AUTOMATIC MULTITHREADED PARALLEL PROGRAM GENERATION FOR MESSAGE PASSING MULTIPROCESSORS USING PARAMETERIZED TASK GRAPHS

EMMANUEL JEANNOT
LORIA, INRIA Lorraine, France
Emmanuel.Jeannot@loria.fr

In this paper we describe a code generator prototype that uses parameterized task graphs (PTGs) as an intermediate model and generates a multithreaded code. 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 million of tasks) and generic (it works for all parameter values).

1 Introduction

In the literature a lot of work exists to help programmers write parallel programs. Various approaches have been taken such as: data parallelism language with directives (HPF [1]); linear algebra libraries (ScaLAPACK [2]); communication libraries (MPI [3]); shared memory programming environments (such as Athapascan-1 [4]); compilers for automatic parallelization (for example SUIF [5]) or lastly, task parallelism languages (for example Pyrros [6]). Our work is an intermediate approach between the last two. In this paper we show how to automatically generate a program when modeled by a task graph.

Parallel compilers [5,7,8] 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. Fine grain analysis implies also that all instructions are modeled. Our work is based on a coarse grain representation of the program: instructions are grouped into tasks. We do not make any assumptions about the duration of communications nor about the duration of the tasks. Furthermore, since there are always fewer tasks than instructions the modelization of a program is less complex.

A lot of tools have been developed for generating code from a task graph. This is the case for Pyrros [6], Hypertool [9] or CASCH [10]. They all work in the same way. (1) they extract a task graph from a program, (2) they schedule the task graph using various scheduling algorithms, (3) they build a program that executes the found schedule. This approach has two major drawbacks. First, the task graph can be built only when the parameters of the input

program are instantiated. Hence, it is required to recompute the schedule and regenerate the program each time the parameter values change. Second, the size of the task graph depends on the parameter values of the program. Hence, this method does not work for large parameter values. Indeed, when the parameter values are too large, the task graph becomes too large to be stored in memory and scheduled (in general these tools are not able to schedule task graphs containing more than a thousand tasks).

The main contribution of this paper is that we show how to overcome these two drawbacks for some programs 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 task graphs. It requires a small amount of memory to be stored because it is independent of parameter values. Thus, we can generate a parallel program that works for all parameter values and that is able to execute million of tasks. Our runtime system executes tasks in a multithreaded fashion for computation/communication overlapping and time sharing execution.

2 Background

2.1 The Parameterized Task Graph

A parameterized task graph (PTG) is a compact representation of some task graphs that can be found in scientific applications. It has been introduced by Cosnard and Loi in [11,12] to build automatically task graphs. It uses parameters therefore its size is independent of the problem size.It is composed of three parts: a set of communication rules, a set of generic tasks and a cost function of each task. In the code, generic tasks 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. Communication rules symbolically describe communications between tasks.

It is out of the scope of this paper to describe the PTG model more precisely. For more details refer to: [11,12,13,14].

2.2 A Line of Semi-Automatic Parallelization

We have designed [15] a line of semi-automatic parallelization. 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 [16]. This means that all the array subscripts and loop bounds must be affine functions of the program parameters and enclosing loop indices. For-loops are the only control structure allowed and all the instructions must be assignments.

The first step of our line is to derive the parameterized task graph. This part is done by the PlusPyr tool [12]. Since PlusPyr is only able to treat static control program, our approach is suitable for regular and static computation only. The second step of the line consists in finding a symbolic allocation of the tasks. This means that we build a function that tells on which processor the instance of a task will be executed. Our algorithm [17,13], called *SLC* (*Symbolic Linear Clustering*) guarantee that, for any parameter values, the allocation is a linear clustering. A linear clustering is a set of disjoint paths of the instantiated task graph. The advantage of building a linear clustering is that it reduces unnecessary communications while preserving parallelism [18]. The last step of the line is the code generation and is described in this paper.

3 Overview of the Method

Execution of the tasks follows the macro-dataflow model: each task first, receives data, second executes the code and finally sends data. In order to avoid global management of data, data that are used by a task are stored within this task.

We use a multithreaded runtime system for executing tasks. This means that each task is going to be executed by a thread. We use a thread library called PM2 [19]. It uses Light Remote Procedure Call (*LRPC*) for transmitting data between processes. Sending messages with LRPC works as follows: when process P needs to transmit data to process P' it calls a procedure on process P'. This procedure is executed by a thread and manages the data transmitted as parameters.

The multithreaded approach allows communication/computation overlapping because the sender is not blocked during data transmission. moreover, using LRPC allows a fully asynchronous transmission of messages. The receiver does not wait for data : a thread is launched each time a new message arrives.

Message reception: when a tasks send a data it calls (with an LRPC) a receiving procedure that stores data within the corresponding task. It manages two sets: A set of *waiting tasks* which are the tasks that have already received messages but are waiting for other one(s). A set of *ready tasks* which are the tasks that have received all their messages and are ready to be executed.

Task execution: the number of threads that execute tasks is fixed at the

beginning of the execution. Each thread takes a ready task execute this task and send the data using communication rules. If the ready task set is empty it blocks until a task becomes ready. Concurrent access to the ready task set is managed with a semaphore. For executing a task, the thread calls the procedure that contains the task code. Parameters of this procedure are the data that have been sent to the task.
Sending messages: for sending a message a thread checks the communication rules set and determine which rule(s) have to be executed. The data are copied to a buffer and sent to the receiving process (We will see in the optimization section how to avoid the copying of data when possible).

4 The Parallel Program

4.1 Communication Rule Analysis

First, we determine if an communication rule describes a point − to − point communication or a broadcast [13]. Second, we analyze communication rules to determine which data are transmitted by a task. Third , we determine, for each transmitted data, if this data is read and/or written. Once this is done we are able to determine the memory location required for storing data within tasks.

4.2 Static Part

There are some parts of the program that do not depend on the input sequential program, such has the receiving procedures that store data within tasks, data structure management procedures, etc...

4.3 Generated Part

The code generation is done after the analysis of the rules and a simple analysis of the source program. We describe here the different parts of the code that depend on the source program.
Task code: first, we "*functionalize*" the tasks. Functionalization is the opposite of inlining, a well-known optimization feature of compilers. This means that we decompose the sequential program in functions. Each function being a task. Parameters of these function are the accessed data of the corresponding task.
Allocation and deallocation task functions: a task is created when it receives data for the first time. When data has finished its execution all these informations are freed.

Packing data: For each rule we generate a code that extracts the data described by this rule within the sending task and build the message.
Unpacking data: when a message arrives, the data is to be stored within the receiving task. For each reception rule we generate a code that copies the input buffer into the attached data.

5 Optimizations

In order to generate an efficient parallel program, optimizations are necessary. Our code generator performs two types of optimizations. The first type concerns the code speed, the second type concerns memory utilization.
Merging rules: some communication 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.
Use of global communications: when a message is a broadcast, we send only one message per node that executes some of the receiving tasks.
Merging messages: when a task generates two different messages for the same receiving task, this messages are merged before sending.
Transmission of data on the same node: We have optimized packing and unpacking of data in order to avoid the creation of temporary buffers. The data is directly copied from the sending task to the receiving task.
Pointer transmission of data: In order to avoid data copy, we have optimized the reception of broadcasts: when possible, each receiving task of the same node points to the same data. When all tasks have finished using this data, it is freed from memory.

6 Results

We have run generated programs on a 16 nodes IBM SP2 and a cluster of 14 Motorola PowerPC (The POM: Pile Of Motorola) linked with BIP [20]. Since we obtain similar performances we only show the results for the POM cluster.

6.1 Speedup Results

Fig. 1 shows speedup results for the POM for the Gaussian Elimination and the Jordan Diagonalization. The linear clusters built by SLC are mapped on the physical processor in a cyclic fashion[a]. The matrix sizes are between 1000 and 4000. The baseline of the speedup is a straightforward C translation of

[a] 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*

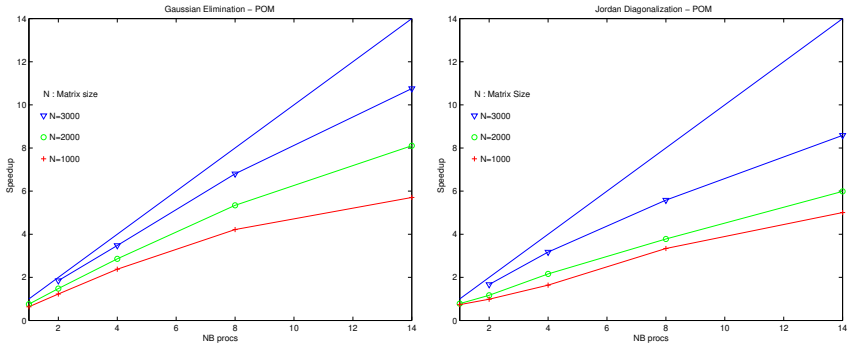

Figure 1. Speedup results for some compute intensive kernels

the original sequential code. For the Gaussian elimination with matrix size 4000 the program executes more than 8 million of tasks and checks more than 16 million of edges. We obtain a speedup of more than 12.28 on 16 processors.

Results for the Jordan Diagonalization are not as good as for the Gaussian Elimination because the Jordan Diagonalization algorithm has more communication.

6.2 Timing Different Parts of the Program

Fig. 2 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 number of processors increases. On the other hand the time proportion for selecting ready task increases as the number of processor increases.

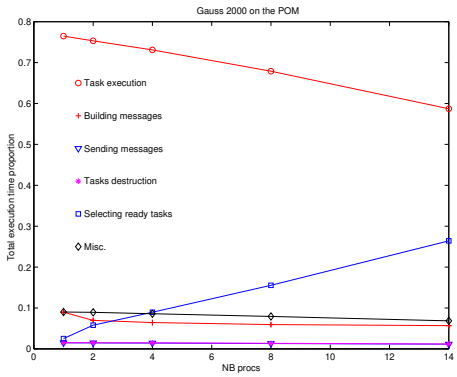

Figure 2. Time Proportion of Different Parts of the Program for the G.E.

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.

6.3 Remark

It has not been possible to compare these results with those obtain by static scheduling tools. Indeed, due to memory constraint static scheduling tools are not able to schedule task graphs larger than a thousand task. However we have shown, in our previous work [13], that SLC obtains similar performances than very good static scheduling algorithm, for small task graphs.

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 contribution is the following. We use the parametrized task graph as an intermediate model, hence the generated code works for all parameter values.

The parallel program executes the symbolic allocation found by SLC, hence this allows to execute a 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 million of tasks and edges. In order to obtain performances, we have described various optimization features. 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. 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.
2. 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.
3. 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.
4. F. Gallilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, October 1998.
5. 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.
6. 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.
7. Alain Darte and Frédéric Vivien. Parallelizing Nested Loops with Approximation Distance Vectors: a Survey . *Parallel Processing Letters*, 7(2):133–144, 1997.
8. P. Feautrier. Toward Automatic Distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
9. 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.
10. I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu. Automatic Parallelization and Scheduling on Multiprocessors using CASCH. In *ICPP'97*, August 1997.
11. M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, 1995.
12. M. Loi. *Construction et exécution de graphe de tâches acycliques à gros grain.* PhD thesis, Ecole Normale Supérieure de Lyon, France, 1996.
13. 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.
14. M. Cosnard and E. Jeannot. Compact DAG Representation and Its Dynamic Scheduling. *Journal of Parallel and Distributed Computing*, 58(3):487–514, September 1999.
15. 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`.
16. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
17. 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.
18. A. Gerasoulis and T. Yang. On the Granularity and Clustering of Direct Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.
19. 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.
20. 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.