Comparison and Analysis of Parallel Computing Performance Using OpenMP and MPI

Shen Hua[1,2,*] and Zhang Yang[1]

[1]*Department of Electronic Engineering, Dalian Neusoft University of Information Dalian, 116023/Liaoning, China*

[2]*Department of Biomedical Engineering, Faculty of Electronic Information and Electrical Engineering, Dalian University of Technology, Dalian, 116024/Liaoning, China*

Abstract: The developments of multi-core technology have induced big challenges to software structures. To take full advantages of the performance enhancements offered by new multi-core hardware, software programming models have made a great shift from sequential programming to parallel programming.

OpenMP (Open Multi-Processing) and MPI (Message Passing Interface), as the most common parallel programming models, can provide different performance characteristics of parallelism in different cases. This paper intends to compare and analyze the parallel computing ability between OpenMP and MPI, and then some proposals are provided in parallel programming. The processing tools used include Intel VTune Performance Analyzer and Intel Thread Checker. The findings indicate that OpenMP is in favor of implementation and provides good performance in shared memory systems , and that MPI is propitious to programming models which require nodes performing a large number of tasks and little communications in processes.

Keywords: Parallel computing, Multi-core, multithread programming, OpenMP, MPI.

1. INTRODUCTION

As the number of transistors has increased to fit within a die according to Moore's Law, manufacturers try packing more cores onto one single chip. However, adding cores is not synonymous with increasing computational power [1].

A new high performance computation technique involving multiple processors on a single silicon die is quickly gaining popularity. Most applications do not commonly utilize the new features of a hardware platform. In this case, few applications currently make efficient use of multiple processors to enhance single application performance. Processors are assigned to separate applications, resulting in suboptimal single application performance and frequent processor idle cycles from having too few applications available to execute. In this day and age of multi-core architectures, programming language support is in urgent need for constructing programs that can take great advantage of machines with multiple cores [2].

OpenMP (Open Multi-Processing) and MPI (Message Passing Interface) are the most popular parallel programming technologies [1]. OpenMP is an Application Program Interface (API) that can be used to explicitly direct multi-threaded. Shared memory parallelism comprised of three primary API components: compiler directives, runtime library routines and environment variables. OpenMP employs the fork-join model, where different tasks are implemented by multiple threads within the same address space. Programs are written in high-level application-specific operations. These operations are partially ordered according to their semantic constraints [3].

MPI is a message-passing library specification proposed as a standard by a committee of vendors, implementers and users [13]. It is designed to support the development of parallel software libraries. The primary functions of MPI are sending and receiving messages among different processes.

In this study, we compare the implementations by showing the results achieved between OpenMP and MPI when executing the same concurrent collections application. Our experiments show that significant improvements in program performance are obtained using multi-threading and parallel computing techniques. The CPU time used in a computation has been greatly reduced, in turn, reduces the usage of CPU resource.

The paper is organized as follows. Section 1 provides the research background and methodology. Section 2 reviews the parallel programming models, including OpenMP and MPI. Section 3 introduces a typical parallel programming environment: Intel Composer XE 2013, utilizing Loop Profiler and VTune Amplifier to analyze bottleneck and overload of programs. Section 4 provides a practical example to reflect the computation power of the multi-core processor by adding parallelism to a C++ application. Section 5 discusses the analytic results and Section 6 concludes the paper.

*Address correspondence to this author at the Department of Electronic Engineering, Dalian Neusoft University of Information Dalian, 116023/Liaoning, China; Tel: +86-411-84832209; Fax: +86-411-84832210; E-mail: shenhua@neusoft.edu.cn

2. PARALLEL PROGRAMMING MODELS

2.1. OpenMP

OpenMP is an Application Programming Interface (API), widely accepted as a standard for high-level shared-memory parallel programming. It is a portable, scalable programming model that provides a simple and flexible interface for developing shared-memory parallel applications in FORTRAN, C and C++. Now its latest version is OpenMP 3.1. Standard is jointly defined by a group with members from major computer hardware and software vendors like IBM, Silicon Graphics, Hewlett Packard, Intel, Sun Microsystems and The Portland Group, etc [3, 4].

OpenMP was structured around parallel loops and was meant to handle dense numerical applications. The simplicity of its original interface, the use of a shared memory model, and the fact that parallelisms of a program are expressed in directives. Those are loosely-coupled to the code. Recently, OpenMP is attracting widespread interest because of its easy-to-use portable parallel programming model.

A. OpenMP API

OpenMP API consists of the following components [3]:

- Compiler directives: Instructs the compiler to process the code section following the directive for parallel execution.
- Library routines: Routines that affect and monitor threads, processors and environment variables. It also has routines to control thread synchronization and get timings.
- Environment variables: Variables controlling the execution of the OpenMP program.

B. Working Mechanism of OpenMP

Parallel execution in OpenMP is based on the fork-join model, where the master thread creates a team of threads for parallel execution [3].

- Program execution begins as a single thread of execution, called the initial thread.
- A thread encountering a parallel construct becomes a master, creates a team of itself and additional threads.
- All members of the team execute the code inside parallel construct.

Each thread has a temporary view of the memory, which is like a cache and a private memory not accessible other thread.

- Relaxed consistency, the thread's view of memory is not required to be consistent with the memory at all times.
- Flush operation causes the last modified variable in the temporary view to be written to memory.
- Private variables of a thread can be copies of data from memory and cannot be accessed by other threads.

The parallel construct can be nested arbitrary number of times. Thread encountering parallel becomes the master.

OpenMP has been very successful in exploiting structured parallelism in applications. With increasing application complexity, there is a growing need for addressing irregular parallelism in the presence of complicated control structures. This is evident in various efforts by the industry and research communities to provide a solution to this challenging problem. One of the primary goals of OpenMP is to define a standard dialect to express and efficiently exploit unstructured parallelism.

2.2. MPI

MPI is a specification for a standard library for message passing, defined by the MPI Forum in April 1992. During the next eighteen months the MPI Forum met regularly, and Version 1.0 of the MPI Standard was completed in May 1994 [12, 14]. Some clarifications and refinements were made in the spring of 1995, and Version 1.1 of the MPI Standard is now available [13].

Multiple implementations of MPI have been developed in many different versions：MPICH, LAM, and IBM MPL, etc [13]. In this paper we discuss the set of tools that accompany the free distribution of MPICH, which constitute the beginnings of a portable parallel programming environment.

A. Features of MPI

MPI is a library which specifies the names, calling sequences and the results of functions. MPI can be invoked by C and FORTRAN programs; the MPI C++ library consisting of classes and methods can also be called. MPI is a specification, not a particular implementation. A correct MPI program should be able to run on all MPI implementations without change [12, 16].

- Messages and Buffers

Sending and receiving messages are the two fundamental operations of MPI. Messages can be typed with a tag integer. Message buffers are more complex than a simple buffer. User can create their own data types by giving options to address combination.

- Communicators

The notions of context and group are combined in a single object called a communicator, which becomes an argument to most point-to-point and collective operations. Thus, the destination or source specified in send or receive operation always refers to the rank of the process in the group identified by a communicator.

- Process Groups

Processes belong to groups. Each process is ranked in its group with a linear numbering. Initially, all processes belong to one default group.

- Separating Families of Messages

MPI programs can use the notion of contexts to separate messages in different parts of the code. It is very useful for writing libraries. The context is allocated at run time by the system in response to user (or library) requests.

B. Basic Frame of MPI Functions

The initialization and end processes of MPI environment are given as follows:

(1) Before calling MPI routines, each process should be implemented by MPI_INIT;

(2) Call MPI_COMM_SIZE to get default group (group);

(3) Call MPI_COMM_RANK to get the size of the default group of logical Numbers (starting from 0);

(4) Send messages to other nodes or meet news from the other nodes through MPI_SEND and MPI_RECV according to the needs;

(5) Uses MPI_FINALIZE to eliminate MPI environment when there is no need to call any MPI routines. The process can either end at this time or continue to execute another irrelevant statement of MPI.

A minimum routine of MPI program with the six functions mentioned above is given below [14][15]:

int MPI Init(int *argc, char ***argv) /*Initialize MPI*/

int MPI Comm size(MPI Comm comm, int *size) /* Find out how many processes there are */

int MPI Comm rank(MPI Comm comm, int *rank) /* Find out which process I am */

MPI_Send(address, count, datatype, destination, tag, comm) /* Send a message */

MPI_Recv(address, maxcount, datatype, source, tag, comm, status) /* Receive a message */

int MPI Finalize() /*Terminate MP*/

3. INTEL PARALLEL PROGRAMMING ENVIRONMENT

Intel Composer XE 2013 is a parallel programming environment. It includes C++ compiler, VTune amplifier and threading building blocks, etc. These entire modules can delivers outstanding performance for applications that run on systems using Intel Core or Xeon processors, including Intel Xeon Phi coprocessors, and IA-compatible processors. It combines all the serial and parallel tools from Intel C++ Composer XE 2013 with those from Intel FORTRAN Composer XE 2013. Visual Studio 2008, 2010 or 2012 is a prerequisite on Windows and the gnu tool chain is supported on Linux [8].

3.1. Intel C++ Compiler

Intel C++ compilers are not available as stand-alone compilers. They are available in packages, some of which include other build-tools, such as libraries, and other which include performance and threading analysis tools.

Intel C++ is part of Intel Parallel Studio XE and Intel C++ Studio XE for Windows and Linux. It includes performance analysis and thread-diagnostic tools. Intel C++ Composer XE (available for Windows, Linux and Apple OS X) and Intel Composer XE, which also includes Intel Fortran (available for Windows and Linux); It does not include the analysis and thread-diagnostic tools. Intel compilers are also included in Intel Cluster Studio (no analysis tools) and Intel Cluster Studio XE (analysis tools included). The cluster tools are available for Windows and Linux. Packages that include Intel C++ also include the Intel Math Kernel Library (Intel MKL), Intel Integrated Performance Primitives (Intel IPP) and Intel Threading Building Blocks (Intel TBB). Fortran-only packages only include Intel MKL [8].

Intel C++ compiler has many advanced performance features, such as High Performance Parallel Optimizer (HPO), Automatic Vectorization, Guided Auto Parallelization (GAP), Inter-procedural Optimization (IPO), Loop Profiler, Profile-Guided Optimization (PGO) and OpenMP 3.1[9].

3.2. VTune Amplifier XE 2013

VTune Amplifier XE 2013 has lightweight hotspots analysis that uses the Performance Monitoring Unit (PMU) on Intel processors to collect data with very low overhead. Increased resolution can find hot spots in small functions that run quickly [9].

When developing, optimizing or tuning applications for performance on Intel architecture, it is important to understand the processor micro-architecture. Having said that, an insight into how the applications are performing on the micro-architecture is gained through performance monitoring. The Intel VTune Performance Analyzer provides an interface to monitor performance of the processor and gain insights into possible performance bottlenecks.

The Intel VTune Performance Analyzer is a powerful software-profiling tool available on both Microsoft Windows and Linux OS. VTune helps to understand the performance characteristics of software at all levels: system, application and micro-architecture. The main features of VTune are sampling, call graph and counter monitor [9]. In this study, we focus on the event based sampling feature of VTune and on how to choose these VTune events and ratios for monitoring performance.

While VTune is used to sample an application, it may not be necessary to use all the events and ratios that are available in the tool. The most commonly used events are clockticks and instructions retired which are selected by VTune when you create a new event.

3.3. Intel Threading Building Blocks

Profiling is a technique for measuring where software programs consume resources, including CPU time and memory. Code profiling tools are visual instruments that help to expose performance bottlenecks and hotspots that inhibit to achieve the desired code parallelism. The instrumentation is done by gathering information, such as execution times and number of calls, during execution of program's entities such as functions and loops. The outcome is visualized graphically on the screen and enables the programmer to detect, for example, the imbalance in either computation or communication that is present in an algorithm.

Intel Thread Profiler is a profiling tool that identifies bottlenecks that limit parallelism of threaded applications and locates overheads due to synchronization, stalled threads and long blocking times. Thread Profiler supports applications threaded with Windows threads, POSIX threads and OpenMP [4, 11].

Thread Profiler creates two kinds of views for analyzing the behavior of threaded application: Profile view and Timeline view. While the Timeline view is more intuitive to the way a sequential programmer think, the Profile view demands to think in parallel which is a different way of thinking.

Intel Threading Building Blocks (Intel TBB) is a widely used, award-winning C++ template library for creating high performance, scalable parallel applications. It includes scalable memory allocation, load-balancing, work-stealing task scheduling, a thread-safe pipeline and concurrent containers, high-level parallel algorithms, and numerous synchronization primitives.

4. REALIZATION METHODS FOR PARALLELIZATION

In this section we use an example to demonstrate how parallel programming can be realized on a computing application, and compare performance indicators of programming parallelization between OpenMP and MPI.

The parallel programming example we have chosen is an application of Newton iteration algorithm. It is a non-liner algorithm, has multi-scales, and is easy to modulate. We assume that Newton iteration algorithm is used in computing temperature compensation for platinum resistance. The central processor computes each of the temperature data in iterations, utilizing the computation power of the multi-core processors. Dedicated threads are used for the data conversion as well as Newton iteration.

4.1. Newton Iteration Algorithm

The resistances of platinum resistor, *R(t)*, with respect temperature t are:

$$R(t) = R_0[1 + at + bt^2 + c(t - 100)t^3] \quad (1)$$

Where -200 ≤ t ≤ 0;

$$R(t) = R_0(1 + at + bt^2) \quad (2)$$

Where 0 ≤ t ≤ 850, a= 3.90802E-3, *b*=-5.802E-7, and *c*=-4.2735E-120.

Using Newton iteration algorithm, consider

$$R^{'}(t) = \frac{R(t_n) - R(t_{n-1})}{t_n - t_{n-1}} \quad (3)$$

Rearranging we get,

$$t_n = t_{n-1} + \frac{R(t_n) - R(t_{n-1})}{R^{'}(t_{n-1})} \quad (4)$$

From equation *R (t)*, we obtain

$$t_n = t_{n-1} + \frac{R - 100(1 + at + bt^2 - 100ct^3 + ct^4)}{100(a + 2bt - 300ct^2 + 4ct^3)} \quad (5)$$

Simplifying the equation

$$t_n = t_{n-1} + \frac{R - 100(1 + at + bt^2)}{100(a + 2bt)} \quad (6)$$

Let the relationship between *R* and *t* be linear. We choose the first approximation

$$t_0 = \frac{\frac{R}{100} - 1}{a} \quad (7)$$

Assuming that the stopping criteria is not a fixed value, the program iterates until $t_n - t_{n-1}$ is less than the given stopping criteria of temperature.

4.2. Multi-Thread Model

The threaded applications can communicate via the inherently shared memory of the process, avoiding the traditional more expensive inter-process communication facilities(pipes, sockets, etc.). Furthermore, the concurrence and the synchronization of the various tasks, that generally need a large effort in a multi-process program, are easily obtained with specific system calls that co-ordinate in a natural manner the activity of the threads [6].

To evaluation the effectiveness of multi-threads, and reduce the overload among threads, we construct a multi-thread model. The Newton iteration algorithm is run on the central processor. To achieve high-speed data processing and demonstrate the computation power of the dual-core processor, several threads, a receive-data pool, and a send-data pool are used. Fig. (**1**) below depicts the relationship of the main elements of data processing above. In particular, the receive thread reads the data from receive-data pool, and then pushes it into the receive queue. Two process threads, performing identical functions and running in parallel, constantly monitor the status of the receive queue. If it is not empty, the two push the data into the send-data pool; Main thread reads data from the send-data pool.

4.3. Analysis Indicators of Parallel Programming Performance

The testing tools were used by Intel VTune Performance Analyzer, Intel Thread Checker and Intel Thread Profiler. For each case, three data parameters, Memory size, CPU time and speed up ratio are collected for comparison. Speed up ratio can be calculated in follow formula:

Speed up Ratio =

(Sequential Execution Time – Parallel computing Execution time)*100 / Sequential Execution Time.

5. DISCUSSION

In this section we introduce a special application based on Newton iteration algorithm. At first, we make a sequential program, then using multi-threads, and then parallelizing this program. The processor is Intel Core i5, and the development environment is Intel Composer XE 2013.

Our methodology is divided into three phases:

The first phase is to compare the performance between single thread and dual threads, and then get performance analysis using Intel VTune Performance Analyzer in order to identify performance optimization opportunities and detect bottlenecks.

The second phase is to modify the original sequential program to accommodate paralleling computing using OpenMP, and then conduct performance analysis using Intel

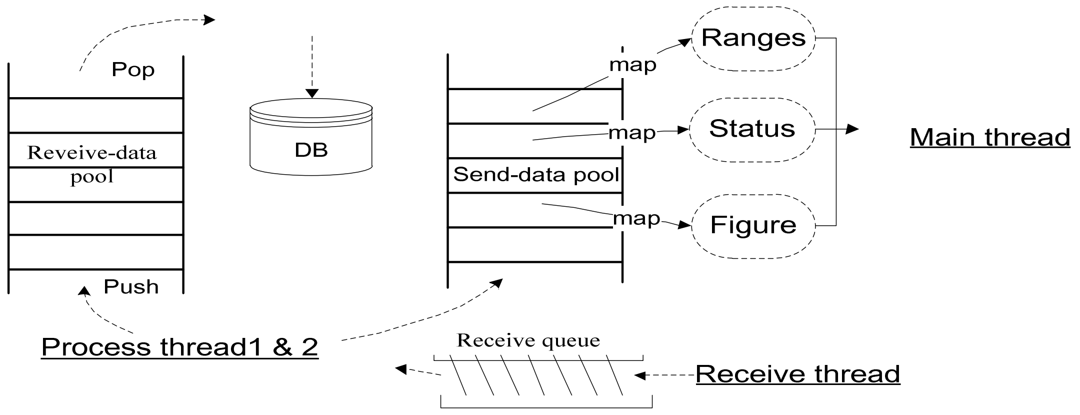

Fig. (1). the Model of Multithread Processing.

Table 1. Performance using Single- and Dual-Threads

Object	Counters	Instructions-Retired	Clockticks
Single thread (Un-optimized, Microsoft compiler)	Processor Queue Length（1.298）	80%～90%	25%～50%
	Context Switches/sec(17542)		
	Memory available bytes(619M)		
	Processor Time(59.367%)		
Dual-thread (Optimized, Intel compiler)	Processor Queue Length(2.941)	<5%	<15%
	Context Switches/sec(4995)		
	Memory available bytes(552M)		
	Processor Time(51.64%)		

VTune Performance Analyzer to identify performance optimization opportunities and detect bottlenecks [9].

In the third phase, we rewrite the sequential program of Newton iteration algorithm using MPI, then we allocate data races, memory leakage and debug the multithreaded applications, using Intel Thread Checker [10, 11].

Experiments are conducted to demonstrate the performance of the three programs implementing the Newton iteration algorithm. A comparative study is conducted on the performances of the program using single thread and dual-thread, using OpenMP or MPI, respectively.

5.1. Performance Analysis of Multithreads

The testing tools used are Intel VTune Performance Analysers, Intel Thread Checker, and Intel Thread Profiler. A most CPU-intensive computation thread was chosen to perform the tests. The experiments are conducted under single-thread and dual-threads cases. For each case, three data parameters, Counter Values, instructions Retired and Clockticks, are collected for comparison. Table **1** below summarizes the results.

In Table **1**, it can be seen that significant improvements on several key parameters in dual-threads, such as Processor Queue Length, Context Switches/sec and Memory available bytes and Processor Time, are obtained using dual-threads and the optimization technique. It shows that the number of instructions used in the computation is reduced to 5% or below from 80%~90%, using dual-threads and optimization. This, in turn, greatly reduces the usage of CPU resource, a drop from 25%～50% to below 15%.

5.2. OpenMP Parallel Realizations

In order to measure the performance speed up ratios of OpenMP over a sequential algorithm, we write a sequential program that computes temperature compensation for platinum resistance using the Newton iteration algorithm, then modify the program to support parallel computing using OpenMP. And finally, the run time of the Newton iteration algorithms are compared.

We use “#pragma omp parallel” statement to open the switch of OpenMP in this algorithm code. Only small changes are required in OpenMP to expose data locality, so a compiler can transform the code. Our notion of tiled loops allows developers to easily describe data locality. Furthermore, we employ two optimization techniques: one explicitly uses a form of local memory (the thread pool) to prevent conflict cache misses, whereas the second modifies the parallel programming pattern with dynamically sized blocks to increase the number of parallel tasks [5-7].

Table 2. Comparison Between Sequential and OpenMP

Cycle Times	Memory Size, Byte	Execution Time, s (Sequential Program)	Execution Time, s (OpenMP Program)	Speed up Ratio, %
1000	1023432	0.013	0.012	7.69
2000	2106430	0.021	0.016	23.8
5000	6225478	0.047	0.032	31.9
10000	12365232	0.082	0.047	42.7

Table 3. Execution Time Based on MPI

Cycle times	Memory Size, Byte	Thread0 Run Time, s (C++ Clock)	Thread1 Run Time, s (C++ clock)	Thread0 Run Time, s (MPI Clock)	Thread1 Run Time, s (MPI Clock)
1000	3426442	0.038	0.037	0.02	0.02
2000	5105630	0.082	0.088	0.07	0.07
5000	12542678	3.216	3.242	3.14	3.14
10000	27354432	7.023	7.071	7.02	7.02

In order to obtain accurate run-time measurements, each algorithm is executed in loops and the total run-time is depended on the cycle times of each loops. The run-time of the algorithm is the average value of each loop. In addition, each program takes one command argument, argv, as a number of loop-backs at run-time, so that the total execution time of the programs can be controlled.

Experiments were conducted to evaluate the performance of different programming implementations of the application-specific Newton iteration algorithm. A comparative study is conducted on the performances of the program between sequential program and parallelization program additional OpenMP. The testing tools used are Intel VTune Performance Analyzer, Intel Thread Checker and Intel Thread Profiler. For each case, three data parameters, Memory size, Execution time and Speed up ratio, are collected for comparison.

We set the cycle times from 1000 to 10000, and then compare the memory size and execution time between sequential program and OpenMP program based on different cycle times. The results of experiments are shown in Table **2**.

Observing the data in Table **2**, it can be seen that OpenMP does not obviously improve the performance of Newton iteration algorithm over the sequential program when the cycle times are small (e.g., 1000). The speed up ratio increases significantly when the cycle times raise from 1000 to 10000.

The experimental results in Table **2** show that OpenMP does increase the performance compared to sequential programs when cycle times are smaller. The parallel algorithm in OpenMP has significantly improved the performance, especially if the cycle times reach a large value.

5.3. MPI Parallel Realizations

Similarly, comparative studies were made using MPI programming environment. Processors P0 and P1 are used as the master and the slave processors, respectively. Unlike the OpenMP program, the MPI program splits the computing task into two subtask: P0 computes the first subtask, sends the second subtask to P1, and collect the result from P1 to come up with the final result [15].

In the first iteration (or even iteration) processors P0 and P1 both computing their data sets. Then, processor P0 performs a send operation of its first subtask to processor P1 using MPI_Send. Processor P1 receives data from P0, using MPI_Recv, and then performs the iteration to next loop.

There are two different clocks used to measure the performance of MPI programs. They are C++ clock and MPI clock [16]. Normally the run time measurements across languages are the time each language runs the parallel computation (not including overhead time), but in MPI programs, it is impossible to exclude the overhead using the C++ clock. Also, the C++ clock shows different values of time from different processes in a same program, while the MPI clock does not. Therefore, the MPI clock is used to measure the performance instead of the C++ clock. Table **3** contains the execution time of a program with different clocks from different processes.

Analyzing the data in Table **3**, we conclude that the performance of Newton iteration program using MPI is much slower than that of the sequential algorithm. It is because MPI-Send and MPI–Recv functions used in this program contain a synchronization mechanism, such that the program waits for the completion of *send* and *receive* data before proceeding to the next computation.

The delay caused by the synchronization between *send* and *receive* increases significantly when cycle times increase. From Table **3**, we conclude that the Newton iteration program does not seem to be beneficial in MPI, especially when the cycle times are large. So MPI is not recommended for this type of computation.

6. CONCLUSIONS

This paper provides solutions using OpenMP and MPI, the two different parallel programming methods, on Windows platform. The goal of this study is to provide programmers patterns of solving special problems in parallel programming.

This study compares our experiences in parallelizing and optimizing the special applications. We classify the performance among (a) sequential program (b) OpenMP parallelization and (c) MPI parallelization. All codes were carried out at the Intel Core i5 processor.

Through the comparison and analysis of parallel program performance, we show how to choose parallel models when designing a parallel system. The basic rules as follows:

- OpenMP is in favor of implementation and provides good performance in shared memory system. For those programming models with quick and small processing with no future extension, such as arithmetic computations, OpenMP should be considered for the best match for parallel programming.
- MPI is propitious to programming models for large systems with long term processing and future expansion. For example, the parallel sort-like algorithms require a large amount of computations, performed by nodes, but have little communication across processes. MPI are the best candidate for these types of systems.
- We also have used disparate tools (VTune, Thread Checker and Thread Profiler) in this work. Note that, Intel Composer tools can help to identify and understand application performance issues. Then, different algorithms and implementations can be used to mitigate the performance issue. Also, it is recommended to utilize thread and process parallel implementations to improve multi-core processor utilization.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

ACKNOWLEDGEMENTS

This work was financially supported by the Research Fund of the Liaoning Provincial Bureau of Education, P. R. China (Fund #: L2012490).

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software", *Dr. Dobb's Journal*, vol. 30, no.3, 2005.

[2] T. Mattson, J. DeSouza, "How fast is fast? Measuring and Understanding Parallel Performance", *Intel Webinar*, 2008.

[3] OpenMP Architecture Review Board, "OpenMP Application Program Interface", http://openmp.org/forum/

[4] B. Chapman, G. Jost, R. van der Pas, "Using OpenMP Portable Shared Memory Parallel Programming", USA, MIT Press, 2007.

[5] Shameem Akhter, "*Multi-Core Programming increasing performance through software multi-threading*", Electrical Industry Press, vol. 3, 2007.

[6] M. J. Garzar´an, M. Prvulovic, J. M. Llaberıa, V. Vinals,L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors". *ACM Transactions on Architecture Code Optimization*, vol. 2, no. 3, pp. 247-279, 2005.

[7] Kathleen Knobe, Carl D. Ofner, "Tstreams: A model of parallel computation ", Technical Report HPL-2004-78, HP Labs, 2004.

[8] Intel Composer XE 2013. http://software.intel.com/en-us/intel-composer-xe

[9] Intel VTune Performance Analyzers. http://www.intel.com/-software/products/vtune/

[10] Intel Thread Checker. http://software.intel.com/en-us/intel-thread-checker/

[11] Intel Thread Profiler, http://software.intel.com/en-us/intel-vtune/

[12] Message Passing Interface Forum, "Document for a standard message-passing interface", Tech. Rept. CS-93-214, University of Tennessee, 1994.

[13] The MPI Forum, "The MPI message-passing interface standard", http://www.mcs.anl.gov/mpi/standard.html, 1995.

[14] Message Passing Interface Forum, "MPI: A message-passing interface standard", 2009.

[15] Chao-Chin Wu, Lien-Fu Lai, Chao-Tung Yang, Po-Hsun Chiu, "Using hybrid MPI and OpenMP programming to optimize communications in parallel loop self-scheduling schemes for multicore PC clusters", *The Journal of Supercomputing*, vol. 60, no.1, pp. 31-61, 2012.

[16] R. Rabenseifner. "*Some Aspects of Message-Passing on Future Hybrid Systems*", EuroPVM/MPI, Springer, Munich, pp. 8-10, 2008.

Received: August 15, 2013 Revised: September 05, 2013 Accepted: September 28, 2013