Laboratoire Bordelais de Recherche en Informatique
Université Bordeaux I, 351, cours de la Libération,
33405 Talence CEDEX, France

Research Report `RR-1230-00`

Automatic Code Generation in the Task Graph Model

Emmanuel Jeannot

January 28, 2000

Automatic Code Generation in the Task Graph Model

Emmanuel Jeannot
LaBRI, Université Bordeaux I
France

January 28,2000

Abstract: *The task graph model is a widely used model for performance prediction and scheduling of parallel applications. When used for automatically building parallel program it presents two major drawbacks. First, the size of the task graph depends on the parameter values of the application. Hence, it is not possible to schedule task graph when dealing with large problems. Second, each time parameter values change the task graph change, therefore the scheduling solution and the generated code have to be recomputed. In this paper we present a code generator prototype that uses parameterized task graphs (PTG) as intermediate model. A PTG is a compact and a problem size independent representation of some task graphs found in scientific programs. We show how, with this model, we can generate a parallel program that is scalable (it is able to execute millions of tasks) and generic (it works for all parameter values).*

Keywords: Code generation, semi-automatic parallelization, coarse grain computation, parameterized task graph.

1 Introduction

Writing and debugging parallel programs is a difficult task for software engineers. Compare to sequential programming, parallel programming demand tremendous efforts both for designing algorithms and writing correct programs. In the literature a lot of work has been done in order to help programmers to write parallel programs. Various approach have been taken such as: data parallelism language with directives (HPF [19]), linear algebra libraries (ScaLAPACK [4]), communication libraries (PVM [15], MPI [17], etc ...), compilers for automatic parallelization (for example SUIF [2]) or lastly, task parallelism languages (for example Pyrros [25]). Our work is an intermediate approach between the last two ones. In this paper we show how to automatically generate a program from a task graph representation.

Compilers [3, 12, 14] are, in general, based on a fine grain representation of the program. Most of the time this implies that communication and computation are considered as unitary. This implies also that all the instructions are modeled, which is very precise but rather complex. Our work is based on a coarse grain representation of the program. We do not do any assumption on the duration of communications or on the duration of the tasks. Since there is always fewer tasks than instructions the coarse grain approach is simpler.

A lot of tools have been developed for generating code from a task graph. This is the case of Pyrros [25], Hypertool [24] or CASCH [1]. They all work the same way. (1) they extract a task graph from a program, (2) they schedule the task graph using various scheduling algorithm, (3) they built a program that executes the founded schedule. This approach has two major drawbacks. First, since the task graph is built when the parameter values of the program have been instantiated, it is needed to recompute the schedule and regenerate the program each time the parameter values change. Second, since the size of the task graph depends on the parameter values of the program, this method does not work for large parameter values. Indeed, the task graph is too large to be stored in memory (in general these tools are not able to treat task graph larger than a thousand tasks).

The main contribution of this paper is that we show how to overcome these two drawbacks for some program found in scientific applications. We have designed a code generator prototype based on an intermediate model called the parameterized task graph (PTG). This model is a symbolic representation of DAG parallelism and it requires a small a small amount of memory to be stored because it is independent of program size. Thus, we can generate a parallel program that works for all parameter values and that is able to execute millions of tasks. Our run time system executes tasks in a multithreaded fashion for computation/communication overlapping and time sharing execution.

This paper is organized as follows: Section 2 describes the background and the definitions. Section 3 gives an overview on how works the parallel program. Section 4 describes the different part of the parallel program. Section 5 presents various optimizations performed by the code generator. Section 6 presents experimental results. Section 7 concludes the paper.

2 Background

The work presented in this paper is the back-end of a complete line of semi-automatic parallelization. For illustrating this paper we use the straightforward implementation of the Gaussian Elimination code of Figure 1 as a running example.

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

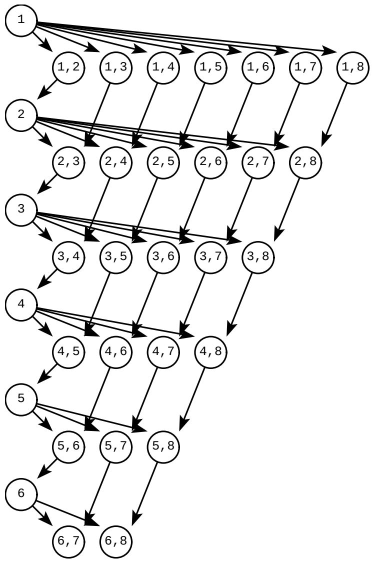

Figure 1: Gaussian Elimination and expanded task graph(n=7)

2.1 The Parameterized Task Graph

A parameterized task graph (PTG) is a compact representation of task graphs that can be found in scientific applications: it is a symbolic model of coarse grain applications. It has been introduced by Cosnard and Loi in [10, 20] to build automatically task graphs. It uses parameters therefore its size is independent of the problem size. Formally it is composed of three parts:

1. A set of communication rules,
2. A set of generic task,
3. A cost function of each task.

A generic task is defined by its name, its code and its iteration vector. In our examples the two generic tasks of the Gaussian Elimination are enclosed by keywords `task` and `endtask`. The cost function gives, for each instance of a task, the number of operations performed by the task. It is used for determining the duration of the task on a target processor. Communication rules symbolically describe the set of sons or fathers of a generic task. The set of sons of a task is described by emission rules which have the following form:

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

This means that task T_a sends to task T_b data $D(\vec{y})$, where $\vec{u}$ (resp. $\vec{v}$) is the iteration vector of T_a (resp. T_b). P is a predicate that describes the valid instances of $\vec{u}$, $\vec{v}$, $\vec{y}$. The set of fathers of a task is described by reception rules which are the dual form of the emission rules. In this paper we will mostly deal with emission rules but the reader should not forget that reception rule representation always exists.

The three communication rules of our running example are shown in Figure 2. Rule number 1

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

Figure 2: Emission rules for the Gaussian Elimination

is read as follow: task $T_1(k)$ send to tasks $T_2(k,j)$ data $A(i,k)$ where $k \in [1, n-1]$, $j \in [k+1, n+1]$ and $i \in [k+1, n]$. Since for each k there is many values for j this rule is a broadcast. Concerning the data $A(i,k)$, k is fixed and i may take many values, this means that a part of the column k of matrix A is sent.

The communication rules are also helpful for describing the set of tasks that have no predecessor [9] as well as the set of tasks that have no successor. The first set is the set of the starting tasks and is needed to begin the computation. The second set is the set of ending task that is necessary to know when the computation is finished.

2.2 A Line of Semi-Automatic Parallelization

Figure 3 describes the line of semi-automatic parallelization we have designed [18].

It starts from a sequential program written in a FORTRAN-like language annotated with `task` and `endtask` delimiters that partition the program into sequential tasks. The program must have a static control [13]. This means that all the array subscript and loop bounds must be affine functions of the program parameters and enclosing loop indices. For-loops are the only control structure and all the instructions must be assignments. Programs of this type are very common as compute-intensive kernels of scientific applications. The first step of our line is to derive the parameterized

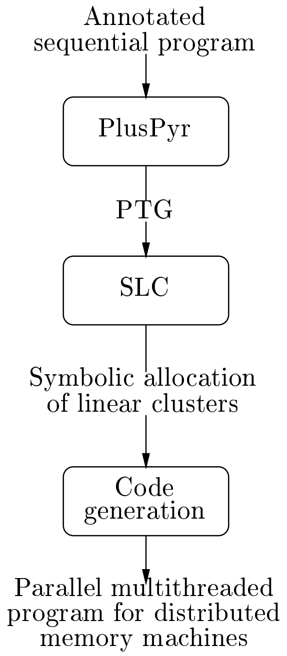

Figure 3: The Line of Semi-Automatic Parallelization

task graph. This part is done with the PlusPyr tool that was developed by Loi in [20]. Originally the PTG was introduced to automate the construction of a task graph (once the parameter values are given). The techniques used for computing the coarse-grained dependences between tasks and the number of operations of each tasks are based on the techniques developed by Feautrier and others on integer parametric programming [11, 13, 23].

In our previous work [7, 6] we show how to use a parameterized task graph to build a program that dynamically allocates and executes the tasks on a parallel machine. The second step of this line consists in finding a symbolic allocation of the tasks. This means that we build a function that tells the cluster where the instance of a task will be executed. Our algorithm [8, 9], called *SLC* (*Symbolic Linear Clustering*) guarantee that the allocation is such that it describes a linear clustering [16] of the tasks. The advantage of building a linear clustering is that it reduces unnecessary communications while preserving parallelism. In our running example the mapping function κ found by SLC is:

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

This means that, for instance task $T_2(2, 5)$ is put in cluster number 5. The last step of the line is the code generation that is described in this paper. Our prototype builds a multi-threaded parallel program which executes the allocation found by SLC.

3 Overview of the Method

Since the source program is decomposed into coarse grain tasks and is modeled by a task graph we are going to generate a program that executes tasks in the macro-dataflow model. This means that each task first, receives data, second executes the code and finally sends data. In order to implement this model of execution we need the data to be accessible only by the task it reads or writes it. In this case, we say that the data are *attached* to a task. The advantage of this approach is that there is no global management of data. On the other hand, this implies that if the same data if read by two tasks we need to duplicate it. We will see how to overcome duplication in the optimization section.

The execution of the tasks is done in a multithreaded fashion. This mean that each task is going to be executed by a thread. Our thread library is called PM2 [21, 5]. It is an efficient API (85 μs for a thread creation and destruction and 8 μs for context switch on a 85 MHz SPARC 5). It uses Light Remote Procedure Call (*LRPC*) for transmitting data between processes: the parameters of the procedure are the data to be transmitted, the name of the procedure is the function that treats receiving data. PM2 is a C API that supports number of communication layers (MPI, BIP, TCP, PVM, VIA, SBP, SCI, ...). This implies that the generated code is very portable. The advantages of the multithreaded approach are the following:

- It allows communication/computation overlapping.
- Using LRPC allows a fully asynchronous transmission of messages. This means that a message can be transmitted anywhere in the code and, contrary to MPI or PVM, it is not necessary to write instructions that test if a message is arrived.

3.1 Message Reception

The algorithm of message reception is given Fig. 4. It handles two sets:

- The set of *waiting tasks* which are the tasks that are have already received messages but are waiting for other one(s).
- The set of *ready tasks* which are the tasks that have received all their messages and are ready to be executed.

for all new arriving message on the process **do**
  T_{recp}=receiving task of the message
  if T_{recp} is in the waiting task set **then**
    Attach the data to T_{recp};
  else
    Create task T_{recp};
    Attach the data to T_{recp};
    Put T_{recp} in the waiting task set;
  if this message is the last one that T_{recp} must receive **then**
    Put T_{recp} in the ready task set;

Figure 4: Messages Reception

Since data are attached to tasks, the incoming data need to be stored in a structure pointed only by the receiving task. We will show in the next section how to generate automatically such a structure.

3.2 Task Execution

The number of threads that are able to execute tasks is fixed at the beginning of the execution. The algorithm they execute is described Fig. 5. When a thread has finished to execute a task it takes one among the ready task. If the ready task set is empty it blocks until a task become ready. Concurrent access to the ready task set is managed with a semaphore. For executing a task, the thread calls a procedure that corresponds to the task code. The parameters of this procedure are taken among the data that are attached to this task.

```
while all ending tasks have not been executed do
   Take T a task in the ready task set;
   Execute T;
   if T is an ending task then
      Decrement the number of ending task to be executed;
   else
      Uses communication rules to send messages to the son of T;
   Remove T and its data from memory;
```

Figure 5: Task Execution

3.3 Sending Messages

The task graph model is such that that only the computed or modified data attached to a task are sent to its son(s). For sending a message a thread scan the emission rules set and determine which one(s) have to be executed. The data are copied to a buffer and sent to the receiving process. Therefore once a message is sent the corresponding data attached to the task can be freed without risk. We will see in the optimization section how to avoid copy of data when possible. Sending of messages is shown Fig. 6:

1 A thread starts to execute task T, while other threads are running in the process.

2 and 2' Execution of task T is over. The thread buffers and sends the data to the sons of T.

3 and 3' The process that is going to execute a son of task T receives the message. It creates a thread that reads the message and attaches the data to the right task.

4 and 4' Once the data is attached the thread is destroyed.

5 The threads choose the nest ready task and executes it.

4 The Parallel Program

The multithreaded program, which is executed on the parallel, machine is always composed of two parts. On part does not depend on the source code and is always the same: it is called the *static* part. The other part depends on the source code and must be generated at compile time.

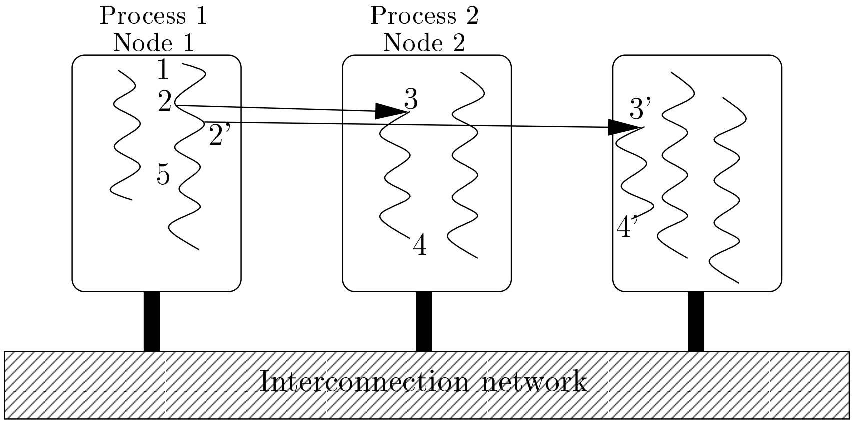

Figure 6: The Different Steps of Asynchronous Messages Transmission

Before generating the code we need to analyze the communication rules in order to derive useful informations about the behavior of the program.

4.1 Communication Rule Analysis

A communication rule is a symbolic representation of some edges in the task graph. Numerous informations are extractable from the rules of a program.

First, we need to determine which data are accessed by a task. In order to do that we normalize expressions. Hence, if a rule describes the transmission of $A(i, j+k+2)$ and an other rule describe the transmission of $A(i, 2+j+k)$ for the same sending and receiving tasks then the data parts are consider to be the same. Once expressions are normalized we analyze the data part of each rule and determine if a dimension is constant. For example if a two dimensional data is transmitted but the first dimension is constant we conclude that a column is transmitted; if no dimensions are constant this means that a full block of the matrix is transmitted.

Second, we need to determine, for each transmitted data, if this data is read and/or written. In the task graph model a data is read by a task if and only if it is received by this task, and a data is written by a task if and only if it is sent by this task. Since reception rules describe the received data and emission rules describe the sent data, determining if a data is written and/or read is not very difficult.

Once this is done we are able to determine the data attached to a generic task. For our running example we are able to determine automatically that task $T_2(k, j)$ reads column k of matrix A and reads and writes column j of matrix A. Hence, the structure describing the attached data to task T_2 is composed of two fields (one for each column).

4.2 Static Part

Sending and receiving messages. The way messages are sent and receive does not vary from an application to an other (what differs is the *content* of messages). The different parts of a message are shown Fig. 7. We use LRPC for message transmission, therefore, sending a message consists in calling the right procedure on the right process. The parameter of this call is the message itself. We have written two receiving procedures: one for broadcasts one for point to point communications. The difference is that a message is sent to one or many processes. Once a reception procedure is

1. Message type: broadcast or point to point;
2. Sending and receiving task (in case of a point to point);
3. List of destination node (in case of a diffusion);
4. Destination node (in case of a point to point);
5. Number of node where the message is sent;
6. Chained list of the sent data. Each element of the list is composed of:
   - The number of element;
   - A pointer to the first element;
   - The communication rule involved by this message;
   - The type of transmission (by pointer/copy);
   - a pointer to the next element.

Figure 7: Different Fields of a Message

called it receives the data and useful informations on how storing it. Since the message describes the reception task the procedure knows on which task the data must be attached. Since the message describes the communication rules involved in this transmission it knows where to store the data in the structure attached to the task.

Ready Tasks and Waiting Tasks Management. The data structure that manages waiting tasks is a hash-table. It is not the most efficient data structure for this type of computation. However, experiments shows that less than 1% of the total computational time is consume in managing tasks in the hash-table. Ready tasks are store in a FIFO list. Concurrent accesses of these two data structures are handle with semaphores.

4.3 Generated Part

The code generation is done after the analysis of the rules and a simple analysis of the code. We describe here the different part of the code that depends on the source program.

Tasks Code. First, we take apart the code between the `task` and `endtask` keyword and we translate it into the C language. We need to subtract 1 to all array access because in C arrays of size n starts from 0 to $n-1$. Second, we "functionalize" this code. Functionalization is the opposite of inlining which is a well-known optimization feature of compilers. This means that we transform our sequential program decomposed into to tasks in a program where each task is replaced by a function call. In our case it is always possible to functionalize the sequential program because: (1) there is no aliases in the array access therefore only one cell of an array is modified when written; (2) the scope of variables is always known: one can determine if a variable is only written or read; (3) there is no `goto` and the only control structure is for-loop. Hence, each task has only one entry and exit point. The analysis of the rules gives, for each task, which are the data that are read and which are the data that are written. The parameters of the task/function are extracted from the data attached to that task. The last step of the Functionalization is to declare the variables that are used in the task but are not described by rules (loop bounds, for instance).

Allocation and Deallocation Task Functions. When a task receive a data for the first time, it must be created. We need to allocate the iteration vector and the structure that represents the data attached to it. We also initialize a counter that describe the number of messages the task still have to receive (when this counter reach zero, the task become ready). When a task is executed all these informations are freed from the memory. Therefore we generate two functions for each generic tasks.

Packing Data. Once a task is executed, the computed data must be sent to its sons. For each rule we generate a code that extract the data described by this rule and copy it into a message structure as described Fig. 7. In order to strictly respect the rule semantic we send exactly the data described by the rule.

Unpacking Data. When a message arrive on a process, the data are to be stored in the right place (in the structure attached to the receiving task). For each reception rule we generate a code that copy the buffer into the attached data.

5 Optimizations

In order the parallel program to be efficient optimizations have appeared to be necessary. Our code generator performs two types of optimizations. The first type concern the speed of the code, the second type concern the memory utilization.

Merging Rules. Some rules describe the transmission of contiguous data. Sometime it is possible to merge these rules into one rule that send the union of the two data. We have designed and implemented an algorithm that check and perform rule fusion, when possible. Hence, the generated code is faster because there is fewer rule to check at run time.

Use of Global Communications. When a message is a broadcast, instead of sending the same message to all the receiving tasks, we send only one message per process that executes receiving tasks. Once a process receives a broadcast message. It copies the data to the tasks that are executed by this process.

Merging Messages. When a task generates two different messages to the same receiving task, it is not necessary to send two messages. Only one send is required. The messages are merged the data are stored in the chain list described field 6 of Fig. 7.

Transmission of Data on the Same Node. As described in the previous section, the packing and the unpacking of data is done even if the receiving task is executed in the same process than the sending task. We have optimized this part in order to avoid the creation of temporary buffers. Data are directly copied from the attached data of the sending task to the attached data of the receiving task.

Pointer Transmission of Data. In the cases of broadcasts, the same data is copied to all the receiving task. This implies an increase of the amount of memory necessary for the execution. We have optimized the reception of broadcasts: when possible, each receiving task points to only one copy of the data. We have added a counter that is initialized to the number of task that points to this data. When a task has finished reading this data the counter is decremented. When it reaches zero the data can be deallocated. This is possible because broadcast data are never written but only read. We have implemented the same kind of feature for point to point communication in order to avoid the copy of data from the task to the sending buffer and from the receiving buffer to the task.

6 Results

We have run the generated program on two platforms: An SP2 with 16 nodes and a cluster of 14 Motorola PowerPC (The POM Pile Of Motorola) linked with BIP [22].

6.1 Speedup Results

Fig. 8 shows speedup results on these platforms for the Gaussian Elimination. The linear clusters of tasks built by SLC are mapped on the physical processor in the cyclic[1]fashion: if κ is the cluster number of the task and P the number of processors then the task is executed on processor $\kappa \bmod P$. The matrix size are between 1000 and 4000. For matrix size 4000 the program executes more than 8 millions of tasks and check more that 16 millions of edges. We obtain a speedup of more than 12.28 on 16 processors for a matrix size of 4000.

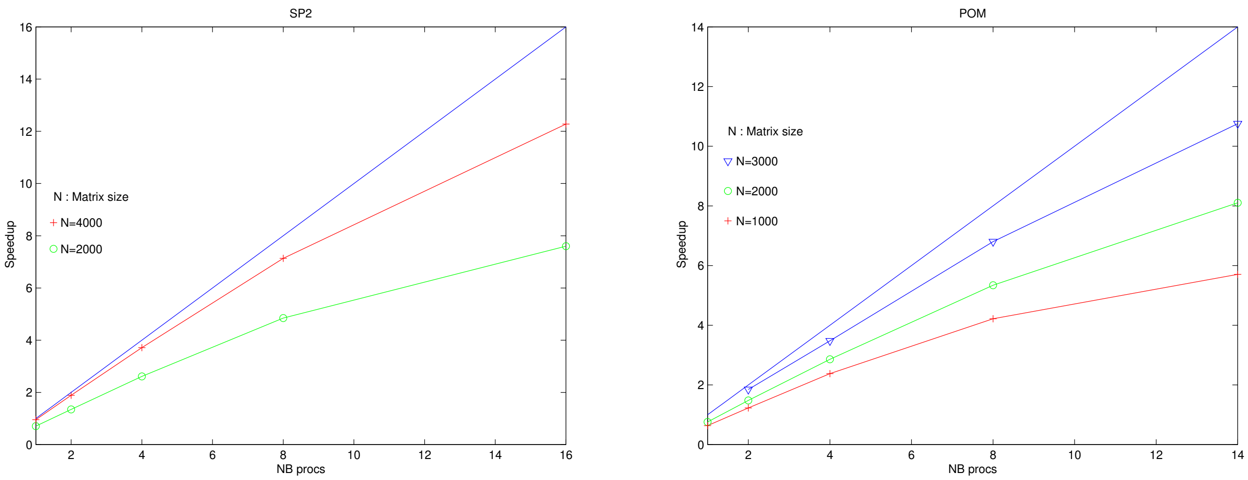

Figure 8: Speedup of the Gaussian Elimination

6.2 Timing of Different Parts of the Program

Fig. 9 shows the proportion of the duration of each part of the Gaussian Elimination program for N=2000 on the POM. Results show that the time proportion in executing tasks decreases as the

[1]The way clusters are mapped on the processors is fixed at the beginning of execution. The user has four choices: *cyclic*, *bloc*, *bloc-cyclic* or *reflect*

number of processors increases. On the other hand the time proportion for selecting ready task increases as the number of processor increases. This is due to the fact that, for a constant problem size, threads have fewer task to execute when processor number increases: they spend more time blocked, waiting for ready tasks.

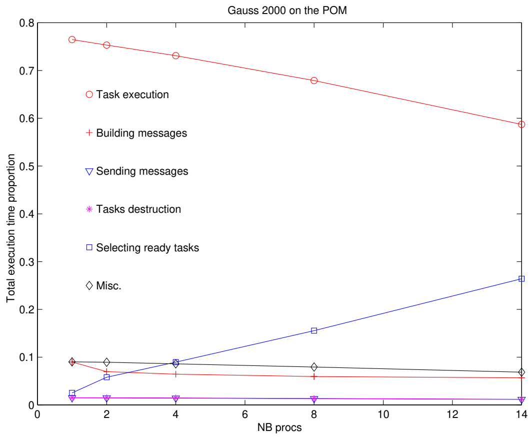

Figure 9: Time Proportion of Different Parts of the Program for the Gaussian Elimination

7 Conclusion

In this paper we have described the back-end of a complete line of semi-automatic parallelization based on coarse grain decomposition of the program. Our contributions are the following:

- We use the parametrized task graph as an intermediate model, hence the generated code works for all parameter values. The analysis of the program is performed once. According to our knowledge this feature is unique in the task graph approach,
- The parallel program executes the symbolic allocation found by SLC, hence this allows to execute very large program. Indeed, (1) the allocation does not depend of the parameter values, (2) computing the processor where to execute a task takes a constant time and memory size. We are able to execute task graph containing millions of tasks and edges, which, according to our knowledge, is not possible with any other tool working on task graphs,
- We have demonstrated that multithreading is well suited for task computation.

Our future works are directed towards the automatic placement of task delimiters. This is the only part that is not automatic in the proposed line. We also would like to extend the input language in order to treat more sophisticated program and extend the class of program that can be automatically parallelized.

References

[1] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu. Automatic Parallelization and Scheduling on Multiprocessors using CASCH. In *ICPP'97*, August 1997.

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

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

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, , and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[5] L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In *1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)*, ENST, Paris, France, October 1998.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] Alain Darte and Frédéric Vivien. Parallelizing Nested Loops with Approximation Distance Vectors: a Survey . *Parallel Processing Letters*, 7(2):133–144, 1997.

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

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

[15] A. Geist, A. Berguelin, W. Jiang J. Dongarra, R. Manchek, and V. Sunderam. *PVM3 User's Guide and Reference Manual*. ORNL, TM-12187 edition, September 1994.

[16] 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.

[17] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994. ISBN 0-262-57104-8.

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

[19] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steel Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

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

[21] 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.

[22] Loc Prylli and Bernard Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Parallel and Distributed Processing, IPPS/SPDP'98*, volume 1388 of *Lecture Notes in Computer Science*, pages 472–485. Springer-Verlag, April 1998.

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

[24] 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.

[25] 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.