and 14 give 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 sections. Figure 15 shows the Gaussian elimination with backward substitution. In this case the 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.

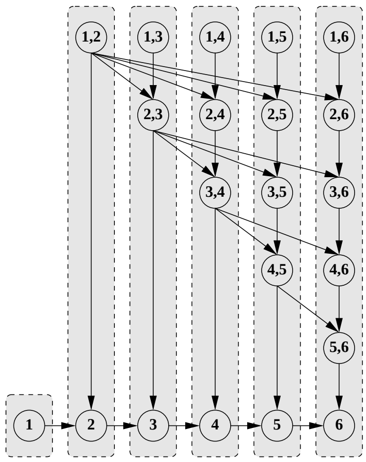
10.2 Comparison with static scheduling algorithms

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 for parameter values larger than 1000. We have used the code granted by the authors of these algorithms.



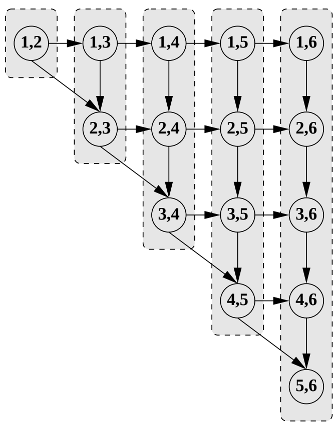
$$\kappa(T_1, (i, k)) = i$$

Figure 11: SLC for matrix multiplication ($n = 6$)



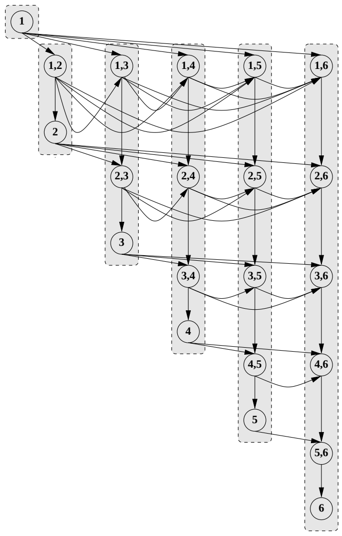
$$\kappa(T_1, (k, j)) = j$$
$$\kappa(T_2, (k)) = k$$

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



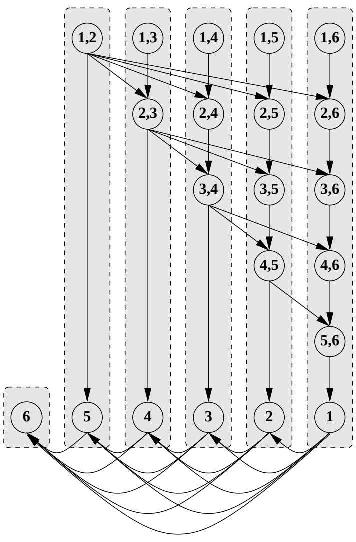
$$\kappa(T_1, (i, j)) = j$$

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 with backward substitution($n = 6$)

Graph	n	Seq. length	# 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 algortihms on an unbounded number of processors

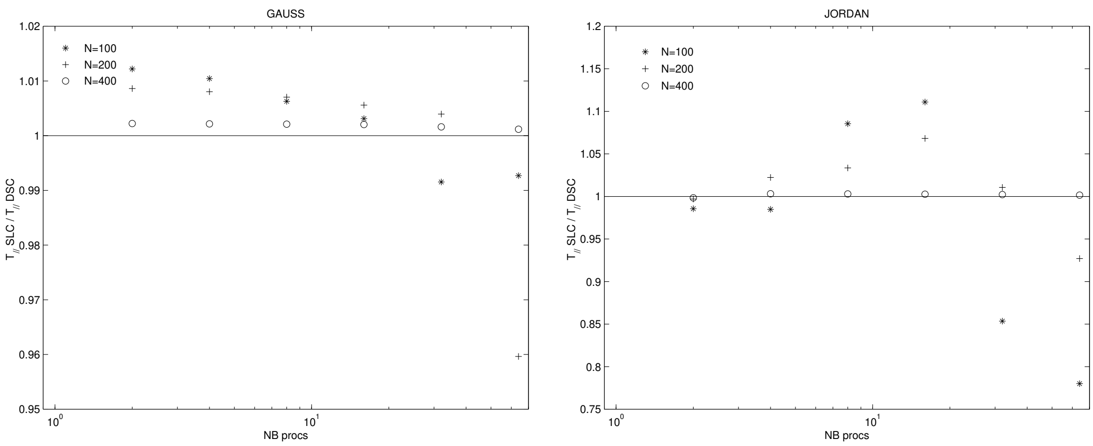


Figure 16: Scheduling ratio between DSC and SLC for 2 to 64 processors for the Gaussian Elimination and the Jordan Diagonalization.

10.2.1 Comparison for an unbounded 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 was not possible to perform the test for matrix size larger than 200. It has not been possible to perform test 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 edges, the space needed to store such a task graph is 117 Mo). Results for matrix multiplication are not shown because SLC, DSC, and CASS-II all find the optimal schedule yeilding no difference.

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

The result show that R is never smaller than 0.83. Furthermore in several cases SLC outperforms the two other algorithms. Hence despite the fact that SLC computes the schedule for all task graph instances and DSC and CASS-II compute a schedule for only one instance, schedules found by SLC are highly competitive to the DSC or CASS-II schedules. We recall as well 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 a constant memory requirement.

10.2.2 Comparison for a bounded number of processors

We have used the Pyrros [30] scheduling tool for simulating scheduling execution found by the SLC and the DSC algorithms for a bounded number of processors. Cluster mapping is done by the RCP* [31] algorithm for DSC and by the cyclic mapping for SLC. Figures 16, 17, and 18 show the ratio between the DSC and the SLC solution. When the ratio is greater than 1 SLC outperforms DSC. The Result show that this ratio is between 0.75 and 1.4. Thus the scheduling quality of SLC

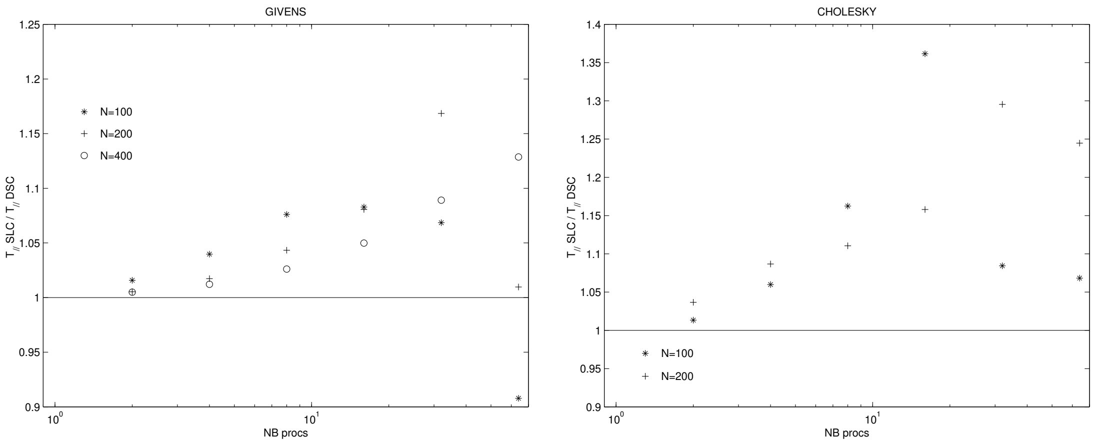


Figure 17: Scheduling ratio between DSC and SLC for 2 to 64 processors for Givens and Cholesky algorithms.

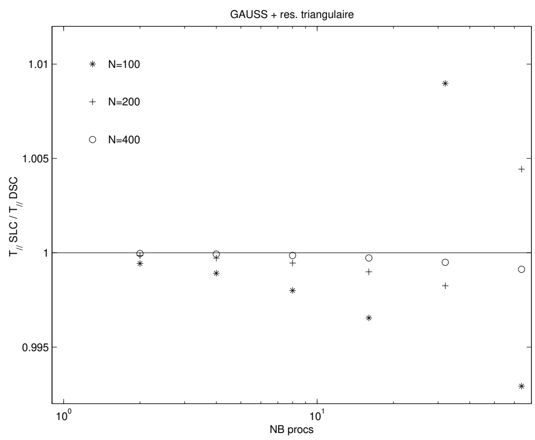


Figure 18: Scheduling ratio between DSC and SLC for 2 to 64 processors for Gaussian elimination with backward substitution.

is also good on a bounded number of processors: most of the time we find equivalent schedules.

11 Related work

There are a number of papers addressing computation mapping to processors. For loop programs, the SUIF compiler [1] uses a mapping algorithm [2] which maximizes parallelism and minimizes communication for `doall` and `forall` parallelism. Both data mapping and computation allocation functions are affine.

In [9], Darte and Robert have introduced the "*communication graph*" model, which describes read/write data dependencies 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 [13] 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. The data are then mapped according to the owner compute rule. In [14], Feautrier extended his previous results in order to map data more efficiently and the owner-compute rule is not always respected.

In [23], Mongenet addresses the problem of data allocation and instruction scheduling when programs are modeled as a system of recursive affine functions. Dependencies are sorted according to the dimension of transmitted data. She gives a condition on dependencies such that they do not imply communication in parallel execution. She shows how point-to-point communication and broadcast can be determined. Once the data is mapped, the owner compute rule is applied for allocating computation.

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

12 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 do not describe multiple-edges but can be merged. This is the case for the following three rules:

$$T_1(k) \to T_1(k+1) : b(i) | 1 \le k \le n-1, k+1 \le i \le n$$

$$T_1(k) \to T_1(k+1) : b(k) | 1 \le k \le n$$

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

witch can be merged into

$$T_1(k) \to T_1(k+1) : b(i) | 1 \le k \le n, 1 \le i \le n.$$

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

13 Conclusion

In this paper we have presented the SLC algorithm which symbolically schedules parameterized task graphs and can perform optimization used in static scheduling, independent of program parameter values and the number of processors.

Our contribution is twofold. First, we have shown how to find a linear clustering for a PTG. Second, we have modified and augmented Feautrier's algorithm with graph splitting for deriving an explicit clustering function in order to allocate clusters onto the processors. We have demonstrated that symbolically scheduled tasks can be executed efficiently on multiprocessors and can deliver good performance for several compute-intensive kernel benchmarks. Our experimental results indicate that SLC finds symbolic schedules that are highly competitive with static scheduling algorithms.

14 Acknowledgment

We would like to thank Jean-Claude König for his idea on the maximal independant set search in section 7.2.

References

[1] 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.

[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] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor Runtime Support for Fine-Grained Irregular DAGs. In Rajiv K. Kalia and Priya Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications.*, New York, 1995. Nova Science Publishers.

[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 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.

[7] 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.

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

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

[10] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sunderam-Stukel, H. Wasserman, V.S. Adve, R. Bagrodia, J.C. Browne, E. Houstis, O. Lubeck, J. Rice, P. Teller, and M.K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. In *First International Workshop on Software and Performance*, Santa Fe, USA, October 1998.

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

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

[15] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *ACM/IEEE Supercomputing'96*, Pittsburgh, November 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, J. Jiao, and T. Yang. Scheduling of Structured and Unstructured Computation. In D. Hsu, A. Rosenberg, and D. Sotteau, editors, *Interconnections Networks and Mappings and Scheduling Parallel Computation* , pages 139–172. American Math. Society, 1995.

[18] 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.

[19] 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`.

[20] 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.

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

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

[23] C. Mongenet. Affine Dependence Classification for Communications Minimization. *International Journal of Parallel Programming*, 25(6), December 1997.

[24] 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.

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

[26] Leonidas S. Pitsoulis and Panos M. Pardalos, editors. *Nonlinear Assignment Problems: Algorithms and Applications*, chapter Symbolic Scheduling of Parameterized Task Graphs on Parallel Machines. Kluwer Academic Publishers, November 2000. ISBN 0-7923-6646-8.

[27] 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`).

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

[29] 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.

[30] 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.

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

[32] 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.