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 adapt to the changes in the number of processors or to the changes of the parameter values and 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 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, 24]. 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 processors of the target machine changes, 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 problem 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 to the same processor in order to reduce inter-task communication while still preserving available parallelism. Then it assigns symbolic clusters uniformly to processors. Since each processor may own several symbolic task clusters, our runtime scheme executes clusters on each processor using a multi-threaded executing fashion to overlap computation with communication.

2. Definitions and Notations

2.1. The Parameterized Task Graph

A parametrized task graph (PTG) is a compact model for parallel computation of some compute intensive kernel found in scientific applicatios. It has been introduce in [5] by us in our previous work.

A PTG is constructed from a sequential program decomposed in task. It is mainly composed of communication rules which decribe the data transfered between task. Each

```
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

task is a set of instruction which have to be executed in sequential. Therefore, this model is a coarse grain model.

Communication rules are either reception rules or the emission rules. Reception and emission rules are dual forms of each other. Reception rules describe the set of fathers of a given task. Emission rules describe the set of sons of a given task.

An emission rule R 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. $\vec{y}$ describes which part of the data D is sent. The number of components of $\vec{y}$ depends on the dimension of variable D (0 if D is a scalar, 1 if D is a vector, etc...). P is a polyhedron which described the valid values of each vector: it is a predicate on the variables of $\vec{u}$, $\vec{v}$ and $\vec{y}$. Rule R means that if predicate P is true, task $Ta(\vec{u})$ sends data $D(\vec{y})$ to task $Tb(\vec{v})$.

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.

2.2. PlusPyr

In order to build a PTG we use the PlusPyr tool [16]. PlusPyr is able, given an annotated sequential program, to derive the parameterized task graph. The analysis performed by PlusPyr is based on the work done by Feautrier on integer parametric programming (see [9, 22] for an introduction and [5, 6], for more details).

The programs that PlusPyr is able to analyze must have a static control and only deal with matrix vector or scalar computation.

The program of Figure 1, a straightforward implementation of the Gaussian Elimination, has static control and has two generic tasks $T1(k)$ and $T2(k, j)$. The emission rules of the parameterized task graph, computed by our tool PlusPyr, are shown in Figure 2

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

Rule number 3 means that task T2, for any k and j such that $1 \leq k \leq n-2$ and $k+2 \leq j \leq n+1$, sends to task T2 for $k+1$ and j the column j of matrix A.

2.3. Bijection Rule

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

Tasks are nested into loops, their iteration vectors are affine functions of the program parameters and loop indices. Let $\vec{\imath}$ be the vector composed of all the parameters and the loop indices taken in lexicographic order. Thus, it exists two matrices $D1$ and $D2$ such that:

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

and

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

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

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

The rule is a bijection rule if, for each instance of $\vec{u}$ there is only one instance of $\vec{v}$ and for each instance of $\vec{v}$ there is only one instance of $\vec{u}$. This is true if $D_2\vec{w_1}$ and $D_1\vec{w_2}$ are both constant.

In Figure 2, rule number 1 is not a bijection rule. Indeed, $\vec{u} = (k)$ and $\vec{v} = \begin{pmatrix} k \\ j \end{pmatrix}$. The vector of all the loops indices is $\vec{\imath} = \begin{pmatrix} k \\ l \\ j \\ i \end{pmatrix}$. Therefore $D_1 = (1,0,0,0)$ and $D_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$. We deduce that: $E_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ and $\vec{w_1} = \begin{pmatrix} 0 \\ l \\ j \\ i \end{pmatrix}$ which are such that $\vec{\imath} = E_1\vec{u} + \vec{w_1}$. Thus, $D_2\vec{w_1} = \begin{pmatrix} 0 \\ j \end{pmatrix}$ is not constant.

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

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

3. Overview of the Method

We describe here the steps to schedule a parameterized task graph:

1. We first modify the PTG by merging rules, when possible, in order to reduce their number. Second we extract and sort the bijection rules. Then, we analyze these bijection rules in order to find a linear clustering. Clusters are built using the zeroing process that will be described in Section 4. The main aspect of this part is that the clusters will always be linear whatever the parameter value: no independent task will be put in the same cluster. In this way parallelism is preserved while unnecessary communication is reduced.

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

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

4. The last part of the process is the execution of the clusters. Each cluster is mapped to the processors. The user may choose between cyclic mapping or bloc mapping. Concerning the tasks, each node of the parallel machine manages two queues. The waiting queue, for the tasks waiting for data and the ready queue for tasks ready for execution. Each node executes t threads (t is fixed at the beginning of the execution). Each thread choose a ready task and executes it. When a task is finished the computed data are sent to the son of the task.

4. Symbolic Linear Clustering

We are going first to allocate task to an unbounded number of virtual processor. In the literature they are often call cluster. Therefore all the tasks assign to the same cluster will be executed on the same processor in the parallel machine (with other tasks of other clusters mapped to the same processor). SLC build a function κ that gives the cluster number of the tasks ($\kappa(T1,(3,7))$ is the cluster number of task $T1(3,7)$).

A cluster is said **linear** if all its task form a path in the task graph. Yang and Gerasoulis [13] have proven that a linear clustering provides good schedules on an unbounded number of processors for coarse grain graphs. If g is the granularity a task graph G, PT_{lc} is the parallel of time any linear clustering of G and PT_{opt} is the optimal parallel time of G, then on an unbounded number of processors:

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

4.1. Zeroing Rules

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

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

Input: R a set of communcation rules
Output: The set $\mathcal{Z}$ of zeroed rules
$\mathcal{Z} := \emptyset$
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,$\mathcal{Z}$) **then**
    $\mathcal{Z}$+=r
  endif
enddo

Figure 3. The Zeroing Algorithm

The zeroing algorithm, proceeds as follows. It first tries to merge some rules. Merging of two rules is possible when each rule describes the same set of edges and each data part may be merged in one. Let us consider the two following rules:

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

and

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

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

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

Merging of rules is important because it reduces the number of rules. In that way, the zeroing algorithm is quicker and the lesser the number of rules the smaller the runtime computations messages.

Then, the zeroing algorithm extracts the bijection rules. This extraction is done by implementing the method described earlier in Section 2.3.

Bijection rules are then sorted. The sorting is done such that the rules implying more communication will be zeroed first. The ordering of the rules is done by taking into consideration the dimension of the sent data (i.e. a scalar, a row or a column, a block of matrix). Once the rules are sorted by decreasing dimension of the sent data, transitive rules are put at the end of the list. R is a transitive rule if it sends data from task $T1$ to $T3$ and it exists a rule $R1$ that send data from $T1$ to $T2$ and a rule $R2$ that send data form $T2$ to $T3$ (see Figure 4.). Indeed, as shown in the next Section, it appears that R may be in conflict with $R1$ and $R2$. But $R2$ and $R1$ can never be in conflict. Since we want to zero as many rules as possible we need to consider rule $R1$ and $R2$ before rule R.

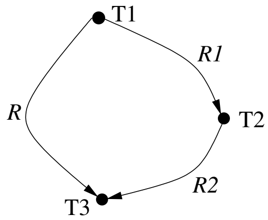

Figure 4. R**: a Transitive Rule**

Then, the zeroing algorithm checks if a given rule is not in conflict with a rule already zeroed. Two rules are in conflict if, once zeroed, the produced clustering is not linear. If the rule is not in conflict with any other zeroed rule then the algorithm adds this rule to the set of zeroed rules. The algorithm repeats this process until all rules have been examined once. We have decribded in [4] how to determine if two rules are in conflict.

Hence, we have the following theorem:

Theorem 1 *Let a PTG where all the communication rules are bijection rules and there is no rule in conflict then zeroing all the rules builds a linear clustering.*

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

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

4.2. Identification

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

Let us consider rule:

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

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

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

The basis of our method is the same as the one used by Feautrier in [10]. As Feautrier we will suppose that $\kappa(Ta, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a}$, where $\vec{\alpha_a}$ is a vector of unknowns which have to be found. We also have $\kappa(Tb, \vec{v}) = \vec{\alpha_b}\vec{u} + \vec{\beta_b}$. Then solving equation 1 leads to $\vec{\alpha_a}\vec{u} + \vec{\beta_a} = \vec{\alpha_b}\vec{v} + \vec{\beta_b}$.

Let us return to our example. Rule 2 is: $T2(k, j) \longrightarrow T1(k+1) : A(i, k+1)|1 \leq k \leq n-2, j = k+1, k+1 \leq$

$i \leq n$. Hence we have $\kappa(T2,(k,j)) = \alpha_{2,1}k + \alpha_{2,2}j + \beta_2$ and $\kappa(T1,(k+1)) = \alpha_{1,1}(k+1) + \beta_1$. This leads to

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

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

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

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

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

This leads to the following system:

$$\begin{cases} \alpha_{2,1} + \alpha_{2,2} - \alpha_{1,1} = 0 \\ \beta_2 + \alpha_{2,2} - \beta_1 - \alpha_{1,1} = 0 \\ \alpha_{2,1} = 0 \end{cases} \quad (4)$$

System 4 has many solutions. In particular this one:

$$\begin{cases} \alpha_{2,2} = \alpha_{1,1} = 1 \\ \beta_1 = \beta_2 = \alpha_{2,1} = 0 \end{cases} \quad (5)$$

Thus, one clustering function κ for the gaussian elimination PTG is:

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

and

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

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

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

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

Rule R1 leads to $\alpha_{1,1} = 0$ but rule R2 leads to $\alpha_{1,2} = 0$. We do not want the solution $\vec{\alpha_a} = 0$ for any task Ta because this solution has no parallelism. However, we know that these two rules are not in conflict, the two functions can be applied to the two disjoint polyhedra, removing the incompatibility:

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

In the general case we shall proceed as follows: We first build G the PTG composed with the rules selected by the zeroing algorithm. Since there is no conflict between the tasks all the edges described by G can be zeroed. However, as in the previous example, multiple edges may induce difficulties (an edge is said multiple between generic tasks Ta and Tb if there is at least one other edge between Ta and Tb). We apply identification process shown above and if the only solution is $\vec{\alpha} = 0$, we split the G into two sub-PTGs G_1 and G_2 and we apply recursively the identification step for G_1 and G_2. Then, the obtained solution is no longer an affine function but a *piecewise affine function*. We first try to split the PTG in order to remove multiple edges. If there is no multiple edges we split the graph at any node with out-degree greater than 1 (see Figure 5).

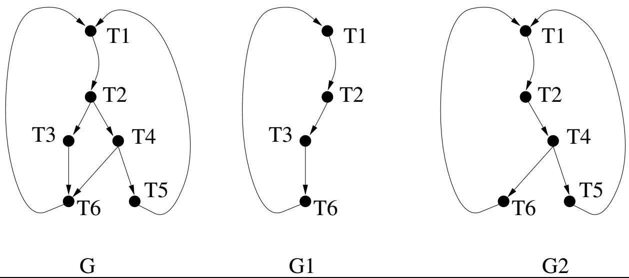

Figure 5. Spliting of Graph G into G_1 and G_2 at node $T2$ (note that G_2 may be splited again at node $T4$)

If no splitting is possible, then we have two solutions:

- A runtime solution consists to allocate the first task of the cluster to the processor corresponding to the value of the first component of the iteration vector. For example, if $T(2,1)$ is the first task of a given cluster for a given execution then all the task of this cluster will be executed on processor 2.
- A compile time solution consists in removing some rules from the PTG and doing again the allocation until a non trivial solution is found.

The first solution executes the clustering found by the zeroing algorithm but incurs many computation at runtime. The second one is simpler but the found allocation implies more communications than the first one because some zeroed rules are removed.

4.3. Finding the Starting Task(s)

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

For any task $Ta(\vec{u})$, using the emission rules, we can compute a polyhedron that describes all the instances of Ta that send data, namely $send(Ta,\vec{u})$. On the other hand,

Graph	n	Sequential length	Nb. clusters DSC	Sched. length DSC	Nb. 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 (slow machine / low granularity)

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

5. Execution of the Clusters

The SLC algorithms described above deals with symbolic parameters and, hence, can be done at compile time.

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

Each time the last data needed for execution arrive to a task this task becomes a ready task and then can be executed by a thread on the processor. We have t threads per node of the parallel machine, thus, at most t tasks can be executed at the same time on a processor.

During the execution it is possible to reduce some remaining communications using a bloc mapping of each cluster c with bloc of size B. Or it is possible to use cyclic mapping which have better load balancing properties.

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

6. Results

SLC has been implemented. We are now able, given a PTG, to automatically find the clustering function κ.

6.1. Comparison SLC - DSC

We have tested SLC on various compute intensive kernels found in scientific applications. These kernels are the following: Gaussian elimination, Cholesky algorithm, Givens algorithm, Jordan diagonalization, Matrix multiply. The code of these kernel, the DAG for a given value of the parameter, and the clustering function is given in the annex. SLC compute a clustering. In Table 1 and Table 2 we have compared the clustering found by SLC with the one found DSC [24] for parameter size of 100, 200, 400. We have not been able to perform test for larger parameter size because the task graphs becomes too large and DSC cannot schedule them.

The difference between Table 1 and Table 2 is that, in Table 2, the edge cost has been reduced, simulating a fast network machine and thus increasing the granularity of the graph. The result of matrix multiply is not shown because, due to the graph structure, SLC and DSC find always the optimal clustering. The last column of these two tables is the ratio R of the DSC schedule length over the SLC schedule length. Thus, R greater than 1 means that SLC outperforms DSC.

Concerning the Cholesky algorithm we need to do two remarks. (1) The Cholesky dag for n=400 is very big (80200 nodes and 10746400 edges, 117 Mo). Thus, even for this size of parameters it has not been possible to perform any experiments. (2) One may think that due to its dag topology it might be better to cluster the dag horizontally

Graph	n	Sequential length	Nb. clusters DSC	Sched. length DSC	Nb. 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 (fast machine / high granularity)

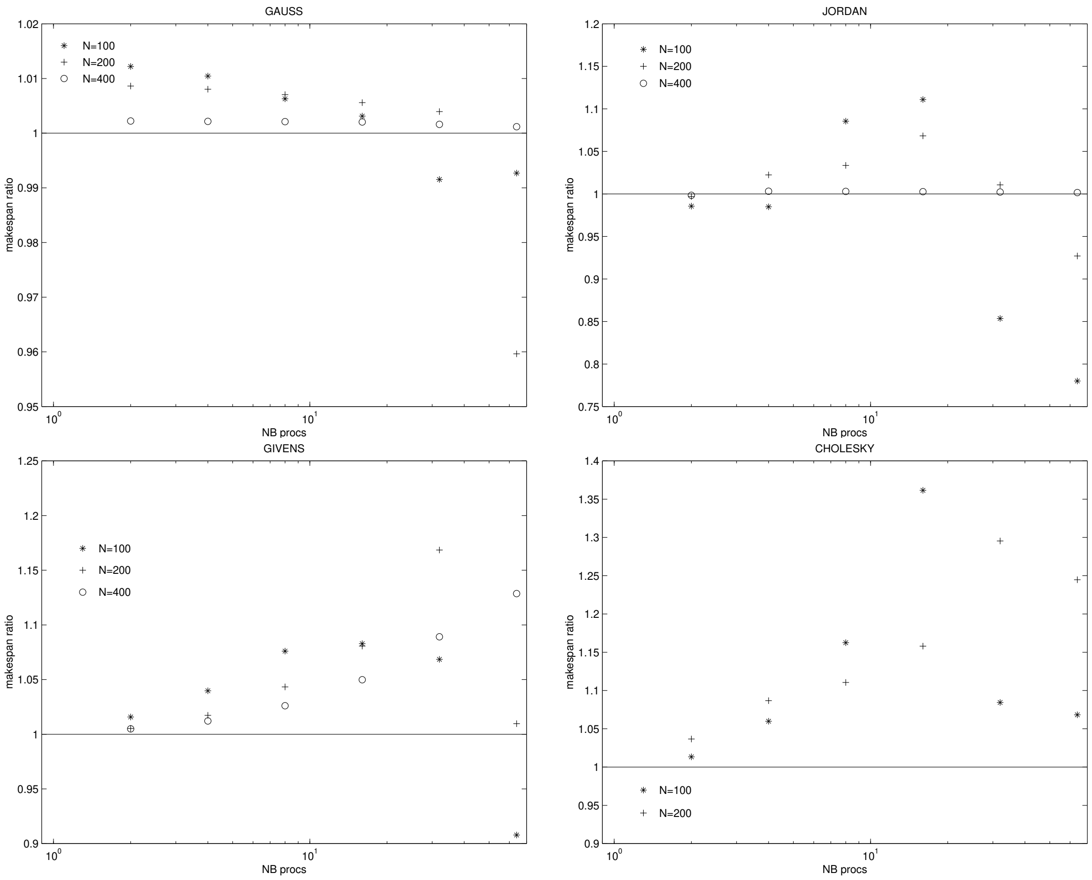

Figure 6. Makespan ratio DSC/SLC for 2-64 processors (cyclic mapping). Merging of clustered is done by the MLS algorithms

($\kappa(T2,(k,i)) = k$). Experiments show this assumption to be false. Indeed, the schedule lengths found for the horizontal clustering are 399469 for n=100 and 1041769 for n=200 with low granularity, and 34951 for n=100 and 139901 for n=200 with high granularity. This figures are all greater than the ones obtained with the SLC clustering. This is due to the fact that vertical edges incur much more computation than horizontal ones. Furthermore, the results shows that, despite DSC computes a clustering suited to each graphs, 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 instances of some graphs.

We also have simulated the execution of the clusters on a bounded number of processors (2 to 64). In order to merge the clusters we have used the modified list scheduling algorithm (MLS) provided by the Pyrros tool [23]. The results are shown in Figure 6. In that case R is between 0.75 and 1.4.

6.2. Parallel Execution of the Gaussian Elimination

We have built a multithreaded parallel program that executes the symbolic static allocation found in the previous Section. We have run it on an IBM SP2 with 32 RS6000 processors. The multithreaded API we use is called *PM2*

Figure 7. Speedup of the Gaussian Elimination

[18]. PM2 (Parallel Multithreaded Machine), allows the user to create threads remotely on a distance node, it can handle a lot of threads on a same node without high overhead. Figure 7 shows the speedup of our program for various size of matrix. We see the high scalability of our method and environment. We have a speedup of x[1] for 32 processors when matrix size is 3000.

7. Conclusion

In this paper we have presented SLC, an algorithm that statically and symbolically schedules parameterized task graphs. This approach is useful when dealing with coarse grain applications when the expanded task graph is too large to fit into memory and when the user does not want to recompute the schedule for each value of the parameters.

Our contribution is threefold. First, we have shown how to find a linear clustering from a PTG. Second, we have modified Feautrier's algorithm for allocating clusters onto the processors in order to take into account conditionals that arise with the PTG model. Our algorithm is then more general and well suited for our problem. Third, we have described a distributed and asynchronous method to schedule and execute our symbolic allocation using a multithreaded method.

SLC have been implemented and tested on several compute-intensive kernels found in scientific applications. Our first experiments are very promising. The results shows that SLC find good schedules and sometime outperforms the well known static algorithm DSC. The multithreaded execution of the gaussian elimination with the clustering found by SLC show our method to be scalable and efficient.

We are now designing a code generator that transforms an annotated sequential program into a generic parallel one and executes the symbolic allocation on a distributed memory parallel computer. In this way we will try show the effectiveness of this method for many more applications.

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.

[1] Michel cela sera etermine qd j'aurai fait l'experience, sinon je remetterai la courbe avec BIP

[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] 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.

[24] 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.