Rahul Dadhich *et al*, International Journal of Computer Science and Mobile Applications,
Vol.5 Issue. 10, October- 2017, pg. 92-99 **ISSN: 2321-8363**
Impact Factor: 4.123

EFFECTIVENESS OF OPENMP FOR ALGORITHM PARALLELIZATION

Rahul Dadhich[1], Prakash Choudhary[2], Neha Mahala[3], P K Bhagat[4]

[1]Samsung India Electronics Pvt Ltd, Bangalore, India
[2]National Institute of Technology Manipur, India
[3]ISM Dhanbad, India
[4]National Institute of Technology Manipur, India pkbhagat22@gmail.com

Abstract: Today in the market, highly efficient, scalable and fast processors are available. This was all about the hardware perspective. But the software markets have not scaled up in the similar fashion. To scale up software efficiency, OpenMP tried to offer a shared memory parallel programming model. OpenMP (Open Multiprocessing) is an Application Program Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP is not a new computer language; it works in conjunction with either standard FORTRAN or C/C+ +. This paper illustrates the basic concepts of parallel computing with a brief overview of OpenMP. The paper also describes an analysis of algorithms from different fields like LU decomposition, PI Value using Monte Carlo method. The observations and results obtained show that how the usage of OpenMP's Pragma are effective in the normal C/C++ programs and how the result varies according to the inputs and available number of threads and shows that it is useful only when we are working on large data set or large computations are involved in the given problem.

Keywords: OpenMP, Parallel programming, LU decomposition, Monte Carlo.

1. Introduction

Parallel computer programs are difficult to write as compared to sequential ones because, various potential software bugs are to be avoided e.g. data dependencies, race conditions etc. Communication and synchronization between the different subtasks are typically most challenging parts, in order to achieve good parallel program performance. Multi-core is simply a popular name for Chip Multiprocessors (CMPs) or single chip multiprocessors. The concept of single chip multiprocessing is not new, and chip manufacturers have been exploring the idea of multiple cores on a uni-processor since the early 1990s.

In single core configurations there is one general purpose processor, although it is important to note that many of today's single core configurations contain special graphic processing units, multimedia processing units, and sometimes special math coprocessors. But even with single core or single processor computers multithreading, parallel programming, pipelining, and multiprogramming are all possible [6].

Single-core processors are really only able to interleave instruction streams, but not execute them simultaneously, the overall performance gains of a multi-threaded application on single-core architectures are limited. On these platforms, threads are generally seen as a useful programming abstraction for hiding latency. This performance restriction is removed on multi-core architectures.

In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application [5]. A thread is a discrete sequence of related instructions that is executed independently of other instruction sequences. In a program there is at least one thread called main thread, which, furthermore, can create other threads. On the other hand, at hardware level, thread is an

execution path that remains independent of other hardware execution paths. To take advantage of multi-core processors, knowledge of details of software threading model as well as capabilities of the platform hardware is necessary [4]. Rest of the paper as follows. Section 2 describe the OpenMp constructs. Section 3 presents the experimental results followed by conclusion in section 4.

2. OpenMP

OpenMP (Open Multiprocessing) is an Application Program Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism [1]. It is, basically, a portable API specified for C/C++ and FORTRAN and a standard for the programming of shared memory systems [7]. OpenMP is an implementation of multithreading, a method of parallelization whereby the master thread forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The programming model of OpenMP is based on cooperating threads running simultaneously on multiple processors or cores. Thus, the OpenMP program begins with a main thread or master thread. Slave threads in the program are created and destroyed in a *fork-join* pattern, as shown in figure 1.

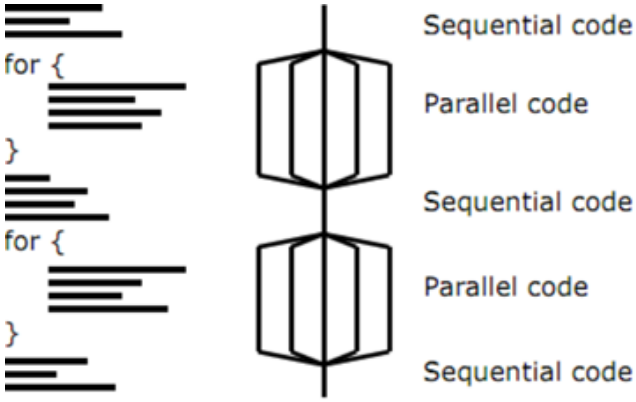

Figure 1. Fork/join model in OpenMP [2].

When the parallel construct is encountered, the initial thread, as a master thread creates a team of threads consisting of a certain number of new threads and the initial thread itself. This fork operation is performed implicitly. The program code inside the parallel construct is called as a parallel region and is executed in parallel by all threads of the team. At the end of a parallel region, there is implicit barrier synchronization, and only the master thread continues to execute after this region (implicit join operation). Considering the case of memory caching, each processor core may have its own cache. At any point in time, the cache on one processor core may be out of sync with the cache on the other processor core, figure 2. Considering a single-core platform, there is only one cache shared between threads; therefore, cache synchronization is not an issue [3].

OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them. Sometimes, however, one needs variables that have thread-specific values. When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is private.

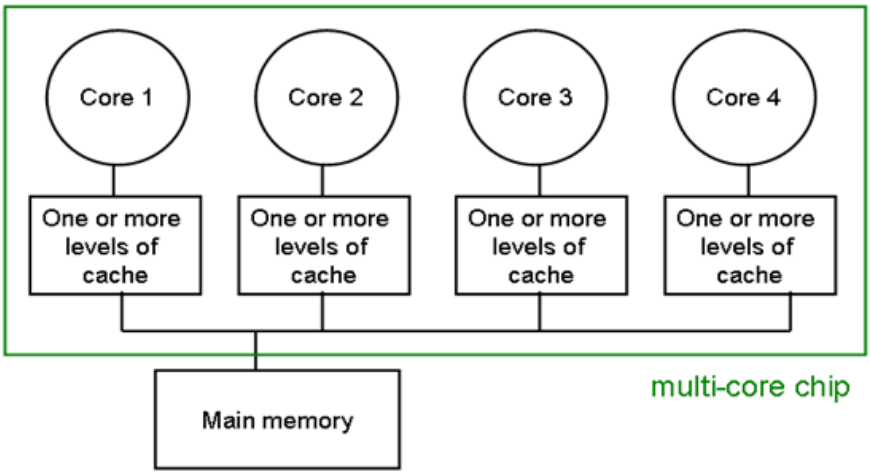

Figure 2. Block Diagram of a Multi-core Chip [10].

3. Experimental Analysis

If we denote by *T1* the execution time of an application on *1* core (sequential execution), then in an ideal situation, the execution time on *P* cores should be *T1 /P*. If *TP* denotes the execution time on *P* cores, then the ratio *S*, equation (1), is referred to as the parallel speedup and is a *measure for the success of the parallelization*.

$$S = T1 / TP \tag{1}$$

Virtually all programs contain some regions that are suitable for parallelization and other regions that are not. By using an increasing number of cores, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same. Eventually the execution time is completely dominated by the time taken to compute the sequential portion, which puts an upper limit on the expected speedup. This effect, known as Amdahl's law, can be formulated using equation (2).

$$S = 1 / (fpar / P + (1 - fpar)) \tag{2}$$

Where *fpar* is the parallel fraction of the code and *P* is the number of processors [8].

3.1 LU Decomposition

In linear algebra, LU decomposition (also called LU factorization) factorizes a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. LU decomposition is a key step in several fundamental numerical algorithms in linear algebra such as solving a system of linear equations, inverting a matrix, or computing the determinant of a matrix.
If you we assume that we have a matrix Ax = b and we have some matrix A = LU then (LU)x = B
L(Ux) = b
Ly = b
Ux = y

The idea is creating each element of the L matrix then proceeds with Gaussian elimination on matrix A. The upper
Triangular of matrix A will be U [9].
$L_{2,1} = a_{2,1}/a_{1,1}$
$a_{2,2} = a_{2,2} - L_{2,1}*a_{1,2}$
$a_{2,3} = a_{2,3} - L_{2,1}*a_{1,3}$

$a_{2,n} = a_{2,n} - L_{2,1} * a_{1,n}$
$L_{3,1} = a^{3,1}/a_{2,2}$
etc..

Given: -A square matrix a of size N X N
Aim: - Find lower, upper triangular matrix L, U such that a=LXU.
Algorithm:-

a) k = 1 to N do
L[k][k] = 1. //set all diagonal element as 1 in lower triangular matrix
b) i = k+1 to N do //assigning values to non diagonal elements
L[i] [k]= a [i] [k] / a [k] [k]
c) j = k+1 to N do .
a [i] [j]= a [i] [j]-L[i] [k]* a [k] [j] //set of row operations
d) j = k to N do
U[k] [j]= a [k] [j] //values to upper triangular matrix

For Speedup of the program-

After analyzing the data dependency of the problem we can say that the values of one column are independent of modification of matrix "a" in particular iteration so we can parallelize by each column. Apply # pragma on k loop in step a.

```
#pragma omp parallel shared(l,u,a,c,chunk,matsize) private(k,j,i)
        {
   #pragma omp for schedule (static, chunk)
   for(k=0;k<matsize;k++)
        {
          l[k][k]=1;
     for(i=k+1;i<matsize;i++)
        {
                        l[i][k]=a[i][k]/a[k][k];

                                for(j=k+1;j<matsize;j++)
                                {
                                        a[i][j]=a[i][j]-l[i][k]*a[k][j];

                                }

        }
     for(j=k;j<matsize;j++)
        {
                u[k][j]=a[k][j];
                }
        }
}
```

Given a square matrix A of size N X N, the aim is to find lower, upper triangular matrix L, U such that a=L×U. We have experimented with 4 different matrices and the observed speed up is shown in figure 3. The best speed up time for LU decomposition with N=500 is 2.080099 with 21 number of threads and LU decomposition with N=1000 is 3.947454 with 10 number of threads. The best speed up time LU decomposition with N=2000 is 6.329494 with 20 number of threads and LU decomposition with N=3000 is 6.097029 with 22 number of threads.

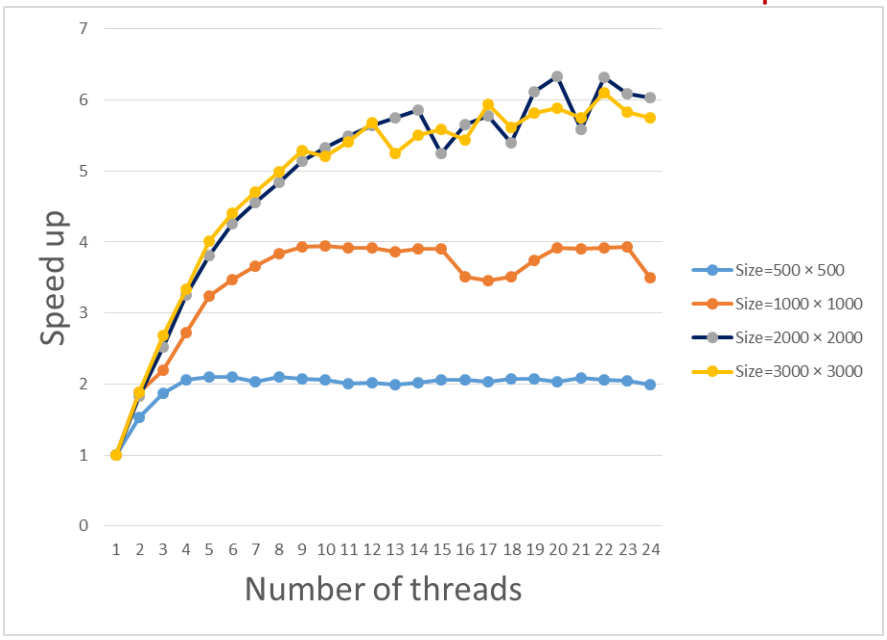

Figure 3. Observation of speed up time for LU decomposition with different matrix size.

3.2 PI Value Using Monte Carlo Method

Monte Carlo methods provide approximate solutions to a variety of mathematical problems by performing statistical sampling experiments. They can be loosely defined as statistical simulation methods, where statistical simulation is defined in quite general terms to be any method that utilizes sequences of random numbers to perform the simulation.

This process involves performing many simulations using random numbers and probability to get an approximation of the answer to the problem, and PI value calculation is one of them which calculated on random numbers.

Monte Carlo Techniques

Monte Carlo technique is consists following approaches

1. Crude Monte Carlo
2. Acceptance - Rejection Monte Carlo
3. Stratified Sampling
4. Importance Sampling

In PI value calculation Acceptance - Rejection Monte Carlo approach is used, figure 4.

In this technique we enclose the interval of given function in a rectangle. Now begin taking random points within the rectangle and evaluate this point to see if it is below the curve or not. If the random point is below the curve then it is treated as a successful sample. Thus, if we take N random points and perform this check, remembering to keep count of the number of successful samples there have been. Now, once we have finished sampling, we can approximate the integral for the interval (a, b) by finding the area of the surrounding rectangle. You then multiply this area by the number of successful samples over the total number of samples, and this will give you an approximation of the integral for the interval (a, b).

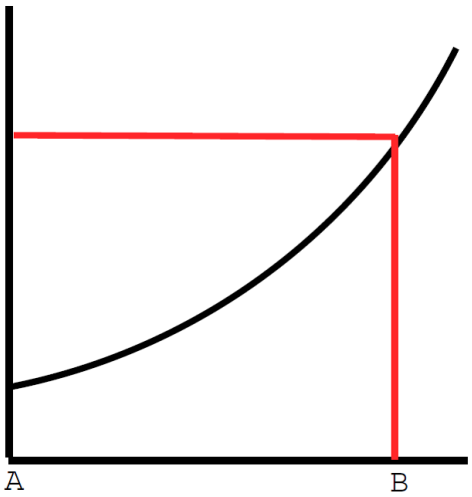

Figure 4. Acceptance-Rejection Monte Carlo method [11].

Given: - few number of sample points.
Aim: - To calculate the value of PI.

a) Repeat steps b to d of all samples
b) Take the sample point as x and y.
c) Calculate the value of z as z= x*x +y*y
d) Check the value of z, if it is less then or equals to 1 then increment the value of count
e) After testing of all sample point PI=count/(niter*4)

For Speedup of the program-

After analyzing the problem we found that there is no data dependency in finding the z value among all samples. So on applying #pragma omp in for loop, which calculate the z value for given number of samples Hence, multiple threads can compute assigned chunks in parallel and as on for maximum values.

```
#pragma omp parallel private(i,x,y,z)
{
#pragma omp for schedule (static, chunk)
  for ( i=0; i<(niter*s); i++)
{
    x = (double)rand()/RAND_MAX;
     y = (double)rand()/RAND_MAX;
     z = x*x+y*y;
     if (z<=1)
#pragma omp critical
                        count++;
}
}
```

Given a few number of sample points, our aim is to calculate the value of PI. We have experimented with 8 different iterations, see table 1, and the observed speed up is shown in figure 5.

Table 1. Number of iterations and corresponding speed up time.

No of Iterations	Time(Sequential)	Time(Parallel)	Speedup
10000	0.001923	0.001495	1.285647
20000	0.004056	0.002761	1.468793

50000	0.009268	0.005737	1.615385
100000	0.018829	0.011108	1.695055
200000	0.037384	0.022639	1.671214
300000	0.044128	0.033324	1.324176
400000	0.058777	0.044677	1.315574
500000	0.068176	0.055877	1.220098

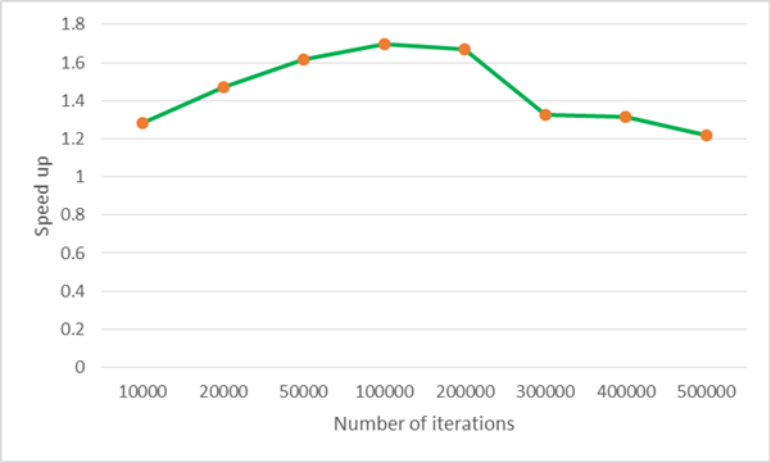

Figure 5. Observation of speed up time for PI value with different number of iterations.

3. Conclusion

As the program runs on HP Xeon Work Station, Speedup increases as the number of thread increases for same input set. In various observation we find that the speedup is slightly decrease as we increase no of thread to 7 from 6, this is because it is a dual processor machine with 6 cores a processor. If we talk about different set of inputs (increased input set), irrespective of the number of cores available, the speedup is large for the case where input is large as well as the computation is large. As the number of threads increases, the OpenMP parallel time stabilizes and is almost same as the number of thread increases. Implied barrier also affect the performance of parallelization because in the reduction clause an implied barrier is there which finally halts for combining the solution space. By the results for schedule clause we can say that the performance with maximum number of threads are same but using dynamic schedule we can get maximum speed in less no of threads.

References

[1] www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf
[2] S. Hirsch, U. Finkler, 2013, To Thread or Not to Thread, *In: IEEE Design & Test, 30(1), pp. 17-25.*
[3] S. Akhter, J. Roberts, 2006, Multi-Core Programming Increasing Performance through Software Multi-threading, *Intel Press*.
[4] C. Hughes, T. Hughes, 2010, Professional Multicore Programming Design and Implementation for C++ Developers, *Wiley*.
[5] T. Rauber, G. Runger, 2008, Parallel Programming for Multicore and Cluster Systems, *Wiley Publishing, 2nd edition.*
[6] https://www3.nd.edu/~zxu2/acms60212-40212/Lec-06.pdf.
[7] OpenMP Manual: *http://www.openmp.org*.
[8] B. Chapman, G. Jost, R. V. D. Pas, 2008, Using OpenMP: Portable Shared Memory Parallel Programming, *The MIT Press, Cambridge*.
[9] P. D. Michailidis, K. G. Margaritis, 2010, Implementing Parallel LU Factorization with Pipelining on a MultiCore using OpenMP, *13th IEEE International Conference on Computational Science and Engineering*.

[10] Jernej Barbic, 2007, Multi-core Architecture, *class lecture slides, Introduction to Computer Systems*, 15-213.
[11] Jonathan Pengelly, 2002, Monte Carlo Method, http://reflect.otago.ac.nz/cosc453/student_tutorials/monte_carlo.pdf.

P K Bhagat is a PhD research scholar at National Institute of Technology Manipur, India. He obtained Mtech form NIT Manipur, India (2017) and Master of Computer Application (MCA) from Indra Gandhi National Open University, India (2013). His research area includes pattern recognition, artificial intelligence, feature extraction, deep learning, image processing, bio-medical image processing, image classification, image retrieval.
Email: pkbhaagt22@gmail.com

