Michał GOZDALIK
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY

An automatic parallel OpenMP code generation

Mgr inż. Michał GOZDALIK

A PhD student at the West Pomeranian University of Technology, Szczecin. Currently involved in creating a tool for the generation of the parallel code in C, with the content of OpenMP standard, taking most possible advantage of multi-processor machines.

e-mail: mgozdalik@wi.ps.pl

Abstract

This paper presents a problem of generating an efficient parallel code from an existing sequential code in an automatic way. The main part of this paper is dedicated to the description of the automatic parallel code generation process. Not only an idea of building an automatic code generation tool is provided, but also a theoretical basis which allows us to understand the optimization problem of parallel code. In the theoretical part of the article the solution has been proposed for measuring the quality of code executed by determining the parameters of speedup and efficiency. Also information about known problems associated with parallel processing and speed of code were provided. Discusses, inter alia, impact on the effectiveness and performance of the barrier synchronization. Also a problem with scheduling in the performance of the CPU load of parallel threads is presented. An example of code generated by a tool under development is explained. Some results of experiments are provided to present code quality measurements. The results come from the first iteration of the program, which does not attempt to optimize the generated code in terms of improved locality. Iteration does not include the attempt to generate code that would contain less of a barrier synchronization. These features are under the implementation phase.

Keywords: OpenMP, iterative code generation, shared memory programming.

Automatyczna generacja kodu równoległego w standardzie OpenMP

Streszczenie

W artykule przedstawiony został problem dotyczący stworzenia automatycznego narzędzia generującego kod w standardzie OpenMP, który byłby efektywnie wykonywany pod danym środowiskiem uruchomieniowym. Artykuł przedstawia podstawy teoretyczne związane ze sposobem pomiaru jakości wygenerowanego kodu, jak również przedstawia model narzędzia wykonującego automatyczną generację wydajnego kodu w standardzie OpenMP. W części teoretycznej zaproponowane zostało rozwiązanie problemu pomiaru jakości wykonywanego kodu za pomocą określenia parametrów przyspieszenia i efektywności. Opisany został sposób, w jaki można uzyskać dokładne wartości tych parametrów podczas wykonywania aplikacji równoległych. Zawarto również informacje na temat znanych problemów związanych z przetwarzaniem równoległym i szybkością działania kodu. Omówiono między innymi wpływ synchronizacji barierowej na efektywność wykonywanych programów. Przedstawiono także problem równomiernego obciążenia procesorów podczas wykonywania wątków programu równoległego. Oprócz architektury narzędzia, zaprezentowane zostały wyniki badań uzyskane z częściowo zaimplementowanej już aplikacji. Wyniki pochodzą z pierwszej iteracji działania programu, która nie podejmuje próby optymalizacji wygenerowanego kodu pod względem zwiększenia lokalności. Iteracja ta nie zawiera również próby wygenerowania kodu, który zawierał by mniej synchronizacji barierowych. Powyższe funkcjonalności są w fazie implementacji.

Słowa kluczowe: OpenMP, programowanie równoległe, automatyczna generacja kodu.

1. Introduction

Through many years computer's architecture evolves. Faster processors provide ability to execute more instructions per second. The need for fast CPUs were caused by graphics programs and by software dedicated for scientific purposes. Unfortunately, further evolution of one-core processors is bounded by technical problems, so multi-core processors became popular in last years. To use all abilities of new multi-core processors, new tools are needed. Programs which were working under a one-core processor can not use more cores to enhance performance. Such programs must be rewritten. It is not as simple as might be thought, so automatic tools are needed for generating parallel code from sequential one. It will take too much time to rewrite all code libraries which were written in a sequential way.

Nowadays all large companies doing researches about parallel processing, but not many automatic code generators are provided and those which exist are under a construction phase. Two of them are interesting to be mentioned.

Fujitsu Parallelnavi Workbench

It is a workbench created for using in WAN networks which allows programmers to develop projects in the OpenMP standard in an easy way. The most interesting subsystem in this workbench is an automatic code generator module. It provides ability to generate, analyze, and correct code in the OpenMP standard. Tuning the execution time function bases on program profiling. This module is under a development stage and as in the VTune performance analyzer, the code generator is in the beginning phase of functionality. More piece of information about this benchmark is provided in paper [4].

iPat/OMP

It is also an automatic tool designed for code generation in the C language under the OpenMP 3.0 standard. The capabilities of this program are quite broad. One of four modules can analyze parallelism to provide information for the code generator how parallel directives should be placed inside code. Having such type of information, another module generates OpenMP directives which are placed inside a source code. The other two modules try to enhance performance. A program restructuring module was design to enhance parallelism and execution effectiveness. To provide measurements, OpenMP standard functions were used and the execution time analysis module is in charge to provide information about changing code to be more efficient. More piece of information about this tool is provided in publication [10].

The above solutions provide some techniques for optimizing the time execution of generated code. In the other sections of this paper it is stated that an optimization process is more efficient when it is done under a specific execution environment. For that reason, a parallel code generator should optimize a generated code under a specific machine under which code will be executed. The execution of such a program on other machine will not be as optimal as under the machine where optimization process took place. For that reason changing an execution environment will cause running an optimization process once again if higher efficiency is needed.

2. Parallel code generator

The automation of a parallel code generating process is not as simple as might be seen in the first time. The problem is to generate an efficient code which will use all of available machine resources in the best possible way. It requires to generate code which will execute as fast as possible under a provided machine. There is no easy way to achieve such a result and in some cases it is impossible. It is reasonably obvious when we understand the fact that computer's architecture differ from each other. Different processors, memories and technology solutions for a computing machines ensure variety which provide to one unsolved problem - how to generate a parallel code which will execute fast in all kinds of computers. As a result, a described code generator will generate an efficient code dedicated for a specific machine. If there will be a need for generating parallel code under other computer architecture machine, the code generation process will be done once again under that specific machine. Such a method will guarantee that the code is optimal under one specific machine and probably will not be efficient on other architecture.

Talking about the code efficiency and the optimization of code, we need to keep under our consideration the fact that it should exist a methodology which can measure the code efficiency. In the next part of this paper, more piece of information about measurements methods of code quality is described.

3. Parameters and code quality

There are two main measurements in a parallel processing paradigm which can easily supply information about the code quality. These parameters are crucial for a program which will allow us to automatically generate code in the OpenMP standard, because of information about the code quality. For better understanding the problem of generating efficient code in OpenMP, it is highly recommended to know how to measure code quality. That is why this piece of information is highly valuable during a parallel code generation process. In the rest of this section more piece of information about speedup and efficiency parameters is provided.

The speedup is a measure which can tell how faster the parallel code is executed than the same code is executed on a single processor machine. The most precise way to measure speedup is to count processor's ticks. Not only for that reason speedup is not measured in seconds, but also communication time is a problem. It is impossible to measure execution time in seconds because these measurements will be inadequate according to a problem with communication time. It is more reasonable to count time in ticks which are provided directly from a processor and counts only the instructions which were executed by a program. As a result, no communication time is included into measurements, so the result is more precise.

To count speedup, the formula provided below is commonly used.

$$S(n,p) = \frac{T(n,1)}{T(n,p)} \tag{1}$$

In the formula above, T is the time counted in processor's clock ticks, n is the portion of code, and p is the number of processors used for computation. S is commonly reserved as a variable which describes speedup. The case when speedup is equal to the number of processors used in a program execution is commonly named *full speedup*. When speedup is higher than the number of processors used for a program execution, the term *hipper speedup* is used.

Speedup is not the only parameter which can provide a piece of information about the quality of parallel code. The second one is efficiency. This measure is strictly connected with speedup. In the matter of fact, it is speedup divided by the number of processors used for program execution. The efficiency formula (2) is provided:

$$\delta_A(n,p) = \frac{S(n,p)}{p} \tag{2}$$

where $\delta A(n, p)$ is the code efficiency, where the size is n, executed on p processors. $S(n,p)$ is the speedup counted from axiom (1). As we can see this parameter can provide information about what was the quality of generated code according to the time execution. Such information is crucial when the code generation process needs to produce efficient code. Having information about code efficiency, the code generator should be able to reduce synchronization and communication costs, and balance load.

4. The Amdahl's law

Not all of the parts of a program can be executed in parallel. For example, all input and output instructions, such as printing something on screen or providing data from a keyboard, cannot be executed in parallel. The presence of such instructions is the reason why in some cases it is impossible to achieve full speedup. For that reason it is essential to know in what way such instructions can affect speedup. Assuming that we can divide code for two parts, the part in which we can execute instructions in parallel and the part where instructions have to be executed in a sequence, we can predict that the execution time will be the sum of these two parts. The equation (3) describes this situation:

$$T(n,p) \approx x * T(n,1) + \frac{(1-x) * T(n,1)}{p} \tag{3}$$

where $x \cdot T(n,1)$ is the execution time of x instructions which can not be execute in parallel. The x variable is normalized to <0; 1 >. The $[(1-x) * T(n,1)]/p$ is the execution time needed to execute the p instructions which can be executed in parallel.

Having such a formula it is simple to deduce some rules according to speedup. Let us divide both sides of formula (3) by $T(n,1)$.

$$\frac{T(n,p)}{T(n,1)} \approx \frac{x * T(n,1)}{T(n,1)} + \frac{(1-x) * T(n,1)}{T(n,1) * p} \tag{4}$$

Next, after doing some reductions, we achieve the equation (5):

$$\frac{T(n,p)}{T(n,1)} \approx x + \frac{1-x}{p} \tag{5}$$

Using axiom (1), we stipulate

$$S(n,p) \approx \frac{1}{x + \frac{1-x}{p}} \tag{6}$$

This formula is known as Amdahl's law. This law is interesting because it is simple to predict the speedup when the number of processors used for a program execution grows to the infinity.

$$\lim_{p \to \infty} S(n,p) \approx \frac{1}{x} \tag{7}$$

It is easy to make the deduction that if the processor number grows twice, speedup will not grow twice. This is because of synchronization and communication costs.

In a parallel execution process, when more than one thread executes loop, and loop's iterations are divided among a thread pool, it is crucial for the program correctness to synchronize all threads from the pool. When one of threads executes instructions faster than the other, the fastest thread must wait for the other threads to continue another part of instructions execution, because the program must be deterministic. So when a thread waits, it looses processor's time and does not execute any code. That affects speedup. Moreover during a wait period, the thread is in a *sleep* mode and needs to communicate with other threads to know if they have finished their execution. More threads means more synchronization and communication costs and lower efficiency. For that reason, it is highly recommended that all processors should have equal load balance. It means that all available processors should execute approximately the same amount of instructions to reduce the synchronization and communication time and to increase speedup and efficiency.

5. Parallel code generator architecture

A parallel code generator will be a modular program which will automatically generate a source code in the OpenMP 3.0 standard ready to be compile via any compiler which is dedicated to the OpenMP 3.0 standard. A modular architecture allows us to connect the code generator we are working on with any compiler suitable for compiling a source code in the OpenMP 3.0 standard. Moreover many external programs can be plugged in. For example, there is possibile to use any program which can produce a pseudo code with no data dependencies. Nevertheless, any program such as the Intel VTune Thread Profiler can be joined to measure quality parameters of a generated parallel code. A plugin architecture provides a flexible mechanism for configuring the code generator program for any suitable machine under which a generated program will be executed.

Intentionally, the generator tool will generate code which will be optimal under a specific runtime environment on a specific machine. What is the meaning of an optimal code or the quality of code is explained in paragraph 3.1 of this paper. In Figure 1, the structure of the code generation program is shown .

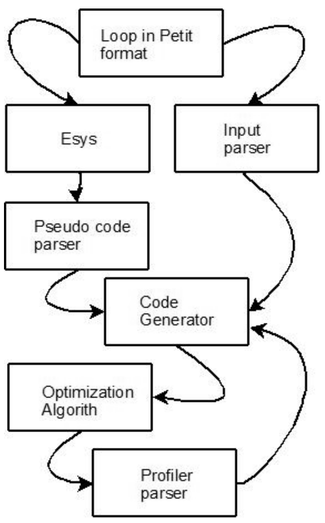

Fig. 1. A modular structure of an automatic parallel code generation program
Rys. 1. Modułowa struktura generatora kodu równoległego

In the first stage a loop must be provided in the Petit format. More piece of information about Petit syntax can be found in the Petit documentation website mentioned in paper [3], where more information about the Presburger's arithmetic is also included. Broadly speaking, Petit and the Omega calculator uses the Presburger's arithmetic to carry out dependence analyze. Only code with none of them can be executed in parallel.

Secondly, pseudo code in the Petit format is taken by the Esys program which was written by Marek Pałkowski [8]. This program provides a pseudo code which is generated from the Omega calculator tool and contains synchronization-free code slices which can be executed in parallel. Moreover, an input file with a loop written in the Petit format is parsed using an Input Parser. The Input Parser is a module which parses an input loop to create a list with lines of code, which are in the loop body. Nevertheless, the Input Parser creates all declared variables objects. Each declared variable is keep as an object with all pieces of information required to produce a syntax correct code in the C language.

When all variables objects are created and the input loop is parsed correctly, the Pseudo Code Parser module starts parsing the pseudo code obtained from the Esys program. Synchronization-free slices are parsed into code lines of a specific blocks of code which are required to generate a correct piece of code in the C language. In this phase all loops iterators are updated to gain data from vectors, and a pseudo code became the code in the C language. To parse a pseudo code, regular expressions were used because of a specific character of pseudo code generated by the Esys parser. The last step of this phase is an error correction process. The Esys program uses the Omega calculator's code generator, which produces a pseudo code with several errors, such as wrong vector indexing, doubling variables names and more, which are provided in the Petit and Omega documentation [2, 3]. Moreover, in this phase a pseudo code is corrected to be more user friendly and understandably during being studied by a programmer.

Having all pieces of information from the Pseudo Code Parser and the Input Parser, the Code Generator module can produce a piece of code written in the C language. All variables objects from the Input Parser are translated into the variable declaration section of C code. Needed libraries are also included into C code using the header mechanism. The obligatory *main* function is created and filled with synchronization-free code. The last step in this module is the creation of OpenMP 3.0 compatible parallel source code. After this step, a program is ready for a compilation process. For this moment only simple parallel code is generated with no optimization algorithm which is under construction. None of the optimization algorithm part is implemented because the creation of this algorithm is in a construction phase.

This algorithm will work iteratively to produce parallel code which is suboptimal for a specific machine. The stop condition of this algorithm is one of the following

- Provided number of iterations exceeded,
- The next iteration generates code which is not as optimal as code generated in previous iteration, according to the measurements described in section 3 of this paper,
- Provided quality of source code is being achieved.

For this moment, the parallel code generator can produce code as the first step of the algorithm. That means no optimization provided. Also measurements are not taken from the code parser but from directives provided into the OpenMP 3.0 standard. In chapter 4, an example is provided which explains the process of automated code generation.

6. Example

In this chapter, an example is provided to show how the code generator works. The code provided below is the input loop for the generator. This loop provides calculations allowing us to filter an image. As a result, the image became more fuzzy. Only the code of the crucial program's loop is provided below.

```
for (j = 0; j  <  ile; j = j + 1)
{
  for (k = 1; k  <  ile; k = k + 1)
    mac[j][k]=(mac[j][k-1]+mac[j][k])/2;
}
```

It is evident that all types of dependencies occur in our example. Figure 2 shows the dependencies in the above code.

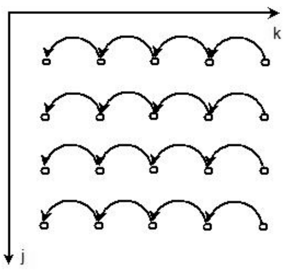

Fig. 2. The iteration analysis
Rys. 2. Analiza iteracji

Keeping Figure 2 under consideration, each iteration iterated by variable *j* can be executed in parallel. This is quite simple example but generally we can not always see loop dependencies from such a chart and not always it is so easy to deduce a solution how code can be executed in parallel. The Esys program generates pseudo code representing synchronization-free slices of code which can be executed in parallel. Such a code is presented below.

```
The number of dependence relations: 1.
1. {[i,j,v] -> [i',j',v'] i' = i && j' = 1+j
&& v = 5 && v' = 5 && 1 <= j <= 99
&& 1 <= i <= 100 }
-----------------------------
Sources of Slices: 1
{[i,j,v] j = 1 && v = 5 && 1 <= i <= 100 }
R_UCS
{[i,j,v] -> [i',j',v'] FALSE }
Outer loops scanning sources of synchronization-
free slices:
for(t1 = 1; t1 <= 100; t1++) {
       s1(t1,1,5);
}
Loops scanning elements of each slice in
lexicographical order:
for(t1 = 1; t1 <= 100; t1++) {
       s1(t1,1,5);
       if (t1 >= 1 && t1 <= 100) {
       for(t2 = 2; t2 <= 100; t2++) {
        s1(t1,t2,5);
        }
        }
}
```

The pseudo code contains information about dependance analysis results. They are delivered in two ways. The first of them is a Presburger's arithmetic representation which is rather not useful during the code generation process. The most useful representation is a simple pseudo code. After the parsing process of this pseudo code and after the code generation phase, the generator produces full code in the C language, which is compatible with the OpenMP 3.0 standard. The most important part of the code is the loop. The generated loop code is provided below.

```
#pragma omp parallel for private(t1,t2)
       for(t1 = 0; t1 <= 100; t1++) {
            mac[t1][1] = mac[t1][1-1]+mac[t1][1];
            if (t1 >= 1 && t1 <= 100) {
               for(t2 = 2; t2 <= 100; t2++) {
                      mac[t1][t2] = mac[t1][t2-1]
                      + mac[t1][t2];
               }
       }
}
```

Because the rest modules of the generator are not yet implemented, compilation and a testing process was done manually. For that reason the GCC 4.2.2 compiler was used. An input square matrix which represents an image was generated randomly. With every step of the test, a dimension of the matrix was enlarged. One code execution was counted as one step. Each of all experiments was repeated five times and the approximate time was taken as the result. As it was mentioned before, the time was represented as processor's clock ticks. Experiments were executed on the Intel Xeon Quad Core Processor with 8 MB L2 cache and 1066 MHz FSB. Each core clock was 1,6 GHz and the Intel Virtualization Technology was used. There were exactly eight processors which executed the parallel code. Figure 3 shows results gathered from experiments.

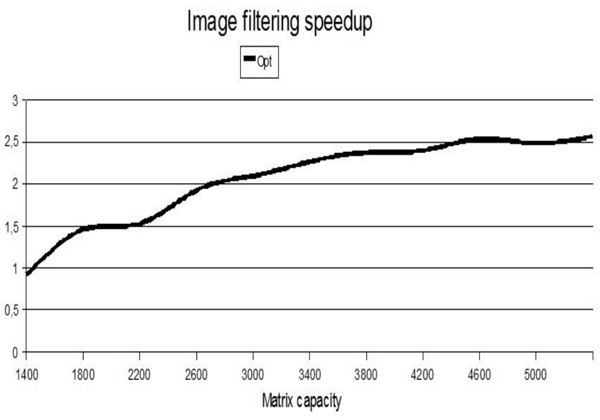

Fig. 3. Speedup of an image filtering
Rys. 3. Przyspieszenie filtrowania obrazu.

The maximum speedup during experiments was 2,6 for the image with 5000x5000 pixels. Speedup is defined by communication costs and additional cost spent on the thread creation process and the thread scheduling process. An optimization algorithm, which we are working on, must keep under consideration the reduction of these costs. There is also the need for the enhancement in the load balancing process to achieve better speedup.

7. References

[1] Van Der Pas R., Chapman B., Jost G.: Using OpenMP. The MIT Press, 2007.
[2] Omega Project Documentation. Omega project website: http://www.cs.umd.edu/projects/omega/. Valid at June 25, 2009.
[3] Petit Documentation. http://github.com/davewathaverford/the-omega-project/tree/2ff0a6563ed1c2b2eee8c9bf82f3657a8e6d6bc2/petit/doc. June 25, 2009. Petit tool project website available June 25, 2009.
[4] Fujitsu Systems Europe Ltd., Url valid at June 25, 2009, http://www.compunity.org/compilers/fujitsu/Workbench.pdf. Parallel Navi Workbench: an OpenMP Development Enviroment for a Wide-Area Network,.
[5] Abd-El-Barr M., El-Rewini H.: Advanced computer architecture and parallel processing. Wiley Interscience, 2005.
[6] OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, 2.0 edition, March 2002, Url valid at June 25, 2009 http://www.openmp.org/blog/specifications/cspec20.pdf..
[7] OpenMP Architecture Review Board, OpenMP Application Program Interface Draft 3.0, 3.0 edition, Oct. 2007. Url valid at June 25, 2009 http://www.openmp.org/drupal/mp-documents/cspec30_draft.pdf.
[8] Pałkowski M.: http://detox.wi.ps.pl/SFS_Project. Valid at June 25, 2009. ESyS project website.
[9] Kohr D., Chandra R., Dagum L.: Parallel programming in OpenMP. Morgan Kaufmann Publishers, 2001.
[10] Univ. of Electro-Communications, Japan. Interactive Parallelizing Assistance Tool for OpenMP:iPat/OMP. Url valid at June 25, 2009 http://www.compunity.org/events/ewomp03/omptalks/Monday/Session2/T08p.pdf.
[11] Addison Wesley. An introduction to parallel computing . Addison Wesley, 2003.

Artykuł recenzowany