SLC: Symbolic Scheduling for Executing Parameterized Task Graphs on Multiprocessors

Michel Cosnard
LORIA INRIA Lorraine
615, rue du Jardin Botanique
54602 Villers les Nancy, France
Michel.Cosnard@loria.fr

Emmanuel Jeannot
LIP ENS-Lyon
46, allée d'Italie
69364 Lyon, France
ejeannot@ens-lyon.fr

Tao Yang
CS Dept. UCSB
Engr Building I
Santa Barbara, CA 93106, USA
tyang@cs.ucsb.edu

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 distributed memory machines. Such an approach cannot be adapted to changes in values of program parameters and the number of processors and also it 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 scheduling algorithm for a parameterized task graph which first derives symbolic 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 [1, 3, 7, 11, 12, 21]. There are a number of algorithms which have been proposed to perform static task graph mapping on distributed memory machines [8, 14, 15, 19, 20, 21, 23, 25]. 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 change or the number of available processors, a scheduling solution has to be recomputed.
- Static scheduling is not scalable. For large problem sizes, the corresponding task graph may contain a large number of tasks and dependence edges, and a static scheduler might fail due to memory constraint.

The previous work in parallelizing compilers [2, 10, 17] has studied the compact parallelism representation based on fine-grain level dependence analysis and their model normally deals with DOALL parallelism. Our goal is to extend these results for exploiting coarse grain DAG parallelism in a symbolic manner, in order to overcome the two drawbacks mentioned above. In this paper we use a parameterized task graph (PTG) [5, 16] to model computation. A PTG can be considered as a DAG, and is augmented by a set of parameters. Thus such a graph is symbolic and its size does not vary if program parameters change. This paper focuses on symbolic scheduling of parameterized task graphs on distributed memory machines.

Our algorithm, called SLC (Symbolic Linear Clustering), first performs symbolic clustering and assigns tasks from the same cluster to the same processor in order to reduce inter-task communication while still preserving available parallelism. Then it assigns symbolic clusters evenly to processors. Since each processor may own several symbolic task clusters, our runtime scheme executes clusters on each processor using a multithreaded and message-driven fashion to overlap computation with communication.

Section 2 describes definitions of parameterized task graphs. Section 3 gives an overview of our approach. Section 4 presents our symbolic clustering method. Section 5 discusses how clusters can be numbered explicitly so that symbolic clusters can be mapped to physically processors. Section 6 presents simulation and experimental results. Section 7 concludes the paper.

2. Definitions and Notations

Parameterized Task Graphs. A parameterized task graph (PTG) is a compact model for parallel computation [5]. It contains a set of symbolic tasks and each sym-

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

Figure 1. Gaussian Elimination

bolic task is represented by a name and an iteration vector. A PTG also contains a set of communication rules that describe data items transfered among tasks. Since each task can contain a set of instructions executed sequentially, this model mainly is targeted at coarse-grain parallelism.

There are types of communication rules which are either emission or reception rules to model how tasks send or 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:

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

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

The number of components in $\vec{y}$ depends on the dimension of data variable D. For example, it is 0 if D is a scalar, and 1 if D is a 1D vector. We assume that the variables $\vec{y}$ do not appear in the predicates describing the variables of $\vec{u}$ and $\vec{v}$. Hence, the data part of a rule can be removed easily without getting *holes* in the polyhedron.

Derivation of a parameterized task graph. We have used the PlusPyr software tool [16] 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 to describe how this program is partitioned into tasks. Figure 1 illustrates an annotated program for Gaussian Elimination without pivoting. It contains two generic tasks $T1(k)$ and $T2(k, j)$. Key words "task" and "endtask" specify the beginning and end of a task and "param" specifies symbolic parameters used in this program, which could vary based on the actual problem size. Figure 5 illustrates an instantiated Gaussian Elimination task graph (along with a linear clustering as we will explain later) when program parameter n is 6.

The current implementation of PlusPyr supports dependence analysis and communication synthesis among tasks if the input program has static control and only deals with matrix vector or scalar computation. The techniques for dependence analysis and communication summarization are based on the work done by Feautrier and others on integer parametric programming (see [9, 22] for an introduction and [5, 6] for more details).

For the input GE program shown in Figure 1, the emission rules of the corresponding parameterized task graph, computed by our tool PlusPyr, are shown in Figure 2. For example, the third emission rule indicates that for any k and j such that $1 \leq k \leq n-2$ and $k+2 \leq j \leq n+1$, task $T2(k, j)$ sends part of column j to task $T2(k+1, j)$.

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

Figure 2. Emission rules for the Gaussian Elimination

Bijective Emission Rules. Our symbolic clustering algorithm identifies a type of rules called *bijection rules* to cluster and we present its definition as follows. Given an emission rule of format $R : Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y})|P$, this rule is **bijective** if for each instance of iteration vector $\vec{u}$, there exists exactly one corresponding instance of iteration vector $\vec{v}$. For example, in Figure 2, the first emission rule is not bijective because this rule describes a dependence from task $T1(k)$ to many of its children $T2(k, k+1), T2(k, k+2) \cdots$. The second and third rules are bijective.

We further discuss how to judge if an emission rule is bijective. We assume that a task contains nested loops and its iteration vector is an affine function of program parameters and loop indices. Let $\vec{\imath}$ be the vector composed of all the problem parameters and loop indices in the lexicographic

order. Thus we can express vectors $\vec{u}$ and $\vec{v}$ as:

$$\vec{u} = D_1\vec{\imath} + \vec{k_1} \quad \text{and} \quad \vec{v} = D_2\vec{\imath} + \vec{k_2}.$$

where $\vec{k_1}$ and $\vec{k_2}$ are constant integer vectors. $D1$ and $D2$ are constant integer matrices. Notice that if there are equalities in P, we should substitute variables to remove some variables before constructing D_1 and D_2.

To find if this rule is bijective, we find E_1 and $\vec{w_1}$ such that $\vec{\imath} = E_1(\vec{u} - \vec{k_1}) + \vec{w_1}$. There could exist many E_1 and $\vec{w_1}$ that satisfy this equation. We impose a condition that $\vec{w_1}$ is composed of the indices that do not appear in $\vec{u}$. Formally, $w_{1_j} = \delta_j i_j$, where the w_{1_j} (resp. i_j) is jth element of $\vec{w_1}$ (resp. $\vec{\imath}$); $\delta_j = 0$ if i_j appears in $\vec{u}$, and $\delta_j = 1$ if i_j does not appear in $\vec{u}$.

Then we have: $\vec{v} = D_2E_1(\vec{u} - \vec{k_1}) + D_2\vec{w_1} + \vec{k_2}$. Similarly we can build E_2 and $\vec{w_2}$ such that: $\vec{u} = D_1E_2(\vec{v} - \vec{k_2}) + D_1\vec{w_2} + \vec{k_1}$.

The above rule is bijective if and only if $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant. This condition ensures that for each instance of $\vec{u}$ there is only one instance of $\vec{v}$ and for each instance of $\vec{v}$ there is only one instance of $\vec{u}$.

In Figure 2, the fact that the first rule is not bijective can be verified as follows. Since $\vec{u} = (k)$, $\vec{v} = \begin{pmatrix} k \\ j \end{pmatrix}$, and the vector of the program parameters and all the loops indices is $\vec{\imath} = (n\ k\ l\ j\ i)^T$, $D_1 = (0, 1, 0, 0, 0)$ and

$$D_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

We deduce that: $E_1 = (0\ 1\ 0\ 0\ 0)^T$ and $\vec{w_1} = (n\ 0\ l\ j\ i)^T$ so that $\vec{\imath} = E_1\vec{u} + \vec{w_1}$. Then, $D_2\vec{w_1} = \begin{pmatrix} 0 \\ j \end{pmatrix}$, which is not a constant.

On the other hand, the second emission rule in Figure 2 is bijective. In this case, $\vec{u} = \begin{pmatrix} k \\ j \end{pmatrix}$ and $\vec{v} = (k+1)$. According to the predicate $j = k+1$, we substitute all the occurrences of j by $k+1$. Hence we have: $\vec{u} = \begin{pmatrix} k \\ k+1 \end{pmatrix}$. Thus, $D_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ and $D_2 = (0\ 1\ 0\ 0\ 0)$. We deduce that: $\vec{w_1} = \vec{w_2} = (n\ 0\ l\ j\ i)^T$. Therefore, both $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are constant.

3. Overview of the SLC Method

The optimization goal of our symbolic clustering algorithm is the same as that of static scheduling algorithms [21, 25]: 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.

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

1. Given a PTG, we first simplify this graph by merging rules, when possible, in order to reduce the number of rules. Then we extract and sort all bijective rules in the PTG. Then, we perform a linear clustering on these bijection rules, The clustering process is discussed in details in Section 4. The important aspect of this process is that merged clusters are always linear whatever parameter values are, namely, no independent task are placed in the same cluster. In this way, parallelism is preserved while unnecessary communication is eliminated.

   Clustering is conducted by assigning two end tasks of a dependence edge to the same cluster (so communication between these two tasks is considered zero). This process is called *edge zeroing*. Since each emission rule represents a set of dependence edges, we will call this process as "rule zeroing".

2. Given a linear cluster of a PTG, the second step of SLC is to provide the identification of each cluster, which is a mapping function from a task ID to a cluster number. For example, mapping function $\kappa(T1, (3, 7))$ is the cluster ID of task $T1(3, 7)$. This procedure is trivial if clustering is not done symbolically. In our setting, this mapping function allows us to map a cluster symbolically to a physical processor.

3. The third step is to derive the mapping and packaging of data items used during execution. This is done using communication rules. When a rule is not zeroed this means that data will be sent from a processor to another. Rules with cluster mapping describe which kind of data (scalar, vector, matrix,...) should be sent. We use these informations for generating message packaging code.

4. The last part is the execution of symbolic clusters assigned to each processor. At runtime, symbolic clusters are mapped to the given number of processors (p) evenly using a cyclic (cluster ID mod p) or block mapping formula (cluster ID /p).

   Task execution is asynchronous and is driven by message communication. Each processor maintains a ready queue and if all data items needed for a task are available locally, this task becomes ready. For each

processor, we maintain t active threads and each of them can pick up a ready task for execution. The advantage of having multiple threads is that the system can take advantages of SMP nodes if applicable. After a task completes its execution, data items are sent following the emission rules related to this task.

There is an important issue on how to initiate execution. At the beginning of execution, each processor needs to find starting tasks (i.e. tasks without parents) and put them on its ready task queue. This is done by computing all valid instances of each task assigned to this processor, called $valid(Ta)$. Then this processor computes all the instances of Ta that need to receive data following the reception rules (call it $recv(Ta)$). The difference $valid(Ta) - recv(Ta)$ is a polyhedron that describes all starting task instances of Ta assigned to this processor.

In the next two sections, we will discuss in details how tasks can be clustered and mapped symbolically.

4. Symbolic Linear Clustering

In this step, we allocate tasks to an unbounded number of virtual processors and in the literature this is often call clustering. All the tasks assigned to the same cluster will be executed on the same physical processor.

A cluster is said **linear** if all its task form a path in the instantiated task graph of a PTG for given program parameters. The motivation for linear clustering is based on a study by Yang and Gerasoulis [13]. They have proven that a linear clustering provides good performance on an unbounded number of processors for coarse grain graphs. If g is the granularity a task graph G, PT_{lc} is the parallel time of any linear clustering, and PT_{opt} is the parallel time of an optimal clustering applied to G, then $PT_{lc} \leq (1 + 1/g)PT_{opt}$. Thus as long as g is not too small, linear clustering produces a schedule competitive to the optimum when there are a sufficient number of processors.

Our clustering algorithm contains two parts: 1) merge rules together if possible to reduce the searching space for clustering. 2) find bijective clusters and cluster those rules linearly.

Rule merging. Before clustering the given PTG, we merge a few emission rules if possible. Two emission rules are mergable if each rule describes the same set of dependence edges and their data items communicated can be combined. For example, let us consider the two following rules:

$$R1 : T1(k) \longrightarrow T2(k+1) : A(k) | 1 \leq k \leq n$$

and

$$R2 : T1(k) \longrightarrow T2(k+1) : A(i) | 1 \leq k \leq n, k+1 \leq i \leq n.$$

```
Input: R a set of communication rules
Output: The set Z of zeroed rules
Z := ∅
merge_rules(R)
bijection_rule_set:=compute_bijection_rules(R)
sort_rules(bijection_rule_set)
foreach r in bijection_rule_set do
   if not_in_conflict(r,Z) then
      Z+=r
   endif
enddo
```

Figure 3. The rule clustering algorithm.

$R1$ and $R2$ describe the same set of edges. We see that rule $R1$ sends element $A(k)$ and rule $R2$ sends elements of vector A from $k+1$ to n. Hence, these two rules can be merged in a rule R that sends elements of vector A from k to n:

$$R : T1(k) \longrightarrow T2(k+1) : A(i) | 1 \leq k \leq n, k \leq i \leq n.$$

Merging of rules is important because it reduces the number of rules. In that way, the clustering algorithm discussed below can spend less time in searching and zeroing bijective rules.

Rule zeroing. We give a formal definition of rule zeroing as follows. Given a rule with form $Ta(\vec{u}) \longrightarrow Tb(\vec{v}) : D(\vec{y})|P$, it is *zeroed* if and only if $\kappa(Ta, \vec{u}) = \kappa(Tb, \vec{v})$ for all the valid instances of $\vec{u}$ and $\vec{v}$ in P. The zeroing algorithm is summarized in Figure 3.

The zeroing algorithm extracts all bijection rules, using the method described in Section 2. Then it sorts all bijective emission rules. The sorting is done such that rules implying more communication are to be zeroed first. The rule ordering is done by taking into consideration the dimension of data communicated (i.e. a scalar, a row or a column, a matrix block). Once bijective emission rules are sorted by a decreasing dimension of data communication, *transitive* rules (as defined below) are placed at the end of this sorted list.

An emission rule R is called *transitive* if it sends data from task $T1$ to $T3$ and there exist two rules $R1$ and $R2$ such that $R1$ sends data from $T1$ to $T2$ and $R2$ sends data from $T2$ to $T3$ (see Figure 4). As shown in

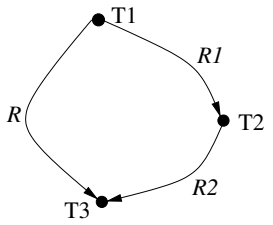

Figure 4. R: a Transitive Rule

next section, it appears that a clustering decision based on R may be in conflict with that for $R1$ and $R2$. But the clustering decisions for $R2$ and $R1$ do not conflict with each other.

Since our goal is to zero as many rules as possible, we rank the zeroing proiority of rules $R1$ and $R2$ higher than that for rule R and thus we place transitive rules at the end of the sorted rule list.

During the zeroing loop shown in Figure 3, the algorithm checks if zeroing a selected rule is not in conflict with a rule already zeroed. Two rules are in conflict if, once zeroed, the produced clustering is not linear. If the rule is not in conflict with any other zeroed rules then the algorithm adds this newly-zeroed rule to the set of zeroed rules. The algorithm repeats this process until all rules have been examined once in the sorted order. We have decribded a method in [4] on how to determine if two rules are in conflict.

We can prove the following theorem to show the correctness of our rule zeroing algorithm.

Theorem 1 *If bijective emission rules of a PTG are zeroed using the SLC zeroing algorithm, then the result of clustering is linear.*

Sketch of proof: When a bijective rule is zeroed, exactly one edge for each instance of the rule is zeroed in the instantiated task graph.

Since SLC ensures that there are no conflict in zeroing multiple bijective rules. Therefore, zeroing those rules produces a clustering with no independent task instances assigned to the same cluster. ∎

5 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 [21, 25], producing such a number is easy since clusters can be explicitly numbered.

Given a set of zeroed rules, we show how to explicitly build a symbolic function $\kappa(Ta, \vec{u})$ such that for any generic task Ta and any valid instance $\vec{u}$, this function gives a cluster number for this task instance. We call this procedure as *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. More formally, let us consider a zeroed rule:

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

Under constrain P_1, we must have:

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

We briefly discuss our method using an example as follows. The basis of our method is the same as the one used by Feautrier in [10] for automatic distribution. We assume that our clustering function is affine with respect to program parameters and the iteration vector. Thus we let

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

where $\vec{\alpha_a}$ and γ_a are vectors of unknowns which have to be found, β_a is a constant and $\vec{p}$ is the vector of the program parameters. We also have

$$\kappa(Tb, \vec{v}) = \vec{\alpha_b}\vec{u} + \beta_b + \vec{\gamma_b}\vec{p}.$$

Then solving Equation 1 leads to $\vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p} = \vec{\alpha_b}\vec{v} + \beta_b + \vec{\gamma_b}\vec{p}$.

For example, let us look at the second rule in Figure 2: $T2(k, j) \longrightarrow T1(k+1) : A(i, k+1)|1 \leq k \leq n-2, j = k+1, k+1 \leq i \leq n$. By zeroing this rule, we have $\kappa(T2, (k, j)) = \alpha_{2,1}k + \alpha_{2,2}j + \beta_2 + \gamma_{2,1}n$ and $\kappa(T1, (k+1)) = \alpha_{1,1}(k+1) + \beta_1 + \gamma_{1,1}n$. This leads to

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

Equation 2 is satisfied with $\alpha_{2,1} + \alpha_{2,2} = \alpha_{1,1}$, $\beta_2 + \alpha_{2,2} = \beta_1 + \alpha_{1,1}$ and $\gamma_{2,1} = \gamma_{1,1}$.

For emission Rule 3 in Figure 2: ($T2(k, j) \longrightarrow T2(k+1, j) : \{A(i, j)|1 \leq k \leq n-2, k+2 \leq j \leq n+1, k+1 \leq i \leq n\}$), we have:

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

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

The above analysis for the second and third rules of Figure 2 leads to the following linear system:

$$\begin{cases} \alpha_{2,1} + \alpha_{2,2} - \alpha_{1,1} = 0 \\ \beta_2 + \alpha_{2,2} - \beta_1 - \alpha_{1,1} = 0 \\ \gamma_{2,1} = \gamma_{1,1}\alpha_{2,1} = 0 \end{cases} \tag{4}$$

System 4 has many solutions. In particular,

$$\begin{cases} \alpha_{2,2} = \alpha_{1,1} = 1 \\ \beta_1 = \beta_2 = \alpha_{2,1} = \gamma_{1,1} = \gamma_{2,1} = 0 \end{cases} \tag{5}$$

Thus, one explicit clustering function κ for the Gaussian Elimination PTG is:

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

and

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

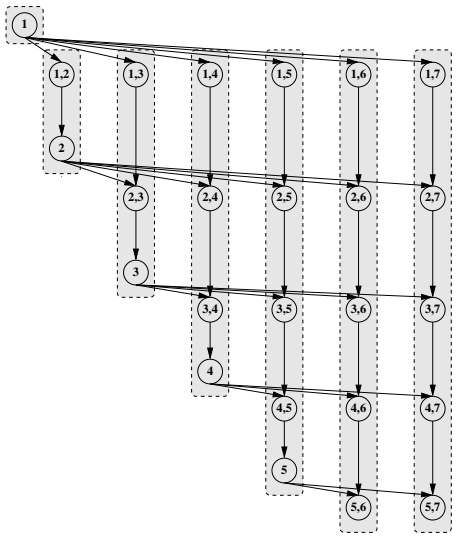

Figure 5. A clustering for an instantiated Gaussian Elimination graph with $n = 6$.

This clustering is depicted in Figure 5.

The above example looks simple; however there is a complication we need to handle. The mapping derivation from two rules for the same task may lead to a restrictive condition on the $\vec{\alpha}$ vector value in computing the cluster affine function of this task. For instance, given the following two zeroed rules :

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

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

Zeroing rule R1 leads to $\alpha_{1,1} = 0$ but zeroing rule R2 leads to $\alpha_{1,2} = 0$. We do not want the $\vec{\alpha}$ solution vector to be zero for all the tasks because this solution maps all task instances into one cluster, which yields no parallelism. In the above case, we know that zeroing these two rules are not in conflict in terms of the linear clustering constraint. To fix this problem, we can use the two mapping functions for task $T1$ with two disjoint polyhedra:

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

In general, we have proposed a technique called graph splitting which uses multiple mapping functions for each selected task if the derived solution is $\vec{\alpha} = \vec{0}$ for all tasks. Given a zeroing of a PTG, we construct a dependence graph G only based on all zeroed bijective rules. We compute the cluster mapping function for each task based on zeroing results. If for all the tasks Ta the mapping solution is $\vec{\alpha} = 0$, we identify the rules that cause setting, we split the G into a few sub-PTGs by dividing the polyhedra domain of Ta into disjoint parts based on these rules. For the above example, the restriction is caused by two rules among the instances of the same task. Then we apply the cluster numbering procedure recursively to each subgraph. The obtained solution for task T_a is no longer an affine function but a *piecewise affine function*. Notice if a task has multiple out-going edges in G, those rules can also potentially impose a restrictive condition on cluster numbering and we can also split based on these rules. Figure 6 illustrates splitting of a task $T2$ based on the rules "$T2 \rightarrow T3$" and "$T2 \rightarrow T4$". Notice that all rules in G are bijective, thus subgraphs $G1$ and $G2$ deal with disjoint sets of all task instances. For example, task instances of $T2$ are divided into two parts, one in $G1$ and another in $G2$. Then task instances of $T1$ are also divided in two parts accordingly due to bijection and rule "$T1 \rightarrow T5$" will only need to appear in $G2$.

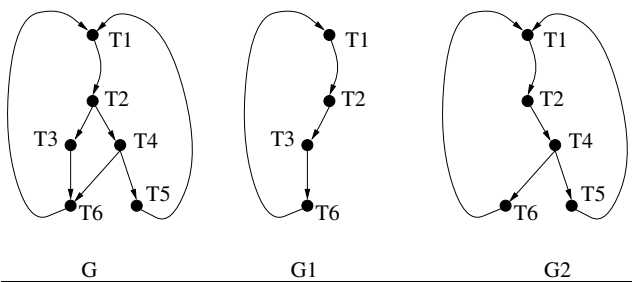

Figure 6. Spliting of Graph G into G_1 and G_2 at node $T2$ (note that G_2 may be partitioned again at node $T4$)

6. Simulation and Experimental Results

We have implemented the SLC for compile-time symbolic scheduling of a PTG, which automatically finds the clustering function κ for each task. We have also implemented part of runtime support for executing a PTG using the SLC schedule, which allows us to conduct experiments in assessing performance of an SLC symbolic schedule.

The benchmarks used in this paper are several compute-intensive kernels in scientific applications: Gaussian elimination, Cholesky factorization, Givens algorithm, Jordan diagonalization, and matrix multiplication. The advantage of our algorithm is that it requires small memory space in clustering and produces symbolic solutions independent of problem sizes and the number of processors used. In this section, we mainly assess if the scheduling performance of our algorithm is still competitive to a static algorithm while retaining symbolic processing advantages.

Graph	n	Sequential time	No. clusters DSC	Sched. length DSC	No. clusters SLC	Sched. length SLC	R
Gauss	100	1009800	92	364532	102	402572	0.91
Gauss	200	8039600	196	994960	202	1033735	0.96
Gauss	400	64159200	402	2979410	402	2990280	0.98
Jordan	100	1509950	92	495149	101	779170	0.64
Jordan	200	12039900	199	1325682	201	1978317	0.67
Jordan	400	96159800	395	3895200	401	5645048	0.69
Givens	100	3720750	114	896350	101	913707	0.98
Givens	200	39214956	371	2654398	201	2539794	1.05
Givens	400	257418656	1269	6964054	401	7845122	0.89
Cholesky	100	510150	93	241918	101	312250	0.77
Cholesky	200	4040300	212	521662	201	687350	0.76

Table 1. Comparison between SLC and DSC on an unbounded number of processors (relatively slow machine/coarse grain tasks)

A performance comparison of SLC and DSC. Table 1 and Table 2 compare performance of SLC with the DSC static clustering algorithm [25] for parameter size of 100, 200 and 400 (matrix dimension). We have not been able to perform tests for larger matrix sizes because task graphs become too large (over 1 million tasks/edges) and DSC cannot schedule them in our machine. The difference between Table 1 and Table 2 is that in Table 2, the edge cost has been reduced, simulating a fast parallel machine and thus increasing the task granularity. The result of matrix multiplication is not shown because SLC and DSC always find the optimal clustering, which yields no difference. The last column of these two tables is the ratio R of the DSC schedule length over the SLC schedule length. If R is greater than 1, it means that SLC outperforms DSC. The results show that despite that DSC computes a clustering suitable to each graph, R is never lower than 0.64. For coarser graphs R is never lower than 0.83. It appears that SLC outperforms DSC for some graph instances. These results indicate that the SLC algorithm delivers a symbolic scheduling solution with performance highly competitive to the DSC static solution in terms of scheduling quality.

We also have simulated the execution of clusters on a bounded number of processors (from 2 to 64). We have used the PYRROS scheduling tool [24] to merge clusters and simulate execution of tasks. The results are shown in Figure 7. In that case R is between 0.75 and 1.4, thus the scheduling quality of SLC is also good on a bounded number of processors.

Gaussian elimination on IBM SP2. For Gaussian elimination code, we have developed a preliminary version of multithreaded code to execute symbolically mapped clusters using a multithreading package called *PM2* [18]. We have run it on an IBM SP2 with 15 RS6000 processors (the 16-th processor was out of order at the time of the experiments). Figure 8 shows the speedup of the Gaussian elimination code with various matrix sizes. We have a speedup of 11.74 for 15 processors when the matrix size is 4000. The result shows that our Gaussian elimination code scales well.

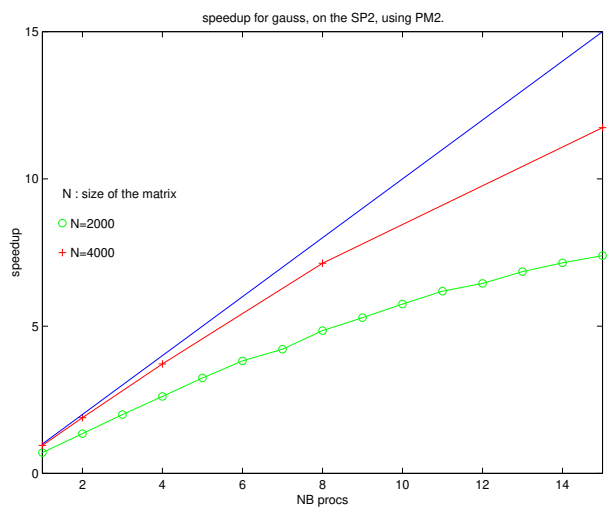

Figure 7. Speedups of the Gaussian Elimination on IBM SP2.

7. Conclusions

In this paper we have presented the SLC algorithm which symbolically schedules parameterized task graphs and can

Graph	n	Sequential time	No. clusters DSC	Sched. length DSC	No. clusters SLC	Sched. length SLC	R
Gauss	100	1009800	101	25093	102	30001	0.84
Gauss	200	8039600	201	100193	202	120001	0.83
Gauss	400	64159200	401	400393	402	480001	0.83
Jordan	100	1509950	101	40002	101	40202	0.99
Jordan	200	12039900	201	160002	201	160402	1.00
Jordan	400	96159800	401	640002	401	640802	1.00
Givens	100	3720750	98	166393	101	175906	0.95
Givens	200	39214956	318	671163	201	755580	0.89
Givens	400	257418656	496	2668550	401	2844880	0.94
Cholesky	100	510150	100	32156	101	30100	1.07
Cholesky	200	4040300	239	121759	201	120200	1.01

Table 2. Comparison between SLC and DSC on an unbounded number of processors (relatively fast machine/fine-grain tasks)

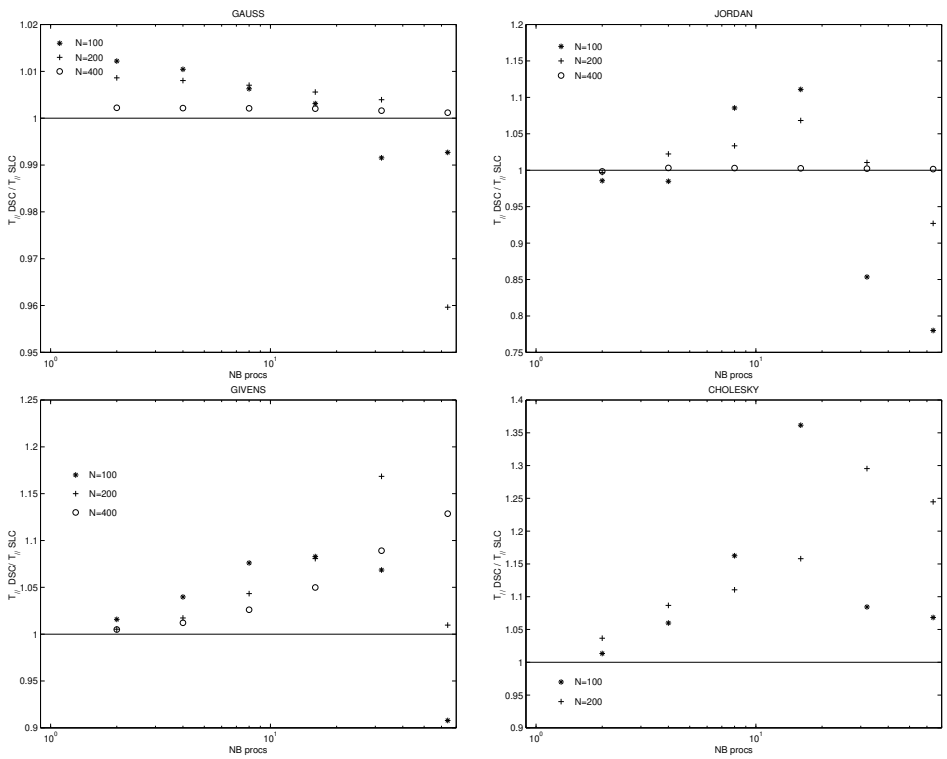

Figure 8. Makespan ratio DSC/SLC for a bounded number of processors (using PYRROS cyclic cluster merging). Cluster execution within each processor is done using the PYRROS RCP* algorithm.

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 the 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 with performance highly competitive to static scheduling algorithms such as DSC.

Currently we are developing a code generator that transforms an annotated sequential program into parallel code using the SLC algorithm and evaluate our method in a wider range of benchmarks.

Acknowledgement. This work is supported in part by the European Community Eurêka EuroTOPS Project, the NSF/CNRS grant 139812, and NSF grant INT-95113361/CCR-9702640.

References

[1] Vikram S. Adve and Mary K. Vernon. A Deterministic Model for Parallel Program Performance Evaluation. (Submited for publication).

[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] M. Cosnard, E. Jeannot, and T. Yang. Symbolic Partitionning and Scheduling of Parameterized Task Graphs. Technical Report RR1998-41, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, September 1998. (`www.ens-lyon.fr/LIP/publis.us.html`).

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

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

[7] 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 *Proceedings of the First International Workshop on Software and Performance*, Santa Fe, USA, October 1998.

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

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

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

[11] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *Proceedings of ACM/IEEE Supercomputing'96*, Pittsburgh, November 1996.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[19] M.A. Palis, J-C. Liou, and D.S.L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

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

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

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

[23] M. Wu and D. Gajsky. Hypertool a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[24] 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.

[25] 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.