An OpenMP support for Anahy's Model by ApenMP

Cristian Castañeda
Universidade Federal do Rio Grande do Sul
cffcastaneda@inf.ufrgs.br

Gerson Cavalheiro
Universidade Federal de Pelotas
gersonc@anahy.org

Abstract

In this work, we present the development of ApenMP, an implementation of the execution model based on tasks dependencies providing a programming interface implementing a subset of the OpenMP standard. ApenMP is a pre-processing tool that converts OpenMP code in Athreads code, allowing the execution of programs with support of scheduling strategies. The ApenMP project considers the execution model proposed by Anahy and the programming interface specifications proposed by OpenMP. In terms of performance, the ApenMP results were satisfactory for tasks and data parallelism.

1. Introduction

The Anahy's project proposes an execution environment model for high performance for multicores and clusters. This model is composed by two layers, which the first one defines a high level programming interface and the second one defines an execution mechanism implemented by a task scheduling strategy. With these layers, the model allows concurrency to be specified without the real parallelism offered by the architecture. Thus, it is possible to adapt the executive core to improve the scheduling considering the features by the combination of the architecture and the application [3].

The Anahy's model is implemented in Athreads [3], oriented to multiprocessors and clusters. Athreads offers a programming interface based on Pthreads [2] [5] allowing the description of concurrency of the application by a tasks decomposition model [1]. Furthermore, OpenMP [4] is highlighted by providing a programming interface dedicated to data parallelism and by the simplicity of code parallelization.

This works presents the development of ApenMP, a programming tool that implements the execution model proposed by Anahy and the programming interface based on OpenMP specifications. In this case, it supported the data decomposition model, respecting the programming interface proposed by OpenMP. The performance results of the code in OpenMP, Pthreads and Athreads were in accordance with the parallelism presented in the applications used.

This work is organized as follows. The Section 2 presents the Anahy's execution model. The Section 3 presents the OpenMP's programming model. Section 4 presents the ApenMP specifications. Finally the Section 5 presents the performance results of ApenMP and Section 6 and 7, presents the conclusions and references, respectively.

2. Anahy's Execution Model

In the Anahy's model there is a layer between the programming resources and the scheduling mechanism, responsible by the creation of a directed acyclic graph (DAG), which represents the program execution. In this DAG, each vertice corresponds to a task to be executed by the application and each edge represents the data communications between the tasks.

In the moment that all dependencies of a given task are satisfied, this task can be executed, and the entry data set is available to read. This graph is explored by lists scheduling algorithms for execution optimization. In Anahy, the scheduling strategies are based on the exploration of the characteristics of the program in execution and the architecture where the application is executing.

In model of Anahy, the scheduling strategies have to explore the program characteristics to be executed and the destination machine, where the program is executed. By this way, there is a scheduling algorithm composed by two internal subsystems: one of manipulation of information, responsible for the load of the system, and other responsible for the manipulation of the program's task and its execution on the architecture.

The strategy adopted in Anahy is the limitation by the executive core of the maximum number of concurrent activities. This limit is given by the number of active virtual processors (PV) owned by the executive core. By this way, the programmer can define the number of concurrent activities that exceed the available capacity in the architecture.

So the executive core should enable the execution of activities in the available computational resources.

3. OpenMP

OpenMP is based on fork/join model to the creation and synchronization of threads in the parallel regions. These threads are created inside of a core named thread-pool. This pool is created at the beginning of the execution and when a task is created, that task executes on a thread in the pool, returning the thread to the pool when the task is done.

The OpenMP interface programming has compilation directives like abstraction for the creation and synchronization of threads, and library services to enable the interaction of the program in execution with its environment [Chandra et al. 2001].

In OpenMP, the *parallel* directive indicates a parallel region, and the *for* directive indicates a loop parallelization. In the OpenMP scheduling, it can be *static*, *dynamic*, *guided* and *runtime*. The scheduling identifies the own policies related to the distribution of chunks in the threads of the thread-pool.

4. ApenMP

This programming tool consists in a pre-processor tool. This tool applies several techniques of compiling to translate OpenMP directives and clauses to Athreads code. The generated code by the pre-processor has the same semantics of the original code, respecting the dependencies between the instructions expressed in the user program. The process of code generation using ApenMP is shown in Figure 1. The input of ApenMP is an OpenMP code, and the output is an Athread code, after that the code is executed in the Anahy environment. The ApenMP prototype was developed in C language with the GCC 4.2.4. compiler over the GNU-Linux environment 2.6.24.

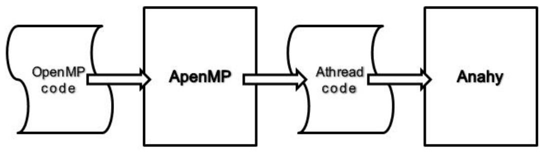

Figure 1. Compiling process using ApenMP

For the development of ApenMP, it was used two compilers tool: Flex and Bison. Both tools allows a good development and maintenance of the program. The process to develop ApenMP is shown in Figure 2. In *apenmp.l* the tokens are declared and in *apenmp.y* the grammar is specified.

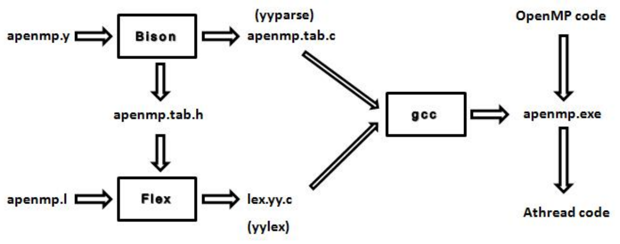

Figure 2. Generation of ApenMP

Another important stage is the code generation. The final code is generated with a set of services that uses the preprocessing results (identification of directives and clauses, and data manipulation). The generated code occupies the original portion code that contains the *pragmas* with the *parallel* and *for* directives.

In the stages of analysis of the original code, some actions must be taken to obtain the final code. In the sequence these actions are described:

- All code that does not express parallelism is unchanged.
- The *parallel* directive must be changed to *athread_create* function.
- The *for* directive is translated *athread_create* and the loop is partitioned in N chunks to the threads.
- All variables that have not been specified in some clause are privated in the thread.
- The data in *private* clause are declared in the threads, because they are privated for all threads.
- The data in *firstprivate*, *lastprivate* and *reduction* clauses are stored in data structures and manipulated for all threads.

In OpenMP, if no clause is specified for a determined variable, this variable is declared shared for all threads. This decision can be explained by the better performance of the execution that can be obtained in OpenMP, because the users omissions does not generate overhead in the execution. Anahy does not allow this kind of shared data, though ApenMP supports the shared clause between the parallel regions to keep the compatibility with OpenMP.

In ApenMP, shared variables generate an additional cost of execution, because they require mutexes to protect critical sections. So to avoid the simple omission by the user, that can generate additional costs of processing, ApenMP considers the *private* clause. The Figure 3 and 4 shows an example of an original and final code. As we can see, the sequential instructions in the original code are maintained in the final code.

```
#include <omp.h>
void main(int argc, char **argv){
     int x, y, soma;
     #pragma omp parallel private(x, y,
soma)
                soma = x + y;
}
```

Figure 3. Original Code

```
#include "athread.h"
void main(int argc, char **argv){
     int x, y, sum, ampAux, numThreads;
     athread_t ampth[numThreads];
     struct valores *head;
     void *ret;
     aInit(&argc, &argv);
     for(ampAux = 0; ampAux<numThreads;
ampAux++)
         athread_create(&ampth[ampAux],
NULL, amp1, head);
     for(ampAux = 0; ampAux<numThreads;
ampAux++)
         athread_join(ampth[ampAux],
&ret);
     aTerminate();
}

void amp1(void *dados){
     struct valores *res;
     int x, y, sum;
     sum = x + y;
     return (void *) res;
}
```

Figure 4. Final code

5. Performance Results

The performance results were obtained using a Core 2 duo 1.86GHz with 4 MB Cache, 2 GB RAM. The operating system used was the GNU-Linux kernel 2.6.24, the compiler used were Intel and GCC 4.2.4. for the compilation of OpenMP programs. The methodology to evaluate the performance was to account the execution time of the program in seconds. The execution time only counts the parallel regions. The times represent the average and the standard deviation of 15 executions and for each case they were used 2, 4 and 8 threads.

Two approaches of parallelism were chosen, the task parallelism and the data parallelism to evaluate the final execution performance with OpenMP and Pthreads through these two approaches. So, it was used an application for each approach, the application for data parallelism was the Monte Carlo 2D algorithm and for the task parallelism was the Sieve of Eratosthenes Algorithm, where for each application, it was obtained 3 versions: OpenMP, ApenMP (Athreads Version) and Pthreads. Some considerations are important:

- The Pthreads code were translated manually without ApenMP execution and compiled with GCC.
- The Athreads code were translated using ApenMP and compiled with GCC and executed in Anahy.
- The OpenMP code were created manually and compiled with Intel and the GCC.
- The OpenMP scheduling used was *static*.

5.1. Task Parallelism

The performance results executing the Sieve of Eratosthenes algorithm that specifies tasks parallelism are shown in Tables 1 and 2. As can be seen, the execution time of the version through ApenMP was lower than OpenMP and Pthreads. This result is satisfactory, but it should be noted that the task parallelism is not the niche of OpenMP.

	OpenMP(intel)			OpenMP(GCC)		
Threads	2	4	8	2	4	8
Average	0,268	0,356	0,587	0,276	0,398	0,623
SD	0,007	0,002	0,003	0,018	0,008	0,009

Table 1. Execution time of Task parallelim

	ApenMP			Pthreads		
Threads	2	4	8	2	4	8
Average	0,259	0,350	0,533	0,271	0,361	0,601
SD	0,002	0,006	0,010	0,014	0,021	0,003

Table 2. Execution time of task parallelim

5.2. Data Parallelism

In the Table 3 and 4, They show the performance by executing the Monte Carlo 2D algorithm that specifies the data parallelism. In this approach, the OpenMP execution time is lower than the others, but the ApenMP execution time was lower than Pthreads too.

	OpenMP(intel)			OpenMP(GCC)		
Threads	2	4	8	2	4	8
Average	0,201	0,204	0,321	0,210	0,218	0,332
SD	0,005	0,009	0,013	0,007	0,016	0,031

Table 3. Execution time of data parallelism

	ApenMP			Pthreads		
Threads	2	4	8	2	4	8
Average	0,334	0,365	0,389	0,354	0,398	0,401
SD	0,051	0,032	0,031	0,005	0,007	0,011

Table 4. Execution time of data parallelism

As we can see in the tables, its interesting the usage of mixed code with OpenMP and Pthreads, but some considerations are important:

- The nature of the application.
- The definition in the code where each approach is better.

6. Conclusions

Using ApenMP, the programmer can benefit from the advantages of simplicity of program parallelization in OpenMP and obtain an efficient performance by using a executive core provided by Athreads. About the performance results, these results were satisfactory, reflecting the expected nature for the studied cases. It was observed that the experiments performed with applications that highlights the task parallelism, the execution time of the versions translated to Athreads was lower than those presented with the compilation of the case in OpenMP. In contrast, the codes that expressed data parallelism, the performance time of OpenMP was better.

The continuity of this work may be given by introducing the complete set of specifications of OpenMP in the preprocessor ApenMP. Other interesting work, can be the implementation of a execution core itself, eliminating the generation of intermediate Athreads code.

References

[1] D. Butenhof. Programming with posix threads. addison-wesley professional computing series, 1997.

[2] G. Cavalheiro. Principios da programacao concorrente. in erad, 2004.

[3] G. Cavalheiro, L. Gaspary, M. Cardozo, and O. Cordeiro. Anahy: A programming environment for cluster computing. in vecpar, 2006.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, and R. McDonald, J. Menon. Parallel programming in openmp, 2001.

[5] C. Leopold. Parallel and distributed computing. john wiley sons, 2001.