Multicore Programming on Small-Scale Systems

R. Johari[1] and M. Othman[2]
[1]Department of Interdisciplinary Studies, University College, Zayed University, Dubai, United Arab Emirates
[2]Department of Communication Technology and Network, Universiti Putra Malaysia, 43400 UPM Serdang, Selangor D.E., Malaysia

Abstract – *Parallel programs previously implemented on complex parallel computers resulted in small number of software developers programming on these computers except for those who understand the application domain and had resources and skill to program on this platform. Most general-purpose computers today having multicore processors and are parallel in architecture. Hence, software developers can cost-effectively take advantage of implementing multicore multithread parallel applications gaining benefits in computing power. This study explores shared memory multicore multithread programming with OpenMP in small scale computers. Laptops with Unix and Linux based platform, Mac OS X and Ubuntu are used to run a popular regular program, matrix multiplication to show the improvement in performance. OpenMP design issues including threading parallelized loop, loop scheduling and partitioning, and sharing and declaring data or memory is emphasized. This can motivate software developers implementing multicore multithread parallel programs in smaller-scale systems and advance to specialized hardware as requirements grow.*

Keywords: Multicore, multithread, OpenMP, shared memory programming, small-scale system

1 Introduction

Multicore and multithreading processing has been the trend of parallel computing today. This resulted from the ubiquity of multicore processors in all general-purpose computers. With newer and faster multicore multithreading processor designs making processing power more affordable and manageable by novice software and application developers when creating their programs from scratch. Many shared memory based parallel algorithms were tested on complex parallel architecture. In [1], the authors implemented parallel Gaussian Elimination on IBM RS/6000 SP machine. Authors in [2] discussed parallel matrix-multiplication algorithm on Intel Server System SR1670HV and Intel Server System SR1600UR. [3] implemented Genehunter genetic program on Compaq Alpha Tru64 systems. [15] implemented parallel Boundary Integral Method on Sequent Symmetry 5000 SE30. Most of the implementations claimed improvement in performance but none of the algorithms can be clearly claimed achieving optimal performance. Hence, these developers need to switch using complex and expensive parallel computers to small scale computer since less complex and easier to manage. Using these general-purpose computers running parallel applications are effortlessly as compare to using parallel computer. Advanced scientific and engineering communities have long used parallel computing solving large-scale computer problems on parallel computer, but these scientists find it hard to implement parallel applications effectively due to the complexity of the system. Moreover, the nature of scientific research have become more complicated and scientists often required to write their own program from scratch and many scientists are not well-versed to write the software. In fact, most scientists are not well trained using complex machine because of their limited exposure to high tech computational tools [4]. This is also strengthened by Uzi Vishkin [5] viewpoint mentioning that innovation in high-end general-purpose desktop application has been minimal due to less production of the hardware. Therefore, portable general-purpose computers such as laptop and desktop replacement computer is very promising for scientist, software and application developers exploring performance of multicore multithread processors.

Many shared memory parallel programming models have been introduced to support the development of parallel applications. Generally, these programming models are based on either language enhancement or run-time libraries [6]. Intel Threading Building Block [7], Microsoft Task Parallel Library [8] and Oracle Parallel Framework Project [9] are several innovations in using run-time libraries to enhance parallelism. Nevertheless, OpenMP [10,11] language enhancement programming model is used to explore the parallel application development due to it able to provides cross-vendor portability and straightforward directive-based approach. OpenMP supported large number of compilers and can be implemented in major platforms including Unix/Linux based platforms and Windows. It can be run on single processor or multiple processors in a shared memory architecture taking the advantage of today general multicore processors. Likewise, OpenMP chosen since it required less programming effort and allowing incremental parallelization on the program solved.

The focused of this paper is to examine the major constructs of OpenMP and discuss the parallel design issues of the parallel programming model on both Mac OS X and Ubuntu. Since both operating systems built on Unix/Linux foundation make it easy for developer to explore other development technologies when required. OpenMP is a large powerful API

for writing portable, multithreaded applications and provides an easy way threading applications. The OpenMP programing model provides a platform-independent set of compiler directives, runtime library routines, and environment variables that explicitly channelize parallelism in the application. The power and simplicity of OpenMP will be demonstrated by going through a matrix multiplication application. Matrix multiplication is used as a case study since many applications covering nearly all subject areas used matrix multiplication as building block. The implementation shown reduced execution time when running using OpenMP on both Unix and Linux based platforms. This could motivate developers implementing shared memory parallel applications in smaller-scale systems.

This paper presents a brief overview of OpenMP shared memory model and introducing the basic features of OpenMP. The matrix multiplication programming using OpenMP on Mac OS X and Ubuntu is discussed and the performance results obtained on both platforms are presented. Finally, a summary of the results is concluded.

2 Overview of OpenMP

OpenMP standard was formulated as an API for writing portable, multithreaded application and consist of a set of compiler directives, runtime library routines and environment variables. Most of the compiler directives are indicated with constructs expressed by a set of compiler pragmas, which is the basis of parallelism in OpenMP. The pragmas used to spawn parallel region, distribute codes among threads, distributing loop iterations between threads, serializing sections of code and synchronizing work among threads. The run-time routines create flexibility in setting and retrieving information about the environment such as setting and querying the number of threads, querying thread ID and querying wall clock time. Withal, several OpenMP environment variables can be used to control the execution of parallel code at run-time.

OpenMP is supported on most widely used native programming languages such as Fortran, C and C++. It offers a common specification that lets programmers easily design new parallel application or parallelize existing sequential applications to take advantage of multicore systems configured with shared memory. Portability is an important characteristic of OpenMP. Any compiler that supports OpenMP can be used to compile parallel application source code that is developed using OpenMP. The resulting compiled binary should be run on the target hardware platform to achieve parallel performance.

3 OpenMP Basic Features

This section summarized OpenMP basic features and it is essential in providing fundamental understanding of threading with OpenMP from which a broader practical knowledge could be achieved. Achieving an optimal performance in a multithreaded application using OpenMP required understanding important issues mainly related to threading parallelized loop, loop scheduling and partitioning, and sharing and declaring data or memory.

Loops threading uses work-sharing, a term OpenMP uses in distributing work across threads. Parallel loops are the most common parallelizable work-sharing construct and OpenMP provide pragma construct *#pragma omp parallel for*. The construct converts independent loop iterations to threads and run these threads in parallel. Threading parallelized loop required no loop-carries dependencies. This mean one iteration of the loop does not depend upon the results of another iteration of the loop. Loop-carried dependence can be in a form of data dependence. Data dependencies can exist in three different forms: flow dependence, output dependence and anti-dependence. Data-race condition due to output dependencies commonly existed in which multiple threads updating the same memory location or variable after threading. In such condition, the code needed to be modified by privatization or synchronization for example adding a private clause. Since loops are the sections of codes having most execution that can be parallelized, effectively scheduling and partitioning the loops is the major contributions in achieving optimal performance.

Achieving optimal performance through good load balancing required an efficient loop scheduling and partitioning. To balance the workload among the threads, OpenMP provide four scheduling schemes: static, dynamic, guided, and runtime. In static scheduling approach, iterations of a loop are distributed among the threads in equal number of iterations. With *m* iteration and N threads, each thread gets *m*/N iterations. When *m* not evenly divisible by N, the compiler and runtime library will handle the case. By default, parallel for loop uses static scheduling. In dynamic scheduling, loop iterations are divided into pieces of size chunk and handled with the first-come, first-serve scheme. The chunks dynamically schedule among the threads. When a thread finishes executing one chunk, it is dynamically assigned another chunk until all the chunks completed. The last set of iterations may be less than the chunk size. Guided scheduling is like dynamic scheduling with the exceptional of the chunk size decreases each time remaining unscheduled loop iteration given to a thread. As for runtime scheduling, the scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE which gives end-user the flexibility in selecting the type of scheduling dynamically among the three previously mentioned. Choosing the right scheduling scheme avoid false-sharing during runtime, and lead to good load balancing.

Sharing and declaring data or memory efficiently is extremely important not only for optimal performance but program correctness. Indicating a piece of memory is shared meaning the thread access the exact same memory location and private indicating the variable is made for each thread to access in private. All variables in a parallel region are shared by default with three exceptions. First, loop index for parallel for

loop is private. Second, all variables local to the parallel region are private. And third, all variables listed through a set of clauses private, firstprivate, lastprivate or reduction are private. All the four clauses take a list of variables, but their semantics are different.

4 OpenMP Matrix Multiplication

Parallel matrix multiplication is always a challenging task for programmer due to its extensive computation and memory requirement. Challenging enough for a dense matrix multiplication and become more challenging when it is large number of floating point dense matrices. With most today general-purpose processors have built-in parallel computational capacity in form of cores and threads, the parallel matrix multiplication can easily be revisited to fully utilize the available multicore multithread and get the maximum efficiency and the minimum execution time. Since the study focusing on parallel matrix multiplication design issues in smaller-scale systems, the implementation will use standard matrix multiplication example to simplify the process of identifying the issues. Many studies charting the performance of standard sequential matrix multiplication algorithm shows drops in performance especially on the multiplication of large matrices. This leads to number of efforts implementing several versions of matrix multiplication in parallel. These implementations run on varieties of parallel architectures. In [12], the authors implemented on hypercube architectures, Thinking Machine Corporation's CM-5 with 512 processors, [13] implemented on distributed-memory concurrent computer, Intel Paragon Computer with both 512 nodes and 256 nodes. Both [12] and [13] implemented the matrix multiplication algorithms on massively parallel computers. While in [14], the authors implemented on shared memory multicore processor system.

This study will discuss OpenMP matrix multiplication implementation on Unix and Linux based platforms in a simplistic way for novice parallel software and application developers to understand and grasp the concepts easily.

4.1 Experimental Setup

The algorithm implemented on Intel Pentium Dual-Core (2 cores with 2 threads, 2.16 GHz) with Ubuntu and Intel Core i5(2 cores with 4 threads), 2.5 GHz) with Mac OS X. OpenMP comes pre-installed with all gcc compiler. Ubuntu includes gcc compiler and Xcode 7.3 downloaded on Mac OS X includes the clang-gcc compiler (gcc 5.3.0).

4.2 Programming with OpenMP

The straight forward implementation of sequential matrix multiplication is the product C=A*B given A is n*l matrix with the element a_{ij}, B is l*m matrix with element b_{ij} and the product C is n*m matrix with elements c_{ij}. The product C is computed as a dot product of row *i* in A with column *j* in B. Mathematical definition given by,

$$c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{kj}$$

Fig. 1 illustrates diagrammatically the product **C** of two matrices **A** and **B**, showing how each intersection in the product matrix **C** corresponds to a row of **A** and a column of **B.**

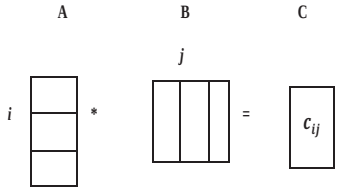

Fig. 1. Product C is row A multiply column B

Consider the 2-by-3 matrix A and 3-by-3 matrix B using the dot product computation as shown in Fig. 2 below

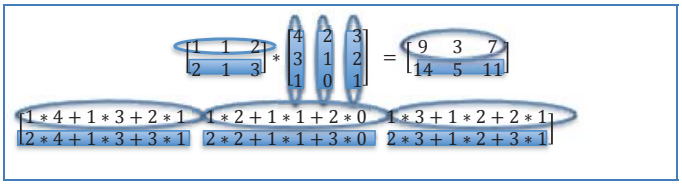

Fig. 2. Example of matrix A multiply matrix B

The problem is simple mathematically but very rich from computational point of view. In the dot product of AB, matrix A partition in row and matrix B partition in column giving a row partitioning of C. The sequential code segment in C produces below.

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      for (k=0; k<l; k++) {
         C[i][j] += A[i][k]*B[k][j];}}}
```

Many studies mentioned the performances of the sequential matrix multiplication are mostly affected by the inefficiency of accessing the memory especially when implemented in C. In C, elements in a row are stored in consecutive memory locations. Thus, accesses to matrices A and C are efficient but not B. As the sizes of matrices getting bigger, the processor will spend more time for memory accesses which could leads to longer execution time.

Parallel implementation of the algorithm is always the solution to performance. The parallel implementation with OpenMP required the understanding of the major constructs of the programming model. From the understanding, major design issues implementing using OpenMP can be determined. Parallelism can be accomplished through the use OpenMP components such as compiler directives, runtime library routines and environment variables. Fig. 3 and Fig. 4

show a simple conversion of the sequential program to OpenMP parallel version using directives and run-time library routines.

Fig. 3 show the sequential code of matrix multiplication comprising of Loop1, Loop2 and Loop3 initializing the matrices A, B and C respectively and can be computed independently by different threads. Loop4 is computing the product of C and each elements of the product C can be threaded independently. *Clock* function is used to get the execution time of all the four sections of the loops.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 500
#define L 500
#define M 500

int main (int argc, char *argv[]) {
int i,j,k;
double  a[N][L], b[L][M],c[N][M];
clock_t  begin, end;
double   time_spent;
begin = clock();

for (i=0; i<N; i++)
    for (j=0; j<L; j++)                    Loop1
        a[i][j]=i+j;                       (Initializing Matrix
for (i=0; i<L; i++)
    for (j=0; j<M; j++)                    Loop2
        b[i][j]= i*j;                      (Initializing Matrix B)
for (i=0; i<N; i++)
    for (j=0; j<M; j++)                    Loop3
        c[i][j] = 0;                       (Initializing Matrix C)
for (i=0; i<N; i++)
    for(j=0; j<M; j++)
        for(k=0; k<L; k++)                 Loop4
            c[i][j] += a[i][k] * b[k][j];  (Dot product
end = clock();                             A*B=C)
time_spent = (double)(end-begin)/CLOCKS_PER_SEC; }
```

Fig. 3 Sequential code

Fig. 4 show the parallel implementation does not require changing the sequential code but adding several lines of code comprised of compiler directives and run-time library routines. Using the runtime library routines required inclusion of header file **#include <omp.h>.** Since we need to measure the execution time of the parallel sections, runtime library routine **omp_get_wtime()** used to get the wall clock time and measures the execution time of the parallel code. Runtime library routine **omp_get_max_threads()** returns the maximum value of threads running and is used in the parallel code to check the running threads. Since all the four loops can be implemented in parallel, a compiler directive construct **#pragma omp for** used to share iteration of the loop across a team of threads. A compiler directive construct #**pragma omp parallel** enclosing the four loops for the directive to execute in parallel. The directive creates a parallel region for the dynamic extent of the structured block that follows the directive. Loop5 is the parallel region construct enclosing the four loops.

Although there are many directives and library routines, the parallel implementation only use the most common and important directives and library routines. However, we can explore the directives and library routines by incrementally adding the constructs.

The sequential and parallel matrix multiplication implemented in C on Intel Pentium Dual-Core (2 cores/2 threads)/Ubuntu and Intel Core i5(2 cores/4 threads)/Mac OS X.

```
#include <omp.h>>  / Include omp.h header file to use OpenMP run-time
library routines /
#include <stdio.h>
#include <stdlib.h>

#define N 500
#define L 500
#define M 500
int main (int argc, char *argv[])
{
   int i,j,k,chunksize,mxthrds;
   double  a[N][L],b[L][M],c[N][M];
   chunksize = 10;
   double  time1, time2;
mxthrds = omp_get_max_threads(); / Return the maximum number of
threads /Run-Time Library/
time1 = omp_get_wtime(); / Provides wall clock time//Run-Time Library/
#pragma omp parallel shared(a,b,c,numthreads,chunksize)
private(threadid,i,j,k)
 / Fork team of threads giving them own copies of variables /Compiler
Directive/
{
#pragma omp for schedule (static, chunksize)
 / Work-sharing construct. Threads share row
iterations based on chunk size //Compiler
directive/                                          Loop1
for (i=0; i<N; i++)
    for (j=0; j<L; j++)
        a[i][j]=i+j;

#pragma omp for schedule (static, chunksize)
/ Work-sharing construct. Threads share row
iteration based on chunk size //Compiler            Loop2
directive/
for (i=0; i<L; i++)
    for (j=0;j<M;j++)
        b[i][j]=i*j;

#pragma omp for schedule (static, chunksize)                    Loop5
/ Work-sharing construct. Threads share row
iterations based on chunk size //Compiler directive/
for (i=0; i<N; i++)                                 Loop3
    for (j=0; j<M; j++)
        c[i][j] = 0;

#pragma omp for schedule (static, chunksize)
/ Work-sharing construct. Threads share row
iteration based on chunk size //Compiler directive/
for (i=0; i<N; i++) {                               Loop4
    for(j=0; j<M; j++)
        for(k=0; k<L; k++)
            c[i][j] += a[i][k] * b[k][j];}
} /End of parallel region. All threads join master
thread and disband/
time2 = omp_get_wtime() – time1; / Provides wall clock time//Runtime
Library/ }
```

Fig. 4 Parallel code

5 Results

Figs. 5 and 6 shows sequential and parallel execution time of matrix multiplication on four threads (Intel Core i5 with Mac OS X) and two threads (Intel Pentium Dual Core with Ubuntu) respectively.

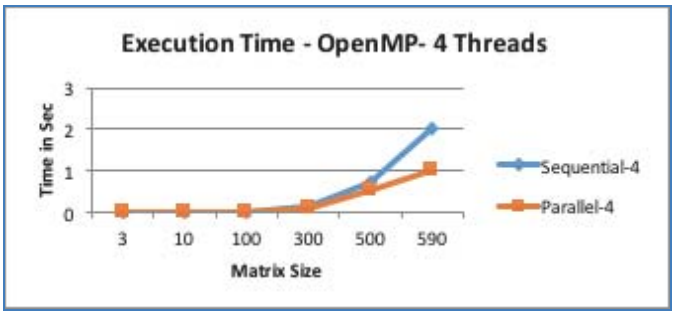

Fig. 5 Intel Core i5 (Mac OS X)

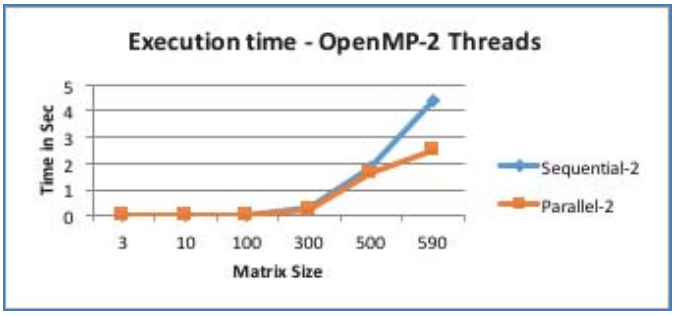

Fig. 6 Intel Dual Core (Ubuntu)

From the graphs above, it clearly shown parallel implementation of matrix multiplication on multicore multithread general purpose computer reduce the execution time. The reduction of the computing time is significance as the size of the matrices become larger. The program implemented on two different architectures running Unix and Linux based platforms. From the implementation on both architectures, we can see transition from one architecture to another under the same platform required no modification to the code. This is because OpenMP provide highest degree of portability making parallel coding easier. Programming multithread code often requires complex co-ordination of threads. However, with OpenMP, conversion from sequential to parallel do not required major changes to the coding. In fact, the conversion can be done incrementally.

The graph also shows for smaller sizes of matrices, the parallel performance decrease compared to similar serial implementation. This is due to the overhead associated with setting up the parallel environment, thread creation and thread termination. These overheads comprise a significant portion of the total execution time for smaller size of matrices.

Intel Core i5 processor with hyperthreading enable two threads to run on each of the processor's cores giving a total of four virtual cores or four threads. With OpenMP, we can easily experiment any number of threads to improve performance. This can be implemented using OpenMP environment variable **export OMP_NUM_THREADS = (Number of threads)** to explicitly set the number of threads used during execution. Fig. 7 show the execution time of parallel matrix multiplication with two, four and eight threads**.** From the graph, the performance improved as the number of threads increased.

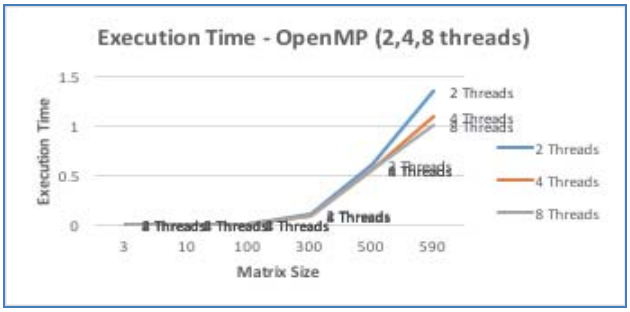

Fig. 7 Execution time (Intel Core i5)

6 Conclusions

The focus of this study is to examine shared memory parallel programming using OpenMP run on portable general purpose computers. From the implementation, it clearly shown multithreading programming improved performance. OpenMP excels in taking a serial program and maintain its structure when introducing parallelism. Furthermore, using OpenMP to incorporate parallelism in applications is easier compare to any form of shared memory programming model since almost all compilers today had some level of support for OpenMP. Hence, scientist and application developer should exploit parallelism in smaller scale system. They can write a code from scratch and incrementally add parallelism for performance and advance to parallel systems if required. Many studies mentioned the improvement of performance using multicore multithread processors depends mainly on implementation of software algorithms where performance gained based on the fraction of the software that can be run in parallel.

7 Acknowledgement

This research is supported by Zayed University through Research Incentive Fund(RIF) Grant R14058.

8 References

[1] S. F. McGinn and R. E. Shaw. "Parallel Gaussian Elimination using OpenMP and MPI"; Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Application, pp. 169, June 2002.

[2] Muhammad Ali Ismail, S. H Mirza, and Talat Altaf. "Concurrent matrix multiplication on multi-core processors"; International Journal of Computer Science & Security, Vol 5(4), pp. 208-220, 2011.

[3] Bob Kuhn, Paul Petersen, & Eamonn O'Toole. "OpenMP versus threading in c/c++"; Concurrency-Practice and Experience, Vol 12, pp. 1165–1176, 2000.

[4] Geraint Lewis, and Chris Power. "The challenge of the modern scientist is to avoid career suicide"; The Conversation, 2014. Retrieved from https://theconversation.com/the-challenge-of-the-modern-scientist-is-to-avoid-career-suicide-22735

[5] Uzi Vishkin. "Is Multicore Hardware for General-Purpose Parallel Processing Broken"; Communications of the ACM, vol. 57(4), pp. 35-39, 2014.

[6] Oracle (2010) "Developing Parallel Programs – A Discussion of Popular Models: An Oracle White Paper"; Retrieved from http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-parallel-programs-170709.pdf

[7] "Intel's Threading Building Blocks Tutorial"; Retrieved from http://www.threadingbuildingblocks.org/documentations.php

[8] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. "The Design of a Task Parallel Library"; Proceedings of the 24th ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2009.

[9] Liang Chen, Deepankar Bairagi & Yuan Lin. "MCFX: A new parallel programming framework for multicore system"; Proceedings of International Supercomputing Conference. 2009. Retrieved from http://www.springerlink.com/content/9004q3172421j810/

[10] "The OpenMP API specification for Parallel Programming"; Retrieved from http://www.openmp.org

[11] Barbara Chapman, Gabriele Jost, & Ruud Van Der Pas. "Using OpenMP: Portable Shared Memory Parallel Programming". MIT Press. 2007

[12] Anshul Gupta & Vipin Kumar. "Scalability of parallel algorithms for matrix multiplication"; Parallel Processing, ICPP 1993. International Conference on. vol. 3. IEEE Press, 2003.

[13] Jaeyoung Choi. "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers"; Concurrency, Practice and Experience, vol. 10(8), pp. 655-670, July 1998.

[14] Tirtharaj Dash & Tanistha Nayak. "Chain Multiplication of Dense Matrices: Proposing a Shared Memory based Parallel Algorithm"; International Journal of Computer Applications (0975-8887), vol 58(1), pp:11-16, 2012

[15] Rozita Johari, Mohd Yazid Mohd Saman, Mohamed Othman, & Bachok Taib. "Implementation of Parallel Boundary Integral Method on Spherical Bubble Dynamics Using Shared Memory Computer"; Malaysian Journal of Computer Science, 13(1), 1-11, 2000.