Specific Problems in Programming Multicore Systems

ANDREEA MARGINEANU and HORIA CIOCARLIE
Computer and Software Engineering Department
“Politehnica” University of Timisoara
V. Pârvan street 2, 300223, Timisoara
ROMANIA
a_margineanu@yahoo.com, horia@cs.upt.ro

Abstract: - In this paper we try to determine which the major problems a developer is confronted with, when building an automatic parallel compiler for legacy code. Most of the code for multicore systems is generated from old, legacy sequential code and the process should be optimized so the time and resources spent adapting it are reduced. The article studies the work done so far and the main solution found in order to automatically parallelize the code. Because the operation is complex and the results are hard to be predicted, the interaction with the user is a necessity. Most of the existing tools for automatic parallelization solved the problem only for particular cases and others are just semi-automatic.

Key-Words: multicore system, automatic parallel compiler, automatic parallelization, parallel programming, code generation, task scheduler

1 Introduction
With the increase usage of multicore systems, a new problem emerged: the parallelization of old, sequential code. The first approach was to manually parallelize the code. But this approach is time consuming and error prone. The programmer had to be specialist in both parallel programming and the domain of the problem. He also has to spend much time understanding the code to be parallelized.

The next approach was to automatically parallelize the code from existing sequential systems. The operation will end in generating parallel code which can be used in multicore systems, from the original sequential one. The most common type of tools is a parallelizing compiler or a pre-processor. The compiler can work in two ways. The first: a fully automatic parallelization compiler which identifies and then parallelizes the suitable sections. The usual targets of the process are the loops (while, for). The compiler also identifies the inhibitors of parallelism like data dependencies and makes an analysis of cost of parallelization and performance obtained. The second way the compiler can work is by programmer directives: the programmer indicates explicitly the parts of the code that will be parallelized, which data is shared and which data is local to each thread. This approach can be used with a degree of automatic parallelization.

The advantages of automatic parallelization are the reduction of time and costs of adapting the sequential code. But the method does not guaranty the correctness of the resulting code and the improvement of the performance. Full automatic parallelization is difficult to be done because of the complex analysis of the program and the unknown factors at compilation.

1.1 Levels of Automatic Parallelization
There are several levels of parallelism that can be done automatically. The first one is the most documented and is being already used in both single core and multicore systems: instruction level parallelism (ILP). This automatic parallelization is based on concurrency analysis which indicates which instructions can be executed in parallel and which have to be executed in order. It is already introduced in compilers to improve the code even for single core architectures, but for multicore architectures this type of parallelization may be improved in order to benefit from the extra processing power and eventual shared cache levels. In a multicore system the split of the code is done on a function base, each instruction being executed on a different processor, the time spent communicating the results can be more then the speed-up obtain by running the instructions in parallel. But the problem of computation/communication is addressed in a global manner, resulting in coarse-grained applications.

The optimization for parallelization of the sequential code can be done by the compiler or by the middleware. If the optimization is done at compiler level, the disadvantage is the runtime information is not available at this step. If the optimization is done in
the middle-ware, it can cause overhead when is processing the runtime information. When the code is analyzed for parallelization, this analysis can be done static or dynamic. Static analysis implies considering code as text and extracting parallelism from how code is written. If the analysis is done dynamic, the code’s behavior when running is analyzed. The last level is the scope of the parallelism. It can have local or global scope. If the scope of parallelization is local, the parallelization will improve the distribution of the functions on the available cores. If the scope is global, the parallelization will optimize the program flow, by distributing the sub-programs to the available cores. Before developing a tool or a compiler to perform automated parallelization, the programmer has to decide in which level the optimization is done, how the code is going to be analyzed and which is the scope of the tool.

2 Current Studies
The parallelization of existing software for using the resources of multicore systems is an on-going topic of research. The main goal of the studies is to parallelize the existing work with less, if not no contribution from a programmer. Most of the existing code was written for single core processors, thus is sequential. With the advance of hardware and the extend usage of multicore processor, this code has to be adapted to benefit of the new resources. First automatic parallel compilers focused on parallelizing the most obvious parallelizable parts of the code: the loops. The compiler has to determine if the loop can be parallelized (there are no data dependencies between the iterations of the loop) and if the parallelization of the code is actually optimizing the performance of the system in terms of execution time. Another approach was to build semi-automatic compilers where the tool benefited from the help of the programmer. The process was helped by adding “hints” for the parallel compiler or by creating an interactive system between the tool and the programmer. OpenMP proposes a C/Fortran compiler with support for parallelization: on the sequential code [2], the programmer will introduce only preprocessing directives. These directives describe the areas of the code which will be parallelized, which data is local for a thread and which is shared, what portion of the code is a critical section etc. without the programmer needing to interfere in the process of creating and managing the threads. pMapper [3] is a tool were the parallelization is realized in the middle-ware (not at compiler time), dynamic and global. The tool parallelizes the processing of large data structures from MATLAB code. After the code was written, the optimizations were made by using maps to specify the characteristics of the used architecture and the data structures that will be parallelized. ROSE [5] is a multi-language source-to-source compiler which automatically parallelizes C++ code and also preserves the high-level abstractions. The compiler determines which loops are good candidates for parallelization and inserts OMP preprocessing directives. The code can be then compiled using the C compiler for OpenMP. OSCAR [6] is a compiler for FORTRAN programs which automatically parallelize code by introducing OMP preprocessor directives. OSCAR proposes static scheduling for tasks with only data dependencies and dynamic scheduling for tasks with data and control dependencies. UNIX flags autopar and loopen can be used to parallelize automatically the loops inside the program [4]. This approach has limitation as only loops with no data dependencies are parallelized. Also, it is not guaranteed that the parallelization is improving the performance of the application. Polaris [7] is a source-to-source automatic parallelization compiler for FORTRAN code. The compiler can automatically parallelize loops and arrays accesses. Cepheus [7] transcriber is an alternative for programs written in C, to be automatically re-written in FORTRAN and then parallelized using Polaris. Cetus [10] is an automatic parallel compiler which started from Polaris and also can reason about programs written in symbolic terms. It transforms C code from sequential to parallel. The tool is specialized in automatic loop parallelization: inspects loops and generates data dependence graphs and performs symbolic range analysis to determine the size of the loops. A study [8] proposes a semi-automatic compiler for Java programs, which instantiate classes on multiple machines and redirects invocations to all instances of the class. This compiler must receive from the programmer the rule to combine the results from all processes. Another study [9] presents problems which appear in case of interaction between parallelization and other optimization tools. For example, tilling optimization of the loops if is applied before parallelization, is decreases the performance of the
Parallel programs are not deterministic and the system generates incorrect results.

3 Difficulties in Automatic Parallelization

Parallel programs are not deterministic and the programmer has to have a deeper understanding of the behavior of the software. The main difficulties in automatic parallelization refer to dependence analysis of code with pointers, recursion or indirect function calls, loops with unknown number of iterations or accesses to global resources with synchronization.

Also, decomposition or mapping the data structures used by the parallel sections of the code and synchronization of the parallel code can be difficult to manage in parallel code.

Before the sequential code can be parallelized, the best candidates for parallelization have to be found. The search always involves data dependence analysis. As a result, it will be known which parts of the code have to be executed in a strict order and which parts have no imposed order, thus can be executed in parallel.

After determining all data dependencies in the code, in case the data dependencies can be solved, the code will be split into tasks which will be executed in parallel. The resulting tasks will have to be assigned to the processors. These assignments have to consider load balancing to improve the time and resources consumed. If the load balancing is not done correctly, the parallelization may introduce delays in the system instead of improving it.

Even if the parallelization was done without introducing errors in the existing code (due to undetected data dependencies), this is not a guaranty that the parallelization actually improved the code. This limitation is first explained in Amdahl’s law: the speed-up depends on the percent of code that was parallelized. Another limitation is introduced by the communication between the cores and the time spent on context switching, if there are more threads running on the same core.

If a portion of the code cannot be executed in parallel due to data dependencies, it may also block other threads, if not all, until it is executed because of the synchronization blocks.

Another impediment in parallelization of existing sequential code represents, paradoxically, the improvements done to the sequential code. These improvements can cause data dependencies in the code that did not exist before (for example, after loop tilling). Also, the best sequential algorithm is usually not the most efficient algorithm to be parallelized. This means that in order to obtain better results, some parts of the code have to be rewritten when the code is parallelized.

4 Benefits of Interactions with the Programmer

Parallelization of old sequential code is not a simple task. For developing a tool which can do the parallelization automatically, the programmer has to consider all the aspects and develop a method to automatically determine and also solve the problem.

In the current studies the majority of the tools limited their automatically parallelization to loops. The tool analyzes the loop to determine if there are data dependencies and if not, splits the iterations and then executes them in parallel. If the tool find data dependencies, or the loop contains other nested loops or indirect method calls, the tool will inform that the loop could not been parallelized and then stops. The major problem of the automated tools is not the creation of the threads but the decision of where to create them.

In case of tools which interact with the programmer, the percent of the code which is parallelized is bigger because the programmer is giving hints in cases where the tool cannot investigate the problem. The tools can be them used to determine which the best candidates for parallelization are: the portions of the code where the parallelization produces the highest performance improvement or the parts of the code which use the most CPU time.

After the tool analyzes the code, the programmer indicates the parts of the code he wants to be parallelized, using for example preprocessing directives or other graphical interfaces. The benefits of this approach are the time saved for the analysis of the code and the transparency of creating and managing processes and tasks.

5 Code Generation

After determining which portions of the code can be parallelized and which not, the compiler has to generate the auxiliary code for the parallel regions.

Code generation is the last of the tasks the compiler has to perform on the code to obtain an executable application. There are several steps to be done before the final operation. Possible steps for parallel code generation are presented in [11].

The first step is to parse the code and extract the parts that will be executed in parallel, this step is done only after the entire code was analyzed and the

The last step, after creating the needed data structures and inserting the operations for creating threads is to generate code that can be executed by the machine. For the same example of a C compiler in [11], the code can be now generated by a standard C compiler.

6Automatic Parallelization Techniques

There are several transformation techniques which are used in some of the automatic parallelization compilers. Most common are: privatization, reduction variable recognition and induction variable substitution. These three techniques were used first to develop Polaris [12] and then to develop another compiler, Cetus [10].

Privatization is an algorithm for identifying variables, scalar or array, private to a loop. The nested loops are traversed from inner loop to outer loop. The algorithm finds the definitions (writes) and uses (reads) for all variables. The uses which are defined in inner loops are private to that loop. The others create read references seen in the outer loop. The algorithm aggregates the selections of variables creating private sections and definition sections for each loop.

Reduction variable recognition is described in [12] and used both in Polaris and Cetus development. The algorithm searches for assignments:

\[ A(a) = A(a) + \text{expr} \]  \hspace{1cm} (1)

Where A is an array and a does not depend on A and can be null (in this case, A becomes scalar).

The algorithm finds two types of reduction: histogram reduction in which one dimension of the assignment is invariant and single address reduction, where all dimensions of the assignment are invariant.

Induction variable substitution algorithm is also described in [12] and used for Polaris and Cetus and is done in two steps. The first step is to determine the induction variables:

\[ \text{iv} = \text{iv} + \text{inc\_expr} \]  \hspace{1cm} (2)

where inc\_expr is an expression which can be formed from outer loop indices, other induction variables or loop invariant terms.

The second step is to compute the close form of the induction variables. The computation algorithm is described in [12].

The transformation techniques will result in determining blocked regions of the code, in which synchronization primitives are inserted. The operation from the synchronization block will become atomic.
The variables which can be privatized are used by each processor. In this case, new variables are defined for each processor and used instead of the privatized variables.

After data dependence analysis, some parts of the code can be expanded. Shared arrays variables are defined in order to collect partial sums from the loops. These sums are later combined in a separate loop after the original one which now can be executed in parallel. Sometimes is hard to determine the size of the partial sums, thus the size of the shared arrays. For this case, symbolic range propagation analysis is used. This analysis determines for example for an integer type variable the symbolic range. All these values are kept in a map and used for code reduction.

7 Task Scheduler for Multicore Systems

One model for parallel programming is task parallelism. The model assumes distributing the tasks in which the application is composed to the different cores of the system, in order to reduce the latency. One of the first approaches was to manually distribute the tasks to processors. With the increasing number of cores in each multicore system, manually exploring the potential parallelism of an application is greatly time-consuming. This problem has been proven to be NP-complete. [1]

The automated scheduling of task to cores in multicore systems is based on the construction of dataflow models. Before splitting the application in tasks, the abstract syntax tree is built. The abstract syntax tree (AST) represents the syntactic structure of the source.

One approach is to define types of macros and then split the source code according to these macros. In [1] the blocks types are: Basic Blocks (BB), Repetition Blocks (RB) and Subroutine block (SB). The blocks are code sequences with only one entry point and one exit point. The macro tasks can be built by adding automatically preprocessor directives, for example the ones used by OpenMP.

After generation the macros, the abstract syntax tree is built and the analysis for data dependencies is used. The result is a list which was constructed by calculating a critical path. After the generation, the list will never be modified. This is the fastest algorithm, but has limitations and is usually used as a starting point for other algorithms.

2. The FAST algorithm – this algorithm is a refinement of the list algorithms. It uses probabilistic hops and it keeps the best latency found. It can be stopped by the user. This algorithm exploits the multicore parallelism of the system.

3. A genetic algorithm – this algorithm has the FAST algorithm as the starting point with the initial population formed by the best N solutions of this algorithm. The genetic algorithm can also be stopped by the user and is also multithreaded.

In case the macro tasks have data and control dependencies, the scheduling for these tasks has to be dynamic. The dynamic scheduling is based on Earliest Execution Condition analysis. This analysis determines the earliest moment when the macro task can begin. In this type of scheduling, the tasks are assigned to cores at runtime to cope with the uncertainties determined by the branches. The dynamic scheduling can be centralized, when one core does the scheduling, or distributed, when all cores take available tasks from a pool of tasks and also update the work pool when then task is completed.

For developing this type of compiler, there are several algorithms to implement: data dependency analysis algorithm, AST generating algorithm, control dependency analysis algorithm and the scheduling algorithm. The correctness and efficiency of the optimized code depend on the choices made in each of these cases.

8 Conclusion

The paper tries to find the most common problems encountered when staring to build a tool for helping develop applications for multicore system. As the multicore systems are more and more used, writing parallel code becomes a necessity and tools for assisting this process are needed.

But not only the new application are designed to use the new resources of the system, legacy code has to be adapted. The task is time and resource consuming and also error prone. The paper also presents some
specific methods for generating automated parallel code from old sequential code.

The result after analyzing the challenges and current studies is that the design of the tool is a compromise between the complexity of the tool used to parallelize and the involvement of the user in making the parallelization decisions.

The automatic parallelization is a complicated process which needs complex algorithms. Although several studies and attempts were made, the results were not as promising as expected. Even if the tested tools managed to parallelize many problems, they did not succeed to avoid totally the involvement of the user in parallelizing the application. Most of the existing tools are limited to loop analysis and parallelization. But even semi-automated parallelization reduces the time spent by the programmer to parallelize the application and also the basic creation and management of threads can transparent for the programmer, so he can focus on understanding the problem and the code and parallelizing it.

References: