Performance Analysis of Parallelizing Program Code for Multi-Core Architectures Using OPENMP

1 Neha Agrawal, Ruchira Datta M Rajasekhara Babu, Ch. Pradeep Reddy, 2 Y. Shabber Ali
1 School of Computing Science and Engineering
VIT University, Vellore, 632014, T.N., India
2 BIT Institute of Technology, Hindupur, Andhra Pradesh, India
1Rajababu.m1@gmail.com

ABSTRACT

Multi-core architectures have become more popular due to better performance, reduced heat dissipation, power consumption and more efficient simultaneous processing of multi tasks. If one wishes to run a single application faster, then that application must be divided into sub programs, or threads, that cooperate to deliver the desired functionality. The OpenMP programming paradigm implements loop level parallelism, which is one of the most basic available units of parallelism for parallel OpenMP programs [3]. Loop-level parallelism allows an OpenMP implementation to easily split the work across multiple threads. This paper mainly focuses on Parallelizing programming code using OpenMP and analyzes the program performance using V-tune.

Keywords: Multi-Core Architectures, Open-MP, Parallelism.

1. INTRODUCTION

In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads. Computer architectures can be classified into two different dimensions [1]. The first dimension is the number of instruction streams that particular computer architecture may be able to process at a single point in time [2]. The second dimension is the number of data streams that can be described in terms of how instructions and data are processed. This classification system is known as Flynn’s taxonomy [1].

Modern microprocessors are now moving to multi-core architectures to extract more performance from available chip area [3]. As a result, multi-threaded applications may potentially exploit maximum benefit from a multi-core architecture. These multithreading applications are implementing using OpenMP directives. OpenMP is a method of parallelization where the master "thread" divides a specified number of slave "threads" and tasks are divided among them [3]. The threads then run concurrently, and the runtime environment allocates the threads to different processors.

Here the effort and work has tried to explore openMP method of parallelizing threads to bring about a better performance of the code. For the benefit a scrap code the Fast Fourier Transform Algorithm (FFTW) has been taken and optimization of the code has been done to gain efficiency in its implementation. For this a thorough analysis of the code and its various loops has been done and the ways in which it can implement the programs of open MP to achieve the goal. The code will be optimized in such a way that it efficiently uses the dual core machine to compute the result.

2. FFTW

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data.

The discrete Fourier transform (DFT) is an important tool in many branches of science and engineering and has been studied extensively. For many practical applications, it is important to have an implementation of the DFT that is as fast as possible. In the past, speed was the direct consequence of clever algorithms that minimized the number of arithmetic operations. On present day general-purpose microprocessors, however, the performance of a program is mostly determined by complicated interactions of the code with the processor pipeline, and by the structure of the memory. Designing for performance under these conditions requires an intimate knowledge of the computer architecture. Here is a list of some of FFTW's interesting features: Speed, both one-dimensional and multi-dimensional transforms. Arbitrary-size transforms. (Sizes with small prime factors are best, but FFTW uses O (N log N) algorithms even for prime sizes.) Fast transforms of purely real input or output data. Transforms of real even/odd data: the discrete cosine transforms (DCT) and the discrete sine transforms (DST). Efficient handling of multiple, strided transforms. (This lets you do things like transform multiple arrays at once, transform one dimension of a multi-dimensional array, or transform one field of a multi-component array). Parallel transforms: parallelized code for platforms with Cilk or for SMP machines with some flavor of threads (e.g. POSIX). Portable to any platform with a C compiler. Documentation in HTML and other formats.
a. The 1d Discrete Fourier Transform (DFT)

The forward (FFTW_FORWARD) discrete Fourier transform (DFT) of a 1d complex array $X$ of size $n$ computes an array $Y$, where:

$$ Y_k = \sum_{j=0}^{n-1} X_j e^{2\pi j k/n} \quad \ldots \ldots \ldots (1) $$

The backward (FFTW_BACKWARD) DFT computes:

$$ Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi j k/n} \quad \ldots \ldots \ldots (2) $$

FFTW computes a UN normalized transform, in that there is no coefficient in front of the summation in the DFT. In other words, applying the forward and then the backward transform will multiply the input by $n$. From above equations (1) 7 (2), an FFTW_FORWARD transform corresponds to a sign of -1 in the exponent of the DFT. Note also that the standard “in-order” output ordering is used—the k-th output corresponds to the frequency $k/n$ (or $k/T$, where $T$ is your total sampling period). For those who like to think in terms of positive and negative frequencies, this means that the positive frequencies are stored in the first half of the output and the negative frequencies are stored in backwards order in the second half of the output. (The frequency $-k/n$ is the same as the frequency $(n-k)/n$.)

b. Three main ideas are the keys to FFTW’s performance:

First, the computation of the transform is performed by an executor consisting of highly-optimized, compassable blocks of C code called codelets.

Second, at runtime, a planner finds an efficient way (called a ‘plan’) to compose the codelets. Through the planner, FFTW adapts itself to the architecture of the machine it is running on.

Third, the codelets are automatically generated by a codelet generator written in the Caml Light dialect of ML. The codelet generator produces long, optimized, unreadable code, which is nevertheless easy to modify via simple changes to the generator.

3. DESIGN AND IMPLEMENTATION

The Frame Work above explains the process in which the paper needs to go about. First, the FFT code needs to be written and tested. Then after its analysis and study the parallelization techniques need to be applied. The program obtained thus should be executed on the single core and the dual-core processors available. The results thus obtained should be analyzed with the V-Tune analyzer. The conclusion can be derived thereafter. The above diagram is the architecture of the entire paper. The various software and hardware requirements have been shown along with a brief way of implementation. The basic necessary requirements

**Figure 1:** Overview of system

**Figure 2:** Architecture diagram of FFT
are the Dual-core machine, the FFT code, the Open MP compiler and the V-Tune analyzer for analyzing the performance.

4. IMPLEMENTATION

Performance variation from the general C compilers and the Open MP compilers

Figure 3: Flowchart for FFT

This program has two functions, one which has the parallelization technique applied and the other that has the simple C code. Both the functions are run one after the other and the time of each of their execution is noted. The difference in the time of execution for both is hence calculated and displayed. The results thus obtained show a variably vast difference in the time required to execute the function without parallelization and the time taken by the function with parallelization.

Function without parallelization Algorithm:

Function: Without Parallelization
1. Start
2. For (i=0 to 1000)
   2.1 a=a+b
   2.2 b=b++
3. End for
4. Stop
Function with parallelization Algorithm

Function: With Parallelization
1. Start
2. Insert pragma to parallelize
3. For (i=0 to 1000)
   3.1 a=a+b
   3.2 b=b++
4. End for
5. Stop

FFT program with parallelization techniques applied too it and the enhancements in the performance of the program proved through the output.

This is a program to calculate the Discrete Fourier Transform. The various functions needed are described in the algorithm. Various programs have been inserted in the appropriate places to avoid race conditions and discrepancies in the result and obtain the maximum amount if parallelization that can be achieved.

The performance enhancements have been obtained which show a time difference in the execution of the program with the C compiler and the Open MP compiler. The ompicc compiled time is variably less compared to the time taken by the cc compiler.

Algorithms Used

[1] Init DFT Algorithm:
1. Start
2. Define the required variables
3. Initialize the arrays a space dynamically
4. Assign the initial values to the variables accordingly
5. Stop
[2] DFT Simple Algorithm:

1. Start
2. Call the function Clear Counters
3. Call the function Init DFT
4. Insert the pragma to parallelize the loop
5. For (i=0 to i<Size of array, i++)
   5.1 begin
   5.2 Insert pragma for nested for loop
   5.3 For (j=0 to j<Size of array, i++)
      5.3.1 Begin
      5.3.2 Counter variable: stat_mult += 4
      5.3.3 Counter variable: stat_add += 4
      5.3.4 Counter variable: stat_trig += 4
      5.3.5 Assign values to the dynamic DFT array
      A[i] = A[i] + X[j]*cos(Q*i*j) + Y[j]*sin(Q*i*j)
      B[i] = B[i] + Y[j]*cos(Q*i*j) - X[j]*sin(Q*i*j)
      5.3.6 Display the values of the array A and B
      5.4 End for
   5.5 End for
5.6 Display the values of the array A and B
6. End for
7. Display the values of the counters
8. Stop

[3] DFT Faster Algorithm

1. Start
2. Call the function Clear Counters
3. Call the function Init DFT
4. Insert the pragma to parallelize the loop
5. For (i=0 to i<Size of array, i++)
   5.1 W = M_PI * i
   5.2 initialize the array A, B
   5.3 Insert pragma to parallelize the inner loop
   5.4 For (i=0 to i<Size of array, i++)
      5.4.1 Begin
      5.4.2 Counter variable: stat_mult += 5
      5.4.3 Counter variable: stat_add += 2
      5.4.4 Counter variable: stat_trig += 2
      5.4.5 Assign values to AT and BT as
      AT += C[k]*X[j] - S[k]*Y[j]
      BT += C[k]*Y[j] + S[k]*X[j]
      5.5 End for
   5.6 Assign the values of AT and BT to arrays A and B
6. End for
7. Display the values of the array A and B
8. Display the values of the counters
9. Stop

[4] DFT Table Algorithm:

1. Initialise the look up table dynamically as C and S
2. Call function Clear Counters
3. Call function InitDFT
4. Insert pragma to parallelize the loop
5. For (i=0 to i<Size of array, i++)
   5.1 Array C = cos of (M_PI * i)
   5.2 Array S = -sin of (M_PI * i)
   5.3 counter variable stat_trig += 2
6. End for
7. Insert pragma to parallelize the loop
8. For (i=0 to i<Size of array, i++)
   8.1 initialize AT and BT array with values of X and Y
   8.2 Insert pragma for nested for loop
   8.3 For (i=0 to i<Size of array, i++)
      8.3.1 stat_mult += 4
      8.3.2 stat_add += 5
      8.3.3 Assign values to AT and BT as
      AT += C[k]*X[j] - S[k]*Y[j]
      BT += C[k]*Y[j] + S[k]*X[j]
9. End for
10. Display the values of the arrays A and B
11. Display the values of the counters
12. Stop

[5] DFT Tukey 2 Algorithm:

1. Initialize variables required
2. Call function Clear Counters
3. Call function Init DFT
4. M = log of Size to the base 2
5. N2 = Size of the array
6. Insert pragma to parallelize the loop
7. For (i=0 to i<Size of array, i++)
   7.1 N1 = N2
   7.2 N2 = N2/2
   7.3 E = M_PI *2/N1
   7.4 Z = 0
   7.5 stat_mult += 2
   7.6 stat_add += 1
   7.7 Insert pragma to parallelize nested for
      7.8 For (i=0 to i<Size of array, i++)
         7.8.1 C = cos of Z
7.8.2  S = sin of Z
7.8.3  stat_trig += 2
7.8.4  stat_mult += 1
7.8.5  Z = (j+1)*E //this will be the array indices
7.8.6  Insert pragma to parallelize nested for
7.8.7  For (i=0 to i<Size of array, i++)
7.8.7.1  i += N1
7.8.7.2  stat_mult += 4
7.8.7.3  stat_add += 7
7.8.7.4  L = i+N2 //this will be the array indices
7.8.7.5  XT=A[i]-A[L]
7.8.7.7  YT=B[i]-B[L]
7.8.7.8  B[i] =B[i]+B[L]
7.8.7.9  A[L] = C*XT + S*YT
7.8.7.10 B[L] = C*YT - S*XT
7.8.8  End for
7.8.9  end for
7.8  End for
9.  J=0
10. Insert pragma to parallelize for
11. For (i=0 to i<Size of array, i++)
11.1 if i<j then
11.1.1 Do begin
11.1.2 Swap the values of A[i] and A[j]
11.1.3 Swap the values of B[i] and B[j]
11.2 end if
11.3 stat_add += 1
11.4 k= Size/2
11.5 j := k
12. End for
13. Display the values of the DFT arrays A and B
14. Display the values of the counters
15. Stop

Program to show the splitting of work among various threads created

This program is a further enhancement of the previous DFT code to display the different threads working in the different sections of the program. It explicitly shows the number of the thread that are created by parallelization and which thread number is executing on which part of the iteration.
5. RESULTS AND DISCUSSIONS

This snap shot is obtained after the execution of the program given in 3.4.1. The result clearly shows the difference between the time taken by the function that is not parallelized and the one that is parallelized. A vast difference is obtained.

The time taken by the Open MP is comparatively very less compared to the C compiler. Hence the DFT code runs with improved efficiency and can provide results for complex data’s in a reduced time frame.

Thread Analysis the threads obtained are displayed according to their presence in the different iterations. The threads number along with the iteration number that it is running is explicitly displayed. The graph (fig 9 above) shows the pattern in which the time of execution varies when a program is executed in a C compiler and when it is parallelized and executed with the Open MP. The number of iterations are varied from 200 to 1000 and compiled each time with both the compilers. The time thus obtained is graphically represented. The time difference increases as the
The complexity of the program increases hence showing greater efficiency in with the Open MP.

Fig 4.3.1: Snapshot of the threads obtained for the FFT program after analyzing in the V-Tune

Fig. 4.3.2: Snapshot of the threads obtained for the FFT program after analyzing in the V-Tune

The time difference increases as the complexity of the program increases hence showing greater efficiency in with the Open MP.

6. CONCLUSION AND FUTURE ENHANCEMENTS

The Open MP specifications are analyzed and implemented in the FFT code and the results have shown a remarkable difference compared to the Intel C/C++ compilers. The efficiency obtained after parallelization of the FFT code has also been analyzed with the V-Tune analyzer which clearly shows the splitting of threads. The results have shown a vast improvement in the efficiency of the FFT code. There is scope for future enhancement in this field. The efficiency obtained has been tested in the Dual-core machines and further enhancements can be done by implementing the programs on the Quad-core, etc. The data obtained can be taken further to define a mathematical model to give a relation between the number of iterations and the time taken. The further extensions can bring about relations between the number of threads and the number of processors, therefore making it easy to calculate the efficiency obtained. Various other applications will derive benefits from research in this field.

REFERENCES


