Title	Parallel Processing in Numerical Integration
All Authors	Myint Myint Thein
Publication Type	Local Publication
Publisher (Journal name, issue no., page no etc.)	Universities Research Journal 2010, Vol. 3, No.3
Abstract	Numerical Integration is a method of computing an approximation of the area under the curve of a function. The area under the curves is approximated the sum of the area of the subinterval rectangles. The difference between the sum of the rectangle's area and the area under the curve is reduced increasing the number of subintervals. The trivial parallel implementation is that if p threads are available, the sum is sliced into p pieces. One interval is attached to each of intervals to compute a partial sum. The local sums are collected into one processor, which will know the answer. The computations are parallelized with OpenMP directives supported by Microsoft Visual Studio 2008. Serial time Ts is the time to run the program on one thread and parallel time T p is the time to run the program on p threads are obtained. By comparing Ts and Tp of different threads, the parallel processing is faster than serial processing.
Keywords	parallel processing, decomposition, OpenMP
Citation	
Issue Date	2010

Parallel Processing in Numerical Integration

Myint Myint Thein

Abstract

Numerical Integration is a method of computing an approximation of the area under the curve of a function. The area under the curves is approximated the sum of the area of the subinterval rectangles. The difference between the sum of the rectangle's area and the area under the curve is reduced increasing the number of subintervals. The trivial parallel implementation is that if p threads are available, the sum is sliced into p pieces. One interval is attached to each of intervals to compute a partial sum. The local sums are collected into one processor, which will know the answer. The computations are parallelized with OpenMP directives supported by Microsoft Visual Studio 2008. Serial time T_s is the time to run the program on one thread and parallel time T_p is the time to run the program on p threads are obtained. By comparing T_s and T_p of different threads, the parallel processing is faster than serial processing.

Keyword: parallel processing, decomposition, OpenMP

Introduction

Processing of multiple tasks simultaneously on multiple processors is called parallel processing. The parallel program consists of multiple active processes simultaneously solving a given problem. A given task is divided into multiple subtasks using divide-and-conquer technique and each one of them are processed on different CPU programming on multiprocessor system .Given the computation-intensive nature of many application areas (such as encryption, physical modeling, and multimedia), parallel processing will continue to thrive for years to come.

Stages of designing parallel algorithms are partitioning (decomposition), communication, agglomeration and mapping. Parallel programming can be implemented in distributed memory systems and shared memory systems.

The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel is called decomposition. The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition. Decomposition into a small number of large tasks is called coarse-grained and decomposition into a large number of small tasks is called fine-grained.

Lecturer, Department of Computer Studies, Dagon University

OpenMP **(Open Multiple- Processing)** is an application programming interface(API) that supports multiplatform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface(MPI). OpenMP is shared memory and MPI is distributed memory.

OpenMP programming Model

OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads. OpenMP is explicit (not automatic) programming model, offering the programmer full control over parallelization. OpenMP uses the fork-join model of parallel execution. All OpenMP programs begin as a single process: the master thread.

The master thread executes sequentially until the first **parallel region** construct is encountered.

- **FORK**: the master thread then creates a **team of parallel threads**. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

OpenMP uses the fork-join model of parallel execution (see Figure 1).

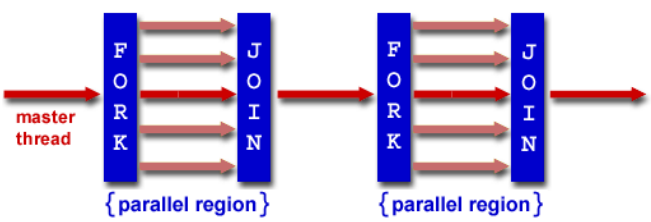


Figure 1: Fork-join parallelism

To develop new applications, a programmer must analyze the original problem, dissect it into tasks using both shared and local data, determine the data dependencies, and then reorder the tasks into execution units, which are coded using a parallel programming environment (see Figure 2).

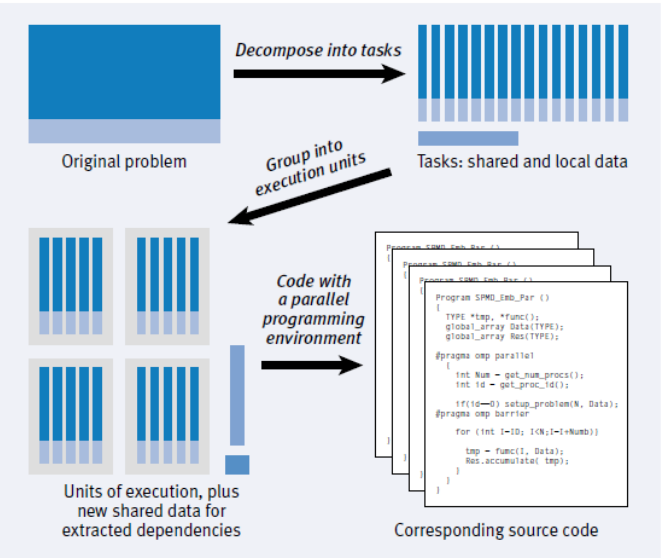


Figure 2: Methodology for writing parallel program

Numerical Integration

Numerical Integration is a method of computing an approximation of the area under the curve of a function, especially when the exact integral cannot be solved. For example, the value of the constant Pi can be defined by the following integral .However, rather than solve this integral exactly, the solution can approximate to the use of numerical integration:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

The pi is almost perfectly parallel. The only communication occurs at the beginning of the problem when the number of divisions needs to be broadcast and at the end where the partial sums need to be added together. The calculation of the area of each slice proceeds independently. This

would be true even if the area calculation were replaced by something more complex. Below are two serial applications, a numerical integration algorithm and a sorting algorithm. The program pi computes in parallel using numerical integration.
If the midpoint rule is used, the above integral can be computed as follows (see Figure 3).

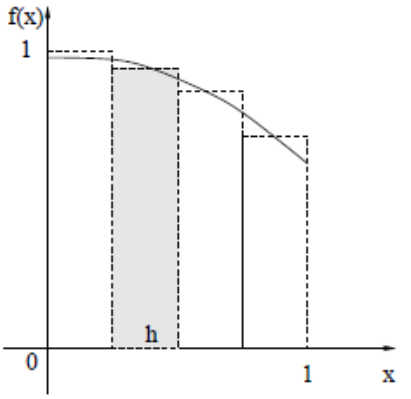


Figure 3: Numerical integration with midpoint rule

$$\int_0^1 \frac{4}{1+x^2}\,dx = h\sum_{i=1}^{n}\frac{4}{1+x_i^2} = h\sum_{i=1}^{n}\frac{4}{1+((i-1/2)h)^2}$$

The parallel program described in the following paragraphs computes an approximation of pi using numerical integration to calculate the area under the curve $4/(1+x^2)$ between 0 and 1 (see Figure 3). The interval [0,1] is divided into *num_subintervals* subintervals of width *1/num_subintervals.* For each of these subintervals, the algorithm computes the area of a rectangle with height such that the curve $4/(1+x^2)$ intersects the top of the rectangle at its midpoint. The area under the curves approximates to the sum of the area of the subinterval rectangles. The difference between the sum of the rectangle's area and the area under the curve is reduced by increasing the number of subintervals. The trivial parallel implementation is that if p threads are available, the sum is sliced into p pieces. One interval is attached to each of intervals to compute a partial sum. The local sums are

collected into one processor, which will know the answer. The computations are parallelized with OpenMP directives supported by Microsoft Visual Studio 2008.
(Speedup is defined as $S_p = T_s / T_p$, where T_s is the time to run the program on one thread and T_p the time to run the program on p threads.)

```
1. #include <omp.h>
2. float num_subintervals = 10000; float subinterval;
3. #define NUM_THREADS 5
4. void main ()
5. {int i; float x, pi, area = 0.0;
6.  subinterval = 1.0 / num_subintervls;
7. omp_set_num_threads (NUM_THREADS)
8. #pragma omp parallel for reduction(+:area) private(x)
9. for (i=1; i<= num_subintervals; i++) {
10. x = (i-0.5)*subinterval;
11. area = area + 4.0 / (1.0+x*x);
12. }
13. pi = subinterval * area;
14. }
```

Figure 4: OpenMP program to compute pi.

The compiler directive inserted into the serial program in line 8 contains all the information needed for the compiler to parallelize the program; *#pragma omp* is the directive's sentinel. The *parallel* keyword defines a parallel region (lines 9–12) that is to be executed by NUM_THREADS threads in parallel. NUM_THREADS is defined in line 3, and the *omp_set_num_threads* function sets the number of threads to use for subsequent parallel regions in line7. There is an implied barrier at the end of a parallel region; only the master thread of the team continues execution at the end of a parallel region. The pi program demonstrates only the basic constructs and principles of OpenMP, though OpenMP is a large and powerful technology for parallelizing applications.

Implementation

The following is serial program to compute PI calculation and figure 5 shows the flow chart for serial processing.

```
void Serial_Pi()
{   double x, sum = 0.0;

  int i;
  step = 1.0/(double) num_steps;
  for (i=0; i< num_steps; i++)
        {     x = (i+0.5)*step;
              sum = sum + 4.0/(1.0
        + x*x);
        }
  pi = step * sum;
}
```

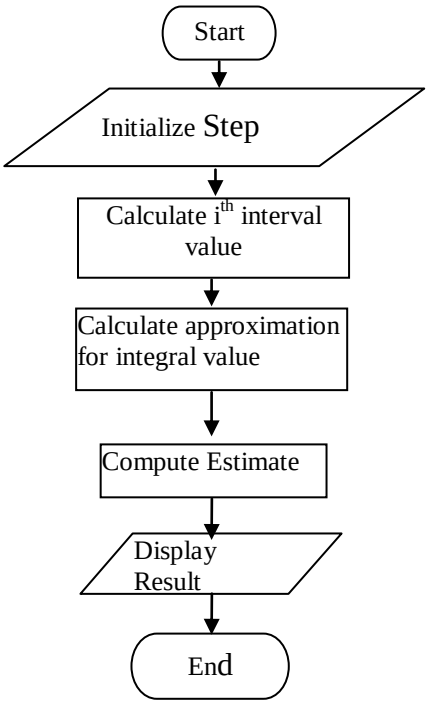


Figure 5: Serial processing flowchart

The following is parallel program for OpenMP to compute PI calculation and figure 6 shows the flow chart for parallel processing.

```
void OpenMP_Pi( )
{
double x, sum=0.0;
int i;
 step = 1.0 / (double)num_steps;
omp_set_num_threads(8);

#pragma omp parallel for private (x)
reduction(+:sum)
  for (i=0; i<num_steps; i++)
              { x = (i + 0.5)*step;
                sum = sum + 4.0/(1. + x*x);
               }
 pi = sum*step;
  }
```

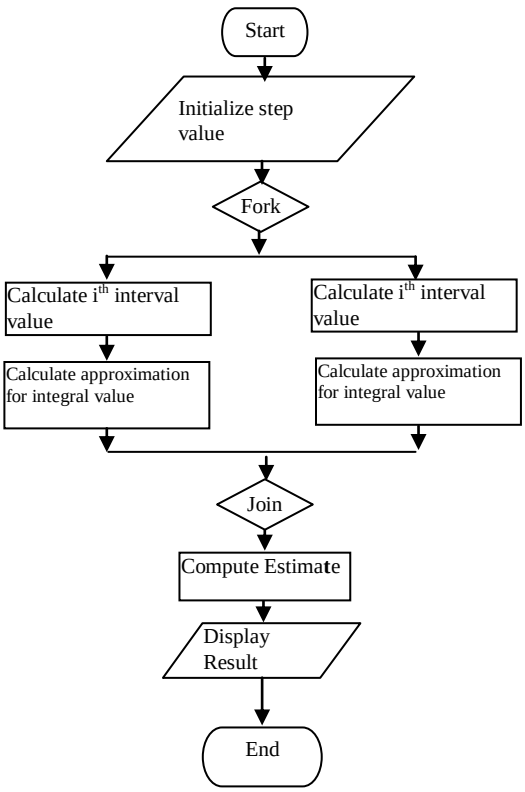


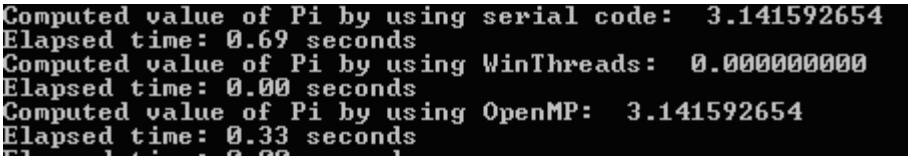
Figure 6: OpenMP coding and parallel processing flowchart

OpenMP starts with the header file (#include "omp.h"). The schedule clause effects how loop iterations are mapped onto threads as schedule (static [chunk]). It deals out blocks of iterations of size "chunk" to each thread. The flow chart shows how to calculate approximation for integral value with OpenMP as shown in Figure (6). By comparing the coding of serial and parallel program, assign num_step(n), omp_set_num_threads(n) and # pragma omp parallel are add in the serial program. Left of all coding is the same. To adjust the number of threads, the desire number of processors are changed at n value in num_step(n) and omp_set_num_threads(n).

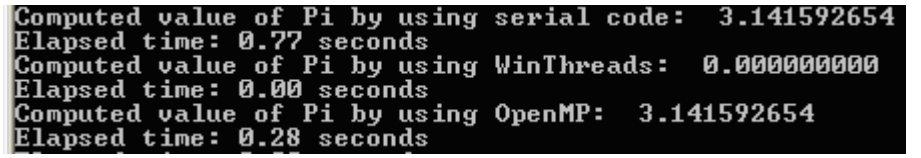
Results and Discussion

The sample numerical integration program is implemented with Microsoft visual studio 2008 and using OpenMP. The results of pi-program (C coding) can be observed as follow:

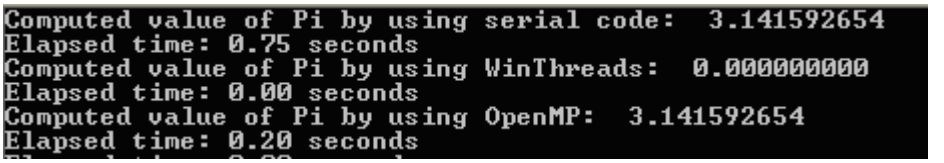
Threads 2



Threads 4



Threads 6



Threads 8

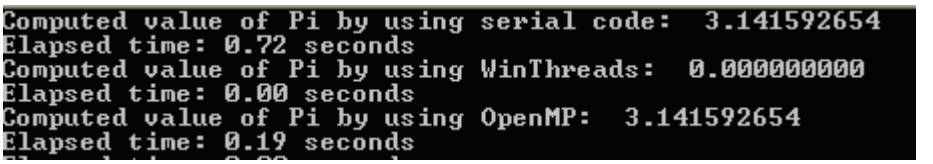


Figure 7: Output result of pi calculation for serial time and OpenMP time using 2, 4, 6 and 8 threads

Number of Threads	Serial time (s)	OpenMP time(s)
2	0.69	0.33
4	0.77	0.28
6	0.75	0.2
8	0.72	0.19

Figure 8: The relation of number of threads, serial time and parallel time

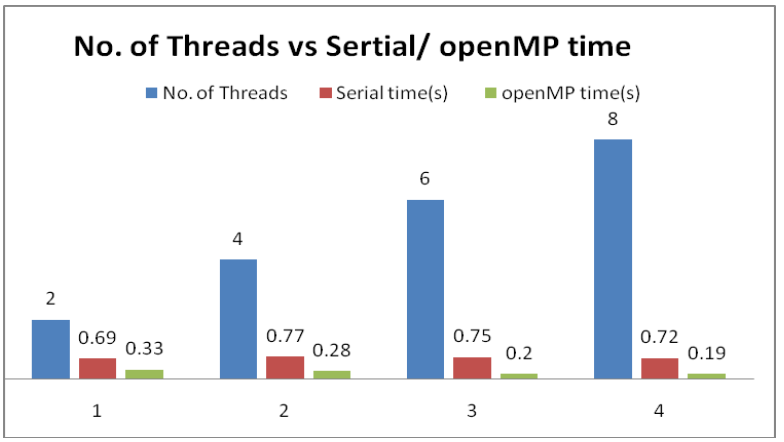


Figure 9: The relation of No.of threads, serial executing time and parallel executing time

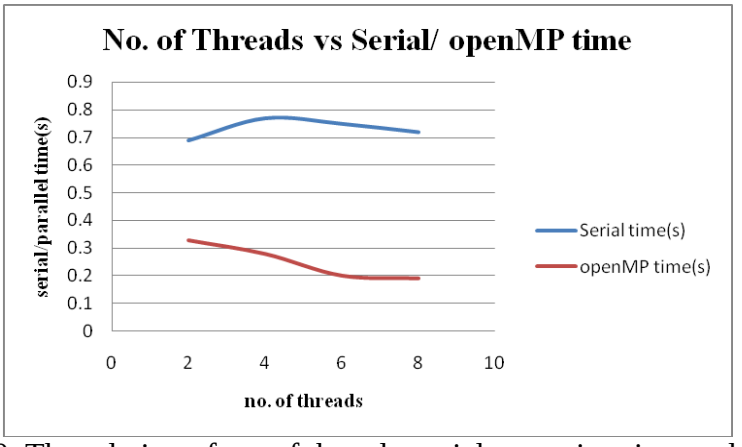


Figure 10: The relation of no. of threads, serial executing time and parallel executing time.

No. of Threads	Speedup
2	2.09
4	2.75
6	3.75
8	3.79

Figure 11: The output data of no. of threads and speedup.

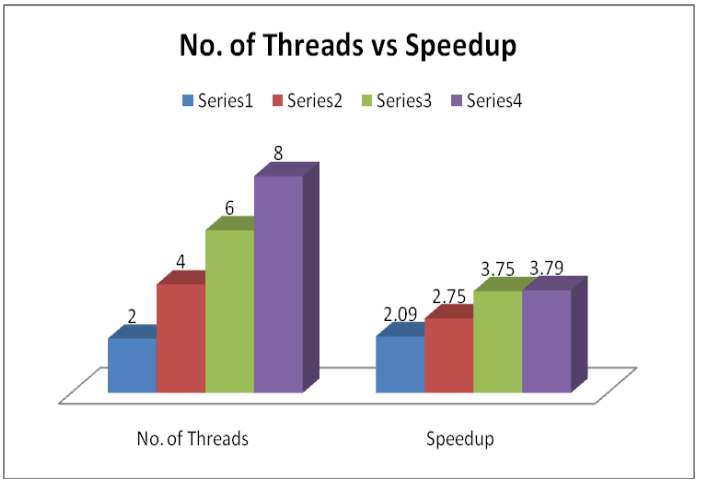


Figure 12: Relationship of No. of threads and speedup

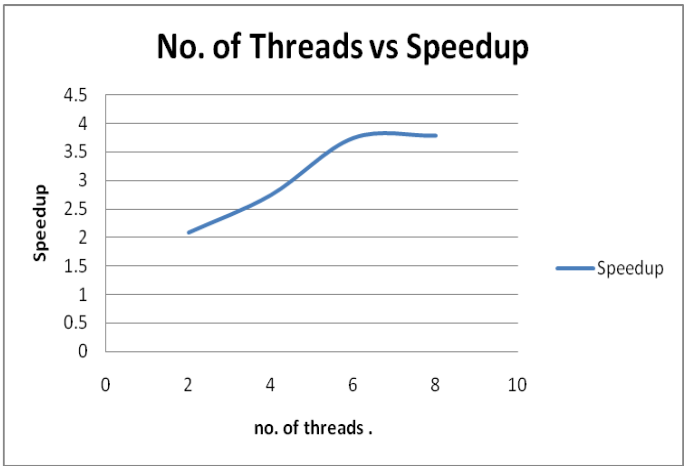
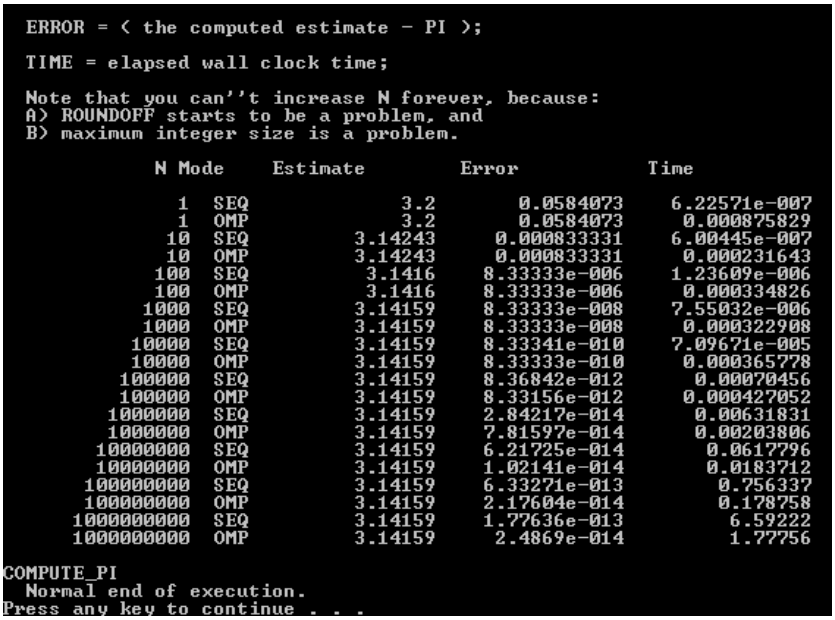


Figure 13: Relationship of no. of threads and speedup

When the sample program are executed with Corei7 multiprocessor, the elapsed time for serial processing T_s and the elapsed time for parallel processing T_p are obtained for different processors such as two, four, six and eight .When the number of threads increases , serial elapsed time T_s has a little changes and it is nearly constant but parallel elapsed time T_p is rapidly decreased as shown in figure(8) ,(9) and (10).

The finding results at Figure (11), (12) and (13) that the program run with the more CPU, the speedup is rapidly high . So the parallel processing is faster than serial processing when the program is executed with higher threads. From Figure (14), parallel computing is more suitable for enormous data or large program. It is no distinct in small program or data.



```
  ERROR = ( the computed estimate - PI );

  TIME = elapsed wall clock time;

  Note that you can''t increase N forever, because:
  A) ROUNDOFF starts to be a problem, and
  B) maximum integer size is a problem.

            N Mode     Estimate        Error           Time

            1  SEQ            3.2       0.0584073    6.22571e-007
            1  OMP            3.2       0.0584073     0.000875829
           10  SEQ        3.14243     0.000833331    6.00445e-007
           10  OMP        3.14243     0.000833331     0.000231643
          100  SEQ         3.1416    8.33333e-006    1.23609e-006
          100  OMP         3.1416    8.33333e-006     0.000334826
         1000  SEQ        3.14159    8.33333e-008    7.55032e-006
         1000  OMP        3.14159    8.33333e-008     0.000322908
        10000  SEQ        3.14159    8.33341e-010    7.09671e-005
        10000  OMP        3.14159    8.33333e-010     0.000365778
       100000  SEQ        3.14159    8.36842e-012      0.00070456
       100000  OMP        3.14159    8.33156e-012     0.000427052
      1000000  SEQ        3.14159    2.84217e-014      0.00631831
      1000000  OMP        3.14159    7.81597e-014      0.00203806
     10000000  SEQ        3.14159    6.21725e-014       0.0617796
     10000000  OMP        3.14159    1.02141e-014       0.0183712
    100000000  SEQ        3.14159    6.33271e-013        0.756337
    100000000  OMP        3.14159    2.17604e-014        0.178758
   1000000000  SEQ        3.14159    1.77636e-013         6.59222
   1000000000  OMP        3.14159     2.4869e-014         1.77756

COMPUTE_PI
  Normal end of execution.
Press any key to continue . . .
```

Figure 14:The values of Ts and Tp in pi calculation results for eight CPUs

Conclusion

Parallel computing can even be made available to students in high school and college, small software houses, and small-business start-ups. Both shared-memory multiprocessors and distributed-memory processors have advantages and disadvantages in terms of ease of programming. Porting a serial program to a shared-memory system can often be a simple matter by adding loop-level parallelism with OpenMP, but one must be aware of race conditions, deadlocks, and other problems associated with the paradigm that may arise. For programmers who are used to a thread paradigm, moving to OpenMP is relatively straightforward. Adding additional processors to a shared-memory multiprocessor increases the bus

traffic on the system, slowing down memory access time and delaying program execution.

By computing with this sample program, the performance of parallel processing is better than serial processing. Then OpenMP programs execute serially until they encounter the parallel directive. This directive is responsible for creating a group of threads. In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs. Parallel programming is no doubt much more tedious and error-prone than serial programming.

A variety of applications induce task-graph parallelism with coarse-grain granularity. Task-graph parallelism occurs when independent program parts are executed on different cores based on precedence relationships among the threads. These applications must be exploited to achieve the best possible performance on multicore processors.

Acknowledgement

I would like to express my sincere gratitude to Professor Daw Nwe Nwe Win, Head of Department of Computer Studies, University of Yangon for her kind permission to carry out this research. I would like to thank Professor Dr Pho Kaung, Pro-rector and Director, Universities' Research Centre, University of Yangon for his valuable suggestions.

References

Ami Marowka, communication of the ACM. September 2007/Vol.50.No.9: Parallel Computing on any desktop

Grama, A. and Gupta, A. and Karypis, G. and Kumar, V. (2003), Introduction to Parallel Computing, 2nd Second Edition, Pearson Education, The Benjamin/Cummings, ISBN: 7-111-12512—6/TP-2782

Science & Technology Support ,High Performance Computing, Ohio Super computer Center 1224 Kinner Road,Columbus, OH 43212-1163: Parallel Programming with MPI.

Website:

https://**computing**.llnl.gov/tutorials/parallel_comp/ Introduction to Parallel Computing, Edward Chrzanowski, May 2004

http://openmp.org/

http://www.ece.ucsb.edu/ Introduction to Parallel Processing Algorithms and Architectures, Parhami@ece.ucsb.edu