STUDY OF PARALLELIZATION OF ALGORITHMS USING OPENMP

Rahul Dadhich¹, Neha Mahala², Prakash Choudhary³, P K Bhagat⁴

¹Samsung Company, Bangalore, India.
²ISM Dhanbad, India
³National Institute of Technology Manipur, India
⁴National Institute of Technology Manipur, India. pkbhagat22@gmail.com

ABSTRACT:

In this era, we all look forward to achieve high performance for our wide field of computational requirements. 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 is an emerging standard for parallelizing programs in a shared memory environment. It provides a set of pragmas for programmers to parallelize their code. This paper explains the concept of parallel processing and the concept of OpenMP and how it is used in the C/C++ programs for effective utilization of available no. of cores-processors. 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 Matrix Multiplication, Saddle point, Cholesky decomposition. 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, Matrix multiplication, Cholesky decomposition, Saddle point.
INTRODUCTION

Parallel programming and design of efficient parallel programs have been well established in high performance, scientific computing for many years. In parallel computing, calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are solved in parallel. 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. There are several different forms of parallel computing: bit-level, instruction level, data, thread level, task parallelism.

A multi-core is an architecture design that places multiple processors on a single die (computer chip). Each processor is called a core. As chip capacity increased, placing multiple processors on a single chip became practical. These designs are known as Chip Multiprocessors (CMPs) because they allow for single chip multiprocessing. These days CMP has become the preferred method of improving overall system performance. This is a departure from the approach of increasing the clock frequency or processor speed to achieve gains in overall system performance. Increasing the clock frequency has started to hit its limits in terms of cost-effectiveness.

In order to achieve optimal application performance on multi-core architectures one must effectively use threads to partition software workloads. Multi-threaded applications running on multi-core platforms have different design considerations than do multi-threaded applications running on single-core platforms. 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 [3].

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]. Threading can enhance performance by making better use of hardware resources, if properly implemented. 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].

The design of parallel algorithm or program for a given application problem starts with decomposition of computations of an applications into several parts, called as tasks, which are independent to each other, and identifying tasks that can run independently of each other in parallel is a challenging work. OpenMP is suited for shared memory systems like we have on our desktop computers. Shared memory systems are systems with multiple processors but each are sharing a single memory subsystem. Using OpenMP is just like writing your own smaller threads but letting the compiler do it. Available in Visual Studio 2008 Professional and Team Suite. Rest of the paper as follows. Section 2 describe the OpenMP constructs. Section 3 presents the experimental analysis followed by conclusion in section 4.

OpenMP (Open Multiprocessing) is an Application Program Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism [1]. Pioneered by SGI and developed in collaboration with other parallel computer vendors, OpenMP is fast becoming the de facto standard for parallelizing applications. OpenMP provides a collection of compiler directives, library routines and environmental variables. It has been designed to introduce parallelism in existing sequential programs. 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]. The OpenMP provides various library routines for avoiding deadlocks, conflicts and race conditions. At compile time, multi-threaded program code is generated based on the compiler directives. Various compilers have a support for OpenMP standard [11].

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

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. The fork-join programming model is supported by OpenMP, [Figure-2]. The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.
STUDY OF PARALLELIZATION OF ALGORITHMS USING OPENMP

Figure: 2. OpenMP Programming Model [8].

For C and C++ programs, pragmas are provided by the OpenMP API to control parallelism. In OpenMP these are called directives. They always start with #pragma omp, followed by a specific keyword that identifies the directive, with possibly one or more so-called clauses, each separated by a comma. These clauses are used to further specify the behavior or to further control parallel execution, e.g., #pragma omp directive-name [clause [[,] clause]. . . ]. OpenMP also provides construct, an OpenMP executable directive and the associated statement, loop, or structured block, if any, not including the code in any called routines, that is, the lexical extent of an executable directive. A large number of applications can be parallelized by using relatively few constructs and one or two of the functions, see [figure-3].

Figure: 3. OpenMP Constructs.

[3] THE EXPERIMENTS

If we denote by $T_1$ the execution time of an application on 1 core (sequential execution), then in an ideal situation, the execution time on $P$ cores should be $T_1 / P$. If $T_P$ 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.

$$S = \frac{T_1}{T_P}$$
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).

\[ S = \frac{1}{(f_{\text{par}} / P + (1 - f_{\text{par}})} \]  

(2)

Where \( f_{\text{par}} \) is the parallel fraction of the code and \( P \) is the number of processors. But there are various obstacles along the way to perfect linear speedup are the overheads introduced by forking and joining threads, thread synchronization, and memory accesses. On the other hand, the ability to fit more of the program’s data into cache may offset some of the overheads. A measure of a program’s ability to decrease the execution time of the code with an increasing number of cores is referred to as parallel scalability [6].

A cost is associated with the creation of OpenMP parallel regions, with the sharing of work among threads, and with all kinds of synchronization. The actual overheads experienced by an application depend on the OpenMP translation strategy used by the compiler, characteristics of the runtime library routines and the way they are used, the target platform, and the way the compiler otherwise optimizes code. In most cases, however, these are relatively modest compared to the cost of barriers and other forms of thread synchronization, as well as the loss in speedup whenever one or more threads are idle. A good OpenMP implementation takes care to use the most efficient means possible to create, deploy, and synchronize threads. All the experiments are carried out on HP Z600 Xeon architecture.

### [3.1] MATRIX MULTIPLICATION

Given two matrices A and B with \( n \times p \) and \( p \times q \) dimension we want to find a matrix C with dimension \( n \times q \). We have experimented with 4 different matrices and the observed speed up is shown in [Figure-4]. The best speed up time for a matrix multiplication with \( N=1000 \) is 11.13427 with 24 number of threads and for a matrix multiplication with \( N=2000 \) is 13.48535 with 24 number of threads. The best speed up time for a matrix multiplication with \( N=3000 \) is 12.88873 with 23 number of threads and for a matrix multiplication with \( N=4000 \) is 14.2047 with 24 number of threads.
[3.2] CHOLESKY DECOMPOSITION

The Cholesky decomposition is mainly used for the numerical solution of linear equations $Ax = b$. If $A$ is symmetric and positive definite, then we can solve $Ax = b$ by first computing the Cholesky decomposition $A = LL^T$, then solving $Ly = b$ for $y$, and finally solving $L^Tx = y$ for $x$. Every symmetric, positive definite matrix $A$ can be decomposed into a product of a unique lower triangular matrix $L$ and its transpose:

$$A = LL^T$$

Where $L$ is called the Cholesky factor of $A$, and can be interpreted as a generalized square root of $A$, as described in Cholesky decomposition [9].

Given a $n \times n$ real, symmetric, and diagonally dominant matrix $A$ our aim is to convert it into a lower triangular matrix $L$ such that $A = LL^T$. We have experimented with 4 different matrices and the observed speed up is shown in [Figure-5]. The best speed up time for a Cholesky decomposition with $N=1000$ is 7.746094 with 24 number of threads and for a Cholesky decomposition with $N=2000$ is 8.688956 with 18 number of threads. The best speed up time for a Cholesky decomposition with $N=3000$ is 8.696612 with 24 number of threads and for a Cholesky decomposition with $N=4000$ is 8.587287 with 24 number of threads.
[3.3] SADDLE POINT

A matrix is said to have a saddle point [10] if some entry $a[i][j]$ is the smallest value in the $i$'th row and the largest value in the $j$'th column. A matrix may have more than one saddle point. Saddle point concept is mainly used branch of mathematics called Game Theory, which is concerned with analyzing games.

Given a matrix $A$ with $p \times q$ dimension, our aim is to find a saddle point in given matrix. We have experimented with 4 different matrices and the observed speed up is shown in [Figure-6]. The best speed up time for finding a saddle point with $N=1000$ is 7.099264 with 12 number of threads and for finding a saddle point with $N=2000$ is 8.174935 with 22 number of threads. The best speed up time for finding a saddle point with $N=3000$ is 8.970795 with 19 number of threads and for finding a saddle point with $N=4000$ is 11.054689 with 24 number of threads.
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. Time for the execution of a program take more time on m core architecture as compared to n core architecture where (m<n),while using same number of threads. Speedup also depends on how much portion of the program is parallelized. 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