ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Parallel Programming with OPENMP

Igli Tafa, Erlind Çuka, Julian Fejzaj
Informatics Engineering Department, Faculty of Information Technology, Polytechnic University of Tirana
Informatics Department, Faculty of Natural Science, University of Tirana

Abstract

In this paper we will make an experimental description of the parallel programming using OpenMP. Using OpenMP, we achieve a high performance parallelizing the code into threads.

We will see the difference when a simple code which calculates the average of a vector will be executed sequentially and in parallel. We will calculate the total execution time, and will monitor CPU loading while is running one or another.

I chose this topic because I wanted to be more familiar with multi-threaded and programming in parallel. More and more, parallel programming is developing rapidly. So understanding of this programming would be good investment in my career.

Keywords: OpenMP, CPU loading, parallel programming

Introduction

As CPU speeds no longer improve as significantly as they did before, multicore systems are becoming more popular.
Parallel programming is the execution of small unit called threads at the same time, increasing the speed of execution. But how threads are created? How they operate? When they finish their work?
Once a program starts executing, then automatically create a process which is responsible for the progress of the program. The newly created process creates a thread which is called the master thread. Only the master thread can create other thread if are necessary in program, giving each one a small task to do. Once they finish their tasks they are terminated leaving only the master thread.

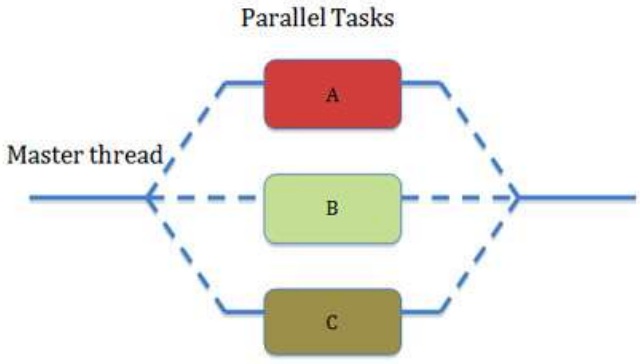

Figure 1. Execution of threads

OpenMP is very easy to use. The same code is used in serial and parallel programming, by equating the distribution of load. It runs only in shared memory. Each thread has its private memory and its stack. They have the right te access the memory, which can be used as communication path between threads. OpenMP use fork and join model during execution of threads.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

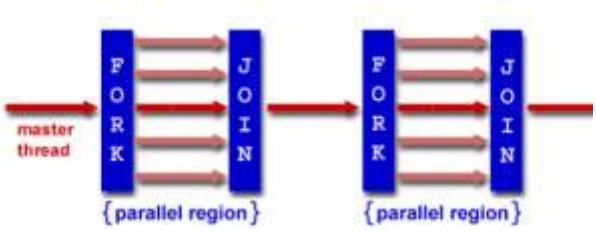

Figure 2. Fork- Join Model

Fork: Master thread creates a team of threads that depends from user or code. Each of these threads make a job independently, accessing memory at the same time.

Join: After threads finish their jobs they are terminated leaving only the master thread.
How many threads do I need?

In general, the number of threads required in a program, is limited to the number of CPUs. Whenever threads are created, a little time is taken to create a thread and when they finish, they join the master thread. If the problem is small, and we have a computer where the number of CPUs is less than the number of threads, the execution time will be longer using threads. So we have to make a deal with the numbers of CPUs and the threads that we are using.

OpenMP is an Application Program Interface (API) which is used for creating multi-threaded, in shared memory parallelism. It has its libraries where are located different functions that it use.

Related works

When hardware development is nearing its end, software development is required. Numerous methods have been developed to parallelize programs using as much CPU and reducing the total time of execution.

The OpenMP standard specifications began in 1997 but first ideas had started earlier. In its beginnings, in parallel programming languages was part of FORTRAN and C/C++ where the implementation was made in parallel. In 2005 was realised the first program OpenMP with an API of its own. Since then, it has been developed even more to reach the standards of present days [1].

Many methods have been developed to parallel programming. Each of them has its own features and characteristics of parallelism conception, but the researchers believe that the best way for parallel programming is OpenMP.

David J.Kuck, Director of Parallel and distributed Solutions, Intel Corporation, when he was interviewed said: "I hope that readers will learn to use the full expressibility and power of OpenMP".[2] Since OpenMP research is often complicated by the tight coupling of the compiler translation and the runtime system, we present a set of rules to define a common OpenMP runtime library (XOMP) on top of multiple runtime libraries. [3]

Another work that I found interesting and would like to mention is the paper [6]. Here is introduced the difference between OpenMP and MPI for execution on distributed memory systems. Also it is discussed how to evaluate the performance achieved by OpenMP applications.

A paper [8] is made for Hybrid MPI/OpenMP Parallel Programming. This is a hybrid model to solve the problems of OpenMP and MPI, where is explained the creation of a hybrid model which takes advantages of both models above.

Other researchers are made in timing behaviour, but it is not the purpose of this paper.[4]

Nowadays, not only CPU but also GPU goes along the trend of multi-core processors so implementing OpenMP presents not only an opportunity but also a challenge at the same time. [5]

Theory of experiment

We are going to create two programs that calculate the average of the vector. First we are writing in serial, meaning that each part of the code is run sequentially. Second we are writing in parallel, using threads. For every program we are calculating the time it needs to be executed and we will see the difference when using threads and without using threads.

- **Environment**

We are operating in Ubuntu 12.04. I find it more familiar. My computer has these parameters

- Intel Pentium Core 2 (2.04 GHz,800Mhz)
- 4 GB DDR2
- 250 GB HDD
- 32 bit operating system

- **Programming Languages**

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

I choose to work with C, because I am more familiar with this language, and OpenMP supports very well this language.

Experimental phase

The experimental phase has to do with finding the average in a vector. We have selected a vector with predetermined number of elements. For each iteration we will collect every number of vectors. At the end of iteration, we will calculate the average by dividing the sum with the number of elements.

The parallelization is done using #pragma omp directory. We will determine the number of the thread that in this case is three. Number of threads is determined by function:

omp_get_num_threads ().

To determine execution time we will calculate the time at the beginning and end of the program, using this function:
omp_get_wtime ();

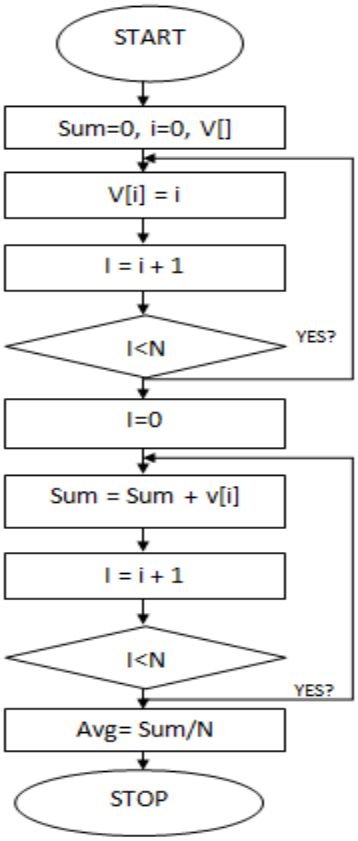

Figure 3. Flowchart of algorithm

After we have written both codes, we are going to run them in terminal. I installed OpenMP library with the command
sudo apt-get install gcc -4.6
This command deals with the installation of the library by itself.
To execute the codes we follow these steps:

- Open Terminal
- cd Desktop
- gcc –fopenmp Averag.c –o Averag
- ./Averag
- gcc –fopenmp ParallelAverag.c –o ParallelAverag
- ./ParallelAverag

After we have written these codes we will see these results:

Figure 4. Execution of programs

As we see from the terminal window, the execution in parallel is faster than serial. In our program is about 2 times faster. The result is the same in both of codes. But let's see the CPU loading while programs are executing.

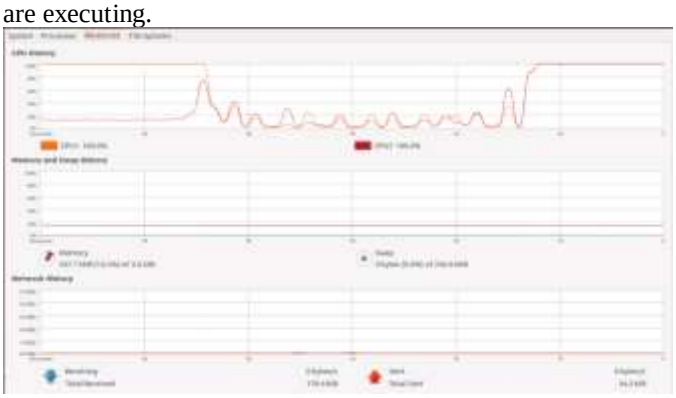

Figure 5. The load of CPU in parallel

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

As seen from the figure above, in parallel programming we have a maximum loading of CPU (100%). Each thread is executed in different core of CPU so the parallelized program is faster. While in sequential programming one core of CPU is loading in time. So we do not have an even distribution of load.

Conclusions

Interpreting the results of experimental part, so comparing the two programs, one in serial and the other in parallel, there is no doubt to use the parallel programming. But not in all cases should be used. In those cases where programs are small and have a few cycles, the creation and destruction of threads will give us greater delays in execution, than using it in serial. So we have to make a deal with the resources of hardware and the source code that will be executed.

In our program we saw how the total time of execution is two times faster in parallel than serial programming. Using threads, both of cores in CPU stay full of charging, using hardware resources as well as possible.

Future work

The main reason why I started this work was because I was very interested in parallel programming. It has a large spreading nowadays.

As the number of cores in a CPU is increased, the use of threads will also grow. Time execution is the most critical point nowadays. In my future work I would like to make a review of parallel programming, comparing in different methods, such as Message Passing Interface.

References

1. OpenMP 4.0 Specifications Released. The OpenMP API specification for parallel programming, Jul 23, 2013
2. Using OpenMP by Barbara Chapman, Gabriele Jost and Ruud van der Pas 2008, Massachusetts Institute of Technology
3. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries, By Chunhua Liao Daniel J. Quinlan , Thomas Panas and Bronis R. de Supinsk, Center for Applied Scientific Computing Lawrence Livermore National Laboratory Livermore, 2010
4. Time Behavior Intel® Fortran Compiler XE 13.1 User and Reference Guides, 26 March 2013
5. Comparison of OpenMP & OpenCL Parallel Processing Technologies By Krishnahari Thouti 2012
6. Introduction to OpenMP By Blaise Barney, Lawrence Livermore National Laboratory, 1 March 2010
7. Developing Parallel Programs — A Discussion of Popular Models : An Oracle White Paper September 2010
8. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming by Georg Hager Gabriele Jost Rolf Rabenseifner Proceedings of the 2009 17th Euromicro International Conference on Parallel, USA
9. Proceedings of the 5th International Workshop on OpenMP, Berlin, Heidelberg, Springer-Verlag (2009) Liao, C., Quinlan, D.J., Willcock, J.J., Panas

Appendix

Below is the source code for serial and parallel programming:

Serial Programming

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
#define N 500000
float A[N];
int main(void)
{
float avg=0,sum=0;
float start_time,end_time;
long j,x=0; int y=0;
start_time=omp_get_wtime();
for(j=0;j<N;i++)
{A[j]=j;
sum=sum+j;
avg=sum/j;
for (x=0;x<100;x++)
y=x*0.55;
}
end_time=omp_get_wtime();
printf("Average is=%f\n",avg);
printf("Calculation :%f seconds\n", end_time-start_time);
}
```

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

Parallel Programming

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define N 500000
#define NUM_THREAD 3
float A[N];
int main(void)
{
float sum=0, avg=0;
float start_time,end_time;
long j,x=0;
int y=0;
int nthr;
float th_max[NUM_THREAD]={0.0,0.0,0.0};
start_time=omp_get_wtime();
#pragma omp parallel
private(j,x,y,sum,avg),shared(A,th_max)
{
#pragma ompfor
for(j=0;j<N;j++)
A[j]=j;
nthr=omp_get_num_threads();
#pragma omp for // creates some new threads
for(j=0;j<N;j++)
{
sum=sum+A[j];
for (x=0;x<100;x++)
y=x*0.55;
}
avg=sum/N;
}
th_max[nthr]=avg;
for(i=0;i<NUM_THREAD;i++)
if(avg<th_max[i])
avg=th_max[i];
end_time=omp_get_wtime();
printf("Average is=%f\n",avg);
printf("Calculation :%f seconds\n", end_time -
start_time);
```