ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 10, October 2014

Efficiency of Parallel Algorithms on Multi Core Systems Using OpenMP

Sheela Kathavate[1], N.K. Srinath[2]

Associate Professor, Department of CSE, Sir M. Visvesvaraya Institute of Technology, Bangalore, India[1]

Professor and Dean, Department of CSE, R.V. College of Engineering, Bangalore, India[2]

Abstract: The improvement in performance gained by the use of a multi core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores and this effect is described by Amdahl's law. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. In order to exploit the complete capabilities of multi core systems, applications have to become increasingly parallel in nature. Writing parallel program is not an easy task. OpenMP programming model helps in creating multithreaded applications for the existing sequential programs. This paper analyses the performance improvement of a parallel algorithm on multi core systems. The experimental results shows Significant speed up achieved on multi core systems with the parallel algorithm.

Keywords: Multi core Systems, OpenMP, Parallel algorithms, Performance analysis, speedup

I. Introduction

There is a continual demand for greater computational power from computer systems than is currently possible. Every new performance advance in processors leads to another level of greater performance demands from businesses and consumers [1]. Numerical simulation of scientific and engineering problems require great computational speed. These problems often need huge quantities of repetitive calculations on large amounts of data to give valid results and the computations must be completed within a reasonable time period. Multicore and multithreaded CPUs have become the new approach to obtaining increases in CPU performance [2] and the result is the invention of parallel computing which can be broadly classified as multi-processor and multi core systems. A multicore is an architecture design that places multiple processors on a single die (computer chip). Each processor is called a core. This concept is called chip multi-processing (CMP). Presently, the CMP has become the preferred method for improving the overall system performance. It is inevitable that paradigm shift from writing sequential code to parallel has to happen.

Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large number of processing elements. The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If α is the fraction of running time a program spends on non-parallelizable parts, then:

$$\lim_{P\to\infty} \frac{1}{\frac{1-\alpha}{P}+\alpha} = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program, with P being the number of processors used. If the sequential portion of a program accounts for 10% of the runtime ($\alpha = 0.1$), we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units.

The level to which an existing application can be parallelized is very important. Programmers must be capable of finding the best places in the application that can be divided into equal work load which can run at the same time and determine when exactly the threads can communicate with each other [3]. Matrix multiplication is one good application that falls in this category and is also required as a fundamental computation for many scientific and engineering applications. In this paper, we have considered parallelizing matrix multiplication program and see the performance improvement on different core configurations.

The paper is configured as follows: section 2 gives an overview of OpenMP, section 3 gives the related work in this field, section 4 gives the algorithm and experimental setup, section 5 gives result analysis and section 6 gives the conclusion and the future work.

II. OPENMP

OpenMP is an application program standard (specification) for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran 77/90 and C/C++ programs. OpenMP uses multiple, parallel threads to accomplish parallelism. It uses the concept of fork and join model as shown in figure 1. OpenMP has been very successful in exploiting structured parallelism in applications [4].

A thread is a single sequential flow of control within a program. OpenMP simplifies parallel application development by hiding many of the details of thread management and communication. It uses a directive-based method to explicitly

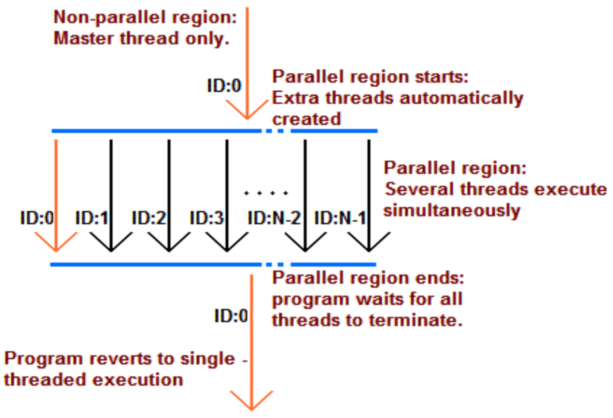
Non-parallel region:
Master thread only.
ID:0
Parallel region starts:
Extra threads automatically created
Parallel region:
Several threads execute simultaneously
ID:0
ID:1
ID:2
ID:3
ID:N-2
ID:N-1
Parallel region ends:
program waits for all threads to terminate.
ID:0
Program reverts to single - threaded execution

Figure 1. Fork and Join Model

tell the compiler how to distribute programs across parallel threads. OpenMP-enabled compilers include ones from Intel and Portland group, and open sourced GNU. Serial code statements usually don't need modification. OpenMP allows code to be parallelized incrementally, one subroutine/function or even one loop at a time. The directives are easy to apply and make the code easy to understand and maintain. OpenMP is widely available and used, mature, lightweight, and ideally suited for multi-core architectures. Data can be shared or private in the OpenMP memory model. When data is private it is visible to one thread only, when data is public it is global and visible to all threads. OpenMP divides tasks into threads; a thread is the smallest unit of a processing that can be scheduled by an operating system. The master thread assigns tasks unto worker threads. Afterwards, they execute the task in parallel using the multiple cores of a processor.

III. RELATED WORK

In [4], the authors present the efforts of the OpenMP 3.0 sub-committee in designing, evaluating and seamlessly integrating the tasking model into the OpenMP specification. The design goals and key features of the tasking model, is discussed which includes a rich set of examples and an in-depth discussion of the rationale behind various design choices. The prototype implementation of the tasking model with existing models is built and it is evaluated on a wide range of applications. The comparison shows that the OpenMP tasking model provides expressiveness, flexibility, and huge potential for performance and scalability.

In [5], the authors design an OpenMP implementation capable of using large pages and evaluate the impact of using large page support available in most modern processors on the performance and scalability of parallel OpenMP applications. Results show an improvement in performance of up to 25% for some applications. It also helps improve the scalability of these applications. In [6], the “The Game of Life” problem is written using OpenMP and MPI and run on Sun E3000 and OpenMP does better compared to MPI. In [8], the calculation of pi and Gaussian Elimination algorithms are tested with and without OpenMP and the speedup achieved with parallelization is better.

IV. ALGORITHM AND EXPERIMENTAL SETUP

In this analysis, we have implemented both the sequential and parallel algorithms for matrix multiplication. The matrices used are both square matrices. First, the sequential code is executed and the time taken for multiplying the matrices are taken. Then the program segment which can be parallelized is divided into threads by using the OpenMP compiler construct. Here each thread runs independent of other threads. The program is written in such a way that it uses the number of threads based on the available cores in the underlying hardware. Once the number of threads is known, the parallel task is divided into that many threads and each thread runs on an individual core.

The sequential and the parallel code with OpenMP are executed on Intel Pentium CPU G630 which has dual cores and also on Intel i7 processor which has dual cores and each core can execute 2 logical threads using Intel’s hyper threading(HTT) technology. For each matrix dimension, the results are taken three times and the average time taken to execute the program is calculated. The main objective here is to get a better performance as the number of cores increase. An overview of this work is shown in Figure 2.

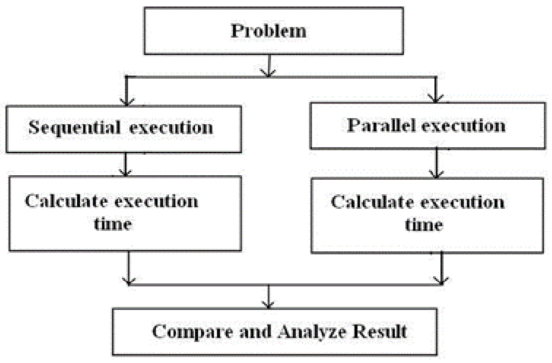

Figure 2. Overview of the proposed work

The sequential and the parallel algorithms are explained in (a) and (b):

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 10, October 2014

Algorithms:

(a). Matrix_Multiplication (int size) (without OpenMP)

Where *size* represents the size of the matrix.

```
Step 1: Declare variables to store allocated memory
Step 2: Declare variables to input matrix size like (m*p) and (q*n)
Step 3: Declare variable to calculate the time difference between the start and end of the execution.
Step 4: Accept number of rows and columns.
Step 5: Allocate dynamic memory for matrix one. {
a = (int *) malloc (10*m)
for ( i=0; i< n; i++ ){
    a [i] = (int *) malloc ( 10*n )
}
Step 6: Allocate dynamic memory for matrix two.
Step 7: Allocate dynamic memory for resultant matrix.
Step 8: Start the timer.
    start = clock ();
Step 9: Initialize first, second and resultant matrix.
  for ( i=0; i<m; i++ ) {
  for ( j=0; j<p; j++ ) {
     a[i][j] = i+j;
  }
  }
Step 10: Do naive matrix multiplication.
   for ( i=0; i<m; i++ ) {
   for(j=0;j<p; j++) {
for(k=0;k<n; k++) {
    c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
    }
   }
Step 11: End the timer.
    end = clock ();
Step 12: Calculate the difference in start and end time.
    diff = ( end – start ) / CLOCKS_PER_SECOND;
Step 13: Free memory for matrix one.
    free ( a );
Step 14: Free memory for matrix two.
Step 15: Free memory for the resultant matrix.
Step 16: Print the time required for program execution.
```

(b). Matrix_Multiplication (int size, int n) (with OpenMP)

Where size represents the size of the matrices and n represents the number of threads.

```
Step 1: Declare variables to store allocated memory
Step 2: Declare variables to input matrix size as m, p, q, n and variables to be used by OpenMP functions as nthreads, tid, and chunk.
Step 3: Declare variable to calculate the starting and ending time for computation.
Step 4: Accept number of rows and columns.
Step 5: Allocate dynamic memory for matrix one.
a = (int *) malloc (10*m){
for ( i=0; i< n; i++ ){
   a[i]=(int *) malloc(10*n)
   }
}
Step 6: Allocate dynamic memory for matrix two.
Step 7: Allocate dynamic memory for the resultant matrix.
Step 8: Start the timer
    double start = omp_get_wtime ()
Step 9: The Actual Parallel region starts here
    #pragma omp parallel shared ( a, b, c, nthreads, chunk )
      private ( tid, i, j, k) {
    tid = omp_get_thread_num ()
    if ( tid == 0 ) {
        nthreads = omp_get_num_threads ()
        printf  nthreads
    }
Step 10: Initializing first matrix.
Step 11: Initializing second matrix.
Step 12: Print Thread starting matrix multiply.
    #pragma omp for schedule ( static, chunk )
    for ( i=0; i<m; i++ ){
        for ( j=0; j<p; j++ ){
            for ( k=0; k<n; k++ ){
                c[i][j]=c[i][j]+a[i][k]*db[k][j]
            }
        }
    }
Step 13: end the timer
    double end = omp_get_wtime ( )
```

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 10, October 2014

Step 14: Store the difference

```
    diff = end – start
```

Step 15: Free memory

```
    for(i=0;i<m; i++){
    free (a)
    }
```

Step 16:

```
    free ( b )
```

Step 17:

```
    free ( c )
```

Step 18: Print the time required for computation.

V. EXPERIMENTAL RESULTS

The sequential and parallel matrix multiplication programs were run on Intel Pentium (two logical threads) and i7 processor (four logical threads) systems and the corresponding time taken for the execution is taken. The run time results are as shown in Table 1 and the corresponding graph is shown in figure 3.

TABLE 1
EXECUTION TIME FOR MATRIX MULTIPLICATION

Data Set (n*n) * (n*n)	Sequential Program	Parallel Program with Two	Parallel Program with Four
500*500	1.44	1.1	1.02
1000*1000	15.40	9.73	8.75
1500*1500	58.75	30.82	22.63
2000*2000	133.71	74.03	52.50
2500*2500	281.11	157.80	111.50
3000*3000	461.24	243.05	217.70
3500*3500	942.47	480.27	330.78
4000*4000	1345.06	726.50	526.04

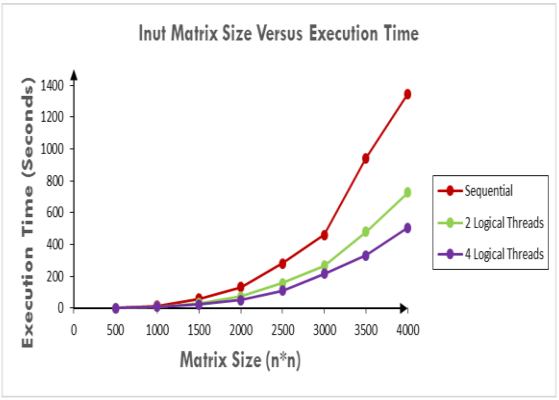

Figure 3: Performance Analysis of Matrix Multiplication Algorithm

The speedup achieved with two logical threads and four logical threads is shown in Table 2 and the corresponding graph is s3hown in Figure 4.

TABLE 2
SPEEDUP CHART FOR MATRIX MULTIPLICATION

Data Set	Two Cores	Four Logical Processors
500*500	1.31	1.41
1000*1000	1.58	1.76
1500*1500	1.91	2.60
2000*2000	1.81	2.55
2500*2500	1.78	2.52
3000*3000	1.72	2.12
3500*3500	1.96	2.85
4000*4000	1.85	2.67

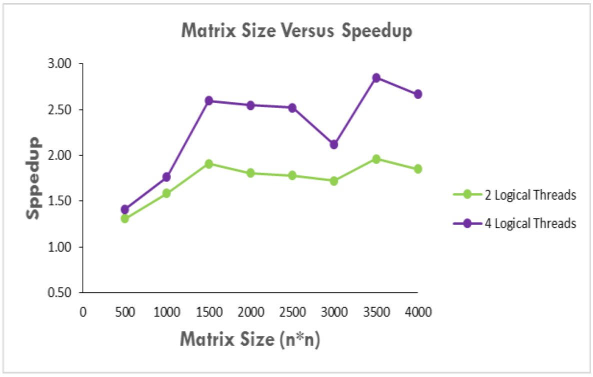

Figure 4: Speedup Graph for Matrix Multiplication

VI. CONCLUSION AND FUTURE WORK

From the results obtained from the experimental analysis, the matrix multiplication algorithm with OpenMP performs better than the sequential algorithm. The maximum speedup achieved with two cores is 1.96 which is almost twice the speed of the execution with sequential algorithm and with four logical processors is almost three times. This clearly indicates that as the number of cores increase, the computation time taken by an algorithm is also less. This analysis is done on a small data set. As the matrix size becomes large and as the number of cores increase, parallel programs written with OpenMP gives much better performance.

Once the application is parallelized using OpenMP, it can be still improved using Intel's Vtune Amplifier tool. The applications written in OpenMP can be further analysed for fine tuning by hotspots analysis provided by the tool.

The tool also gives hardware level details like cache performance, individual core analysis etc. which helps the application developer to improve the algorithm and also the performance.

REFERENCES

[1] R. Ramanathan. Intel multi-core processors: Making the move to quad core and beyond. Technology@Intel Magazine, Dec 2006.

[2] Sodan, A.C, Machina J, Deshmeh A, Macnaughton K, "Parallelism via multithreaded and Multicore CPUs", IEEE Computer Socoety, Volume: 43, issue: 3, pp. 24-32, Mar. 2010.

[3] D. Geer, "Chip Makers Turn to Multicore Processors," IEEE Computer Society, vol. 38, pp. 11-13, May. 2005.

[4] Ayguade, E. Copty, N., Duran, A., Hoeflinger, J. "The Design of OpenMP tasks", IEEE Transactions on Parallel and Distributed systems, volume: 20, Issue: 3, pp. 404-418, June 2008.

[5] Noronha R. and Panda D.K, "Improving Scalability of Open MP Applications on Multi-core Systems Using Large Page Support", IEEE Computer, 2007.

[6] Paul Graham, Edinburgh, "A Parallel Programming Model for Shared Memory Architectures", *Parallel Computing Center,* The University of Edinburgh, March 2011.

[7] D. Dheeraj, B. Nitish, Shruti Ramesh, "Optimization of Automatic Conversion of Serial C to Parallel OpenMP", *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discover,* PES Institute of Technology Bangalore, India, Dec 2012.

[8] J. Breckling Sanjay Kumar Sharma, Dr. Kusum Gupta, "Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP Programming Approaches", *International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol.2, No.5,* October 2012.