Automatic Program Parallelization for Multicore Processors

Jan Kwiatkowski, Radoslaw Iwaszyn
Institute of Applied Informatics, Wrocław University of Technology
50-370 Wrocław, Wybrzeże Wyspiańskiego 27, Poland
fax: (+48)(71) 3211018, tel: (+48)(71) 3203602

Abstract: With the advent of multi-core processors the problem of designing application that efficiently can utilized it's performance stay more and more important. When developing program for the processor with some number of processing units it is not recommended to split it into more processes then the number of available units. It can lead even to decreasing of performance due to frequently context switches of the working processes. The paper deals with the short description of the hardware independent tool that in automatic way parallelized serial programs depending on the number of available processing units by creating the proper number of threads that can be execute in parallel.

1. Introduction

Nowadays multicore processors stay more and more popular for commercial as well as for home usage. It is caused by decreasing cost of it's production and by physical limitations in production of classical processors. It becomes more and more hard to enhance processor speed, therefore multiplying processing units seems to be the best way to achieve larger performance. Developing programs for these processors required from the programmers some specific knowledge about the processor architecture and parallel programming. On the other hand different operating systems provides mechanisms which allows efficiently used more than one processing unit but still the knowledge related with processor architecture is needed. The next problem that should be solved in this case is a problem with the efficient program execution. When the number of program blocks that can be executed in parallel is larger then the number of available processing units, it can happened that it leads even to decreasing the performance due to frequently context switches of the working processes. Therefore the most suitable situation is when the number of program blocks and available processing unit is the same. However it causes that program is not portable be means that during it's execution on the processor with different number of processing units, the efficiency can be worse. On the other hand, there are available tools which helps in program developing, for example SWARM [1], or execution environment as RapidMind [7]. It takes serial program as the input and then executing it in parallel using the "virtual machine". Both of above solutions still require to have some knowledge about the processor architecture and parallel programming. As alternative solution the automatic program parallelization can be used. The paper deals with the short description of the hardware independent tool that in automatic way parallelized serial programs depending on the number of available processing units by creating the proper number of threads that can be executed in parallel.

2. Design overview

Multicore processors gives the opportunity of parallel program execution using the number of available processing units. In common during programming multicore processors, the shared memory paradigm is used. To take advantage of these, it is necessary to develop programs in such way, that there are exactly the same number of tasks (e.g. threads) and available processing units. Typical serial program can utilize only one processing unit, then during it's execution other units are idle. From technical point of view it is easy to change such program behavior, however taken into consideration the "program architecture" it is very tough. Moreover, each situation when threads are created and synchronized can leads to many problems. Developing parallel programs is potentially risky, because of high possibility of making mistakes. Identifying these mistakes is not easy due to program behavior can be different for different it's executions and depends on the way how system dispatcher shared processor time, programmer is not able to predict it. Similarly, much more complicated is the process of identifying and fixing bugs. The reason is similar as above, however this time, it is hard to predict how processor time will be shared between working threads and when context of executing will be switched. On the other hand to take advantage from using multicore processors it is necessary to modify program code in such way, that the most time consuming calculations (for example loops) will be executed in parallel as a separate threads when there are no loop-carried dependence. It is worth to emphasize that dividing program to threads is not the only possibility of it's parallel execution. It is also possible to execute process for each tasks. When doing this it is also necessary to define communication between processes. Choosing the way of parallelization depends on the available hardware platform. Regardless of chosen method it is necessary to divide calculation with respect to each thread. To achieve these algorithms for determining data dependencies and for building dependency graph are used. It leads to choosing proper order of executing instructions to ensure that the result is the same as that from the corresponding serial program.

3. Program Parallelization

The presented tool gives the opportunity of parallel execution of serial programs using multicore processors. In general, automatic parallelization is made in the following three steeps:

- Performing a dependence test to detect potential program parallelism
- Restructuring the program into some blocks which can be executed in parallel. During this step different program transformations are used to obtain the greatest degree of parallelism in a program.
- Generating parallel code for a particular architecture by scheduling program blocks on available processing units and then synthesizing a convenient mechanism for achieving parallelism depending of the used system

It is clear that the last step of above procedure depends on the used system architecture when the first two are hardware independent. To develop the tool that will be hardware independent by means that it can be used at different multicore processors including Cell BE, the following structure of the tool is proposed. The tool

consists of three units. The first is Dependency Analyzer which provides two main functionalities, it performs loop analysis [2,3] and looks for internal parallelism within the program instructions (blocks) [5,6]. The second one is Optimiser that checks the number of available processing units and depending on it's number decided how many program blocks can be executed in parallel. These units are hardware independent. The last unit is Code Generator which is equipped with built in library used during code generation. These solution gives the opportunity to used the tool for different hardware platforms by simple changing the library which consists of functions that depends on the hardware and software platforms.

For the lack of space only the Dependency Analyser will be later described more deeper. One can find two relations related to the program execution: the first that express the dataflow and the second, which express the control flow in the program. Therefore during program parallelization the control dependences have to be identified and "removed", when data dependences will be used for determining which program blocks can be executed in parallel. To avoid the fine grain parallelism which is not suitable for multicore processors with the limited number of processing units in the first step the analyser divides all program instruction onto blocks. Program blocks are disjoint and have exactly one entry and one or more exit points, of course they may have many predecessors and successors and may even be its own successors [4]. For example such program block as "loop block", "if-then block", "if-else block", etc. can be distinguished. It is obvious that program blocks constitute a new program and determining parallelism between program blocks is much more suitable than between program instructions. Using dataflow and control relations the dependence graph for the program divided onto blocks as well as separate graphs for different blocks are created. Then firstly, the possibility of loop parallelization is checked (currently only for "*for loop*") by determining if flow, output and anti dependences exist, later when parallelisation of loop is possible the dependence distance is calculated to determined how many treads can be created, the number of treads used during program execution are determined by the Optimizer. To optimized loop parallelization process four different loop transformation: loop splitting, loop distribution, loop unrolling and loop reversal are performed. In the next tool version more transformation will be taken into consideration and parallelization of other loops will be included. In the second step the parallelism between program blocks is determined. For this aim the created dependence graph and control relation created for the program blocks are used. Depending on the control relation property different program classes have to be distinguished and different ways of control dependency "removal" have been used. When control relation creates a linear sequence of program blocks and all blocks have one exit point only, there are no control dependences and data dependencies specified at the dependence graph are used for determining, which blocks can be executed in parallel. In the case when not all block have only one exit point, due to the possible branches in the program additional conditions related to this property should be fulfill during dependence analysis. In the most general case when the control relation is only transitive and reflexive, the possibilities of existing the local symmetry of the control relation should be taken into consideration.

The tool is implemented in C, as input takes the source code written in C and as output generate the source code of parallel program in C/C++ language. Then the new source is compiled by any available C/C++ compiler, the source code doesn't contain

any instructions from outside of the C++ standard. It means that comparing presented approach with standard program developing process only one additional step, parallelization with the tool is needed. It is not necessary to introduce any other modification in the source code of the program. Even if some instructions used in the program cannot be parallelized they will be leaved without changes. Additionally at this stage there is made the decision about platform, on which program will be executed. If platform has to be changed it is not necessary to modify source code. It is only necessary to parallelized program (using the tool) and compile it (using any compiler as before). Even when operating system is changed it is not necessary to modify source code (assuming there are no direct system commands in the code). Parallelizing is made by executing proper number of threads. It's number depends on the available processing units. Threads shares memory available for the process, so it is not necessary to ensure communication between them in other way than be reading and writing proper memory cells. Threads synchronization is made by instructions added to the code by the tool.

4. Conclusions

The project is in current study, therefore presently the tool covers only a part of the functionalities that normally is supported by the parallelizing compilers. The prototype still misses a lot of features and not all types of instructions can be parallelized, they will be implemented in the further versions of the tool. However experiments performed using the first prototype indicates that the presented tool will be useful for standard multicore processors as well as for Cell BE. The speedup received for programs parallelized by the tool has been close to the number of available processing units. Additionally using the tool, no specific knowledge about the used processors and experience in the parallel programming are required to build the parallel application.

References

1. Bader D., SWARM Framework , www.cc.gatech.edu/~bader/papers/SWARM.html
2. Banerjee U., *Loop Transformations for Restructuring Compilers: The Foundation,* Kluwer Academic Publishers, 1993.
3. Banerjee U., *Loop Parallelization,* Kluwer Academic Publishers, 1994.
4. Kwiatkowski J., *Automatic program Restructuring,* Melecon'91 International Conference, IEEE Catalog No 91CH2964-5, pp. 1041 - 1044
5. Polychronopoulos C.D. *Parallel Programming and Compilers,* Kluwer Academic Publishers, 1988.
6. Randy A., Kennedy K., *Optimizing Compilers for Modern Architectures: A Dependence-Base Approach,* Morgan Kaufmann Pub., Academic Press, 2002
7. RapidMind., *Development Platform*, http://www.rapidmind.com