Literature Survey for the Comparative Study of Various High Performance Computing Techniques

Zahid Ansari[1], Asif Afzal[2], Moomin Muhiuddeen [3], Sudarshan Nayak[4]

[1,3,4]Department of Computer Science, P. A. College of Engineering

[2]Research Scholar, Department of Mechanical Engineering, P. A. College of Engineering
Mangaluru, Karnataka, India-574153

Abstract *— The advent of high performance computing (HPC) and graphics processing units (GPU), present an enormous computation resource for large data transactions (big data) that require parallel processing for robust and prompt data analysis. In this paper, we take an overview of four parallel programming models, OpenMP, CUDA, MapReduce, and MPI. The goal is to explore literature on the subject and provide a high level view of the features presented in the programming models to assist high performance users with a concise understanding of parallel programming concepts.*

Keywords — *OpenMP, MPI, CUDA, MapReduce, GPU.*

I. Introduction

We often happen to meet problems requiring heavy computations or data-intensive processing. The increasing volume of data generated by entities unquestionably, require high performance parallel processing models for robust and speedy data analysis. With problem size and complexity increasing, the need for parallel computing has resulted in a number of programming models proposed for high performance computing. Several parallel and distributed programming models and frameworks have been developed to efficiently handle such problems.

In this paper, we take an overview of four high performance computing techniques, OpenMP, CUDA, MapReduce and MPI. Thirdly Eigen series are used to train the system and finally decision is made by means of matching.

The paper is organized in the following as follows: Section II presents a review of the latest literatures on the various high performance computing techniques. Section III represents the methodologies used in the respective high performance computing techniques. Section IV presents the comparative analysis of the high performance computing techniques. A brief conclusion is presented in section V.

II. LITERATURE REVIEW

As the high performance computing techniques have increasingly become a necessity in mainstream computing, a number of researchers have done work on documenting various features in parallel computing models. Although a number of features in parallel programming models are discussed in literature, OpenMP, CUDA, MapReduce, and MPI models, utilize multi-threading and, as such, a look at abstraction and determinism in multi-threading is given consideration in this paper.

A. OpenMP

OpenMP is a shared-memory multiprocessing Application Program Inference (API) for easy development of shared memory parallel programs [1]. It provides a set of compiler directives to create threads, synchronize the operations, and manage the shared memory on top of pthreads. The programs using OpenMP are compiled into multithreaded programs, in which threads share the same memory address space and hence the communications between threads can be very efficient. Its runtime maintains the thread pool and provides a set of libraries [3]. It uses a block-structured approach to switch between sequential and parallel sections, which follow the fork/join model. At the entry of a parallel block, a single thread of control is split into some number of threads, and a new sequential thread is started when all the split threads have finished. Its directives allow the fine-grained control over the threads [2]. Y.Charlie Hu, Honghui Lu, Alan L Cox and Willy Zwaenepoel (1999), presented the first system that implemented OpenMP on a network of shared memory multiprocessors. The system enables the programmer to rely on a single, standard shared-memory API for parallelization within a multiprocessor and between multiprocessors which is implemented via a translation that converts OpenMP directives to appropriate calls to a modified version of the TrendMarks software distributed shared memory (SDSM) system. This approach greatly simplifies the changes required to the SDSM in order to exploit the intra node hardware shared memory [12]. John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson and Carl D. Offner (2000), in their paper, describes extensions to OpenMP that implement data placement features needed for Non-Uniform Memory Access (NUMA) architectures. Writing efficient parallel programs for NUMA architectures, which have characteristics of both shared-memory and distributed-memory architectures, requires that a programmer control the placement of data in memory and the placement of computations that operate on that data.

Optimal performance is obtained when computations occur on processors that have fast access to the data needed by those computations. OpenMP—designed for shared-memory architectures—does not by itself address these issues. The extensions to OpenMP Fortran presented here have been mainly taken from High Performance Fortran. The paper describes some of the techniques that the Compaq Fortran compiler uses to generate efficient code based on these extensions. It also describes some additional compiler optimizations, and concludes with some preliminary results [13]. It is supported on various platforms like UNIX, LINUX, and Windows and various languages like C, C++, and Fortran [2].

Some of the advantages of OpenMP are-

1) OpenMP is much easier to use because the compiler takes care of transforming the sequential code into parallel code according to the directives [2].
2) The programmer can write multithreaded programs without serious understanding of multithreading mechanism [1].

B. *MPI*

MPI is a message passing library specification which defines an extended message passing model for parallel, distributed programming on distributed computing environment [4]. In their paper, Zaid Abdi Alkareem Alyasseri , Kadhim Al-Attar, Mazin Nasser and Ismail (2014), they implemented the bubble and merge sort algorithms using Message Passing Interface (MPI) approach. The proposed work tested on two standard datasets (text file) with different size. The main idea of the proposed algorithm is distributing the elements of the input datasets into many additional temporary sub-arrays according to a number of characters in each word. The sizes of each of these sub-arrays are decided depending on a number of elements with the same number of characters in the input array. They have implemented MPI using Intel core i7-3610QM, (8 CPUs), using two approaches (vectors of string and array 3D). Finally, they get the data structure effects on the performance of the algorithm for that they choice the second approach [14]. Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, Rajeev Thakur and Jesper Larsson Traff (2011), in their paper, they examine the issue of scalability of MPI to very large systems. They first examine the MPI specification itself and discuss areas with scalability concerns and how they can be overcome. They then investigate issues that an MPI implementation must address in order to be scalable. To illustrate the issues, they ran a number of simple experiments to measure MPI memory consumption at scale up to 131,072 processes, or 80%, of the IBM Blue Gene/P system at Argonne National Laboratory. Based on the results, they identified non-scalable aspects of the MPI implementation and found ways to tune it to reduce its memory footprint. They also briefly discuss issues in application scalability to large process counts and features of MPI that enable the use of other techniques to alleviate scalability limitations in applications [15].

Some of the advantages of MPI are-

1) MPI is fine matched for applications where portability in time and space is significant [3].
2) MPI is as well a brilliant selection for parallel computations and for areas involving data structures, such as unstructured mesh computations which are dynamic [3].

C. *MapReduce*

MapReduce is a programming paradigm to use Hadoop which is recognized as a representative big data processing framework. Hadoop clusters consist of up to thousands of commodity computers and provide a distributed file system called HDFS which can accommodate big volume of data in a fault-tolerant way. The clusters become the computing resource to facilitate big data processing [6]. In their paper, Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski and Christos Kozyrakis (2007), evaluates the suitability of the MapReduce model for multi-core and multi-processor systems [16]. MapReduce was created by Google for application development on data-centres with thousands of servers. It allows programmers to write functional-style code that is automatically parallelized and scheduled in a distributed system. They describe Phoenix, an implementation of MapReduce for shared-memory systems that includes a programming API and an efficient runtime system. The Phoenix runtime automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes. They study Phoenix with multi-core and symmetric multiprocessor systems and evaluate its performance potential and error recovery features. They also compared MapReduce code to code written in lower-level APIs such as Pthreads. Overall, they establish that, given a careful implementation, MapReduce is a promising model for scalable performance on shared-memory systems with simple parallel code [16]. Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao and D. Stott Parker (2007), in their paper, they improve Map-Reduce into a new model called Map-Reduce-Merge [17]. It adds to Map-Reduce a Merge phase that can efficiently merge data already partitioned and sorted (or hashed) by map and reduce modules. They also demonstrate that this new model can express relational algebra operators as well as implement several join algorithms [17].

Some of the advantages of MapReduce are-

1) MapReduce paradigm is a good choice for big data processing because it handles data record without loading whole data into memory and in addition the program is executed in parallel over a cluster [8].
2) It is very convenient to develop big data handling programs using MapReduce because Hadoop

provides everything needed for distributed and parallel processing behind the scene which program does not need to know [8].

D. *CUDA*

CUDA also known as Compute Unified Device Architecture was developed in 2006 by NVIDIA as a general purpose parallel computing programming model, to run on NVIDIA GPUs to for parallel computations [10]. With CUDA, Programmers are granted access to GPU memory and therefore, are able to utilize parallel computation not only for graphic application but general purpose processing (GPGPU) [11]. One of the challenges of HPCs is how to fully take the parallelism advantage presented by the multi-core CPUs and many-core GPUs [10]. In their paper, Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink (2010), they show that the Discrete Wavelet Transform (DWT) which has a wide range of applications from signal processing to video and image compression, by means of the lifting scheme, can be performed in a memory and computation-efficient way on modern, programmable GPUs, which can be regarded as massively parallel coprocessors through NVidia's CUDA compute paradigm [18]. The three main hardware architectures for the 2D DWT (row-column, line-based, block-based) are shown to be unsuitable for a CUDA implementation. Their CUDA-specific design can be regarded as a hybrid method between the row-column and block-based methods. They achieve considerable speedups compared to an optimized CPU implementation and earlier non-CUDA-based GPU DWT methods, both for 2D images and 3D volume data. Additionally, memory usage can be reduced significantly compared to previous GPU DWT methods. The method is scalable and the fastest GPU implementation among the methods considered. A performance analysis shows that the results of their CUDA-specific design are in close agreement with their theoretical complexity analysis [18]. Jedrzej Kowalczuk, Eric T. Psota, Lance C. Pérez (2012), in their paper, A novel real-time stereo matching method is presented that uses a two-pass approximation of adaptive support-weight aggregation, and a low-complexity iterative disparity refinement technique [19]. Through an evaluation of computationally efficient approaches to adaptive support weight cost aggregation, it is shown that the two-pass method produces an accurate approximation of the support weights while greatly reducing the complexity of aggregation. The refinement technique, constructed using a probabilistic framework, incorporates an additive term into matching cost minimization and facilitates iterative processing to improve the accuracy of the disparity map. This method has been implemented on massively parallel high-performance graphics hardware using the CUDA computing engine. Results show that the proposed method is the most accurate among all of the real-time stereo matching methods listed on the Middlebury stereo benchmark [19].

Some of the advantages of CUDA are-

1) CUDA allows use of high level language such as C to developers by providing a software environment [3].
2) CUDA programming language is designed to surmount the challenge of parallel computing by taking gain of parallelization in both CPUs and GPUs [11].

III. METHODOLOGIES

A. *OpenMP*

The OpenMP API [20, 21] defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. There are three kinds of directives: parallelism/work sharing, data environment, and synchronization. In C and C++, the directives are implemented as #pragma statements, and in Fortran 77 and 90 they are implemented as comments. OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the *master thread*. The parallel code sections are executed by all threads, including the master thread.

The fundamental directive for expressing parallelism is the parallel directive. It defines a *parallel region* of the program that is executed by multiple threads. All of the threads perform the same computation, unless a *work sharing* directive is specified within the parallel region. Work sharing directives, such as for, divide the computation among the threads. For example, the 'for' directive specifies that the iterations of the associated loop should be divided among the threads so that each iteration is performed by a single thread. The 'for' directive can take a schedule clause that specifies the details of the assignment of iterations to threads. Schedules can specify assignments such as round-robin or block. OpenMP also defines shorthand forms for specifying a parallel region containing a single work sharing directive. For example, the parallel for directive is shorthand for a parallel region that contains a single for directive. The data environment directives control the sharing of program variables that are defined outside the scope of a parallel region. The data environment directives include: shared, private, firstprivate, reduction and threadprivate. Each directive is followed by a list of variables. Variables default to shared, which means shared among all the threads in a parallel region. A private variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. A firstprivate variable has the same attributes as a private variable except that the private copies are initialized to the variables at the time the parallel region is entered. The reduction directive identifies variables. Finally, OpenMP provides the threadprivate directive for named common blocks in Fortran and global variables in C and C++. Threadprivate variables are private to each thread, but they are global in the sense that they are defined for all parallel regions in the program, unlike private variables which are defined only for a particular parallel region. The synchronization directives include barrier, critical and flush. A barrier directive causes a thread to wait until all of the other threads in the parallel region have reached the barrier. After the barrier, all threads in the parallel region have reached the barrier. A critical directive restricts access to the enclosed code to only one thread at a time. When thread enters a critical section, it is guaranteed to see all modification made by all of the threads that entered the critical section earlier. The flush directive specifies a point in the program which all threads are guaranteed to have a consistent view of the variables named in the flush directive, or of the entire memory if no variables are specified [20, 21].

B. MPI

In MPI model, each process has its own address space and communicates other processes to access others address space. Programmers take charge of partitioning workload and mapping tasks about which tasks are to be computed by each process. MPI provides point-to-point, collective, one-sided, and parallel I/O communication models [4]. Point-to-point communications enable exchanging data between two matched processes. Collective communication is a broadcast of message from a process to all the others. One-sided communications facilitate remote memory access without matched process on the remote node. Three one-sided libraries are available for remote read, remote write, and remote update [1]. MPI provides various library functions to coordinate message passing in various modes like blocked and unblocked message passing. It can send messages of gigabytes size between processes. MPI has been implemented on various platforms like Linux, OS X, Solaris, and Windows. Most MPI implementations use some kind of network file storage. As network file storage, network file system (NFS) and Hadoop HDFS can be used. Because MPI is a high level abstraction for parallel programming, programmers can easily construct parallel and distributed processing applications without deep understanding of the underlying mechanism of process creation and synchronization. To order to exploit the multicore of processors, the MPI processes can be organized to have multiple threads in themselves. MPI-based programs can be executed on a single computer or a cluster of computers [5].

C. MapReduce

MapReduce runtime launches Map and Reduce processes with consideration of data locality. MapReduce organizes an application into a pair (or a sequence of pairs) of Map and Reduce functions. Map processes are independent of each other and thus they can be executed in parallel without collaboration among them. Reduce processes play role of aggregating the values with the same key [7]. It assumes that input for the functions comes from HDFS file(s) and output is saved into HDFS files. Data files consist of records, each of which can be treated as a key-value pair. Input data is partitioned and processed by Map processes, and their processing results are shaped into key-value pairs and shuffled into Reduce tasks according to key [8, 9]. The programmers do not have to consider data partitioning, process creation, and synchronization. The same Map and Reduce functions are executed across machines. Hence, MapReduce paradigm can be regarded as a kind of SPMD model [8]. The programmers do not have to consider data partitioning, process creation, and synchronization. The same Map and Reduce functions are executed across machines. Hence, MapReduce paradigm can be regarded as a kind of SPMD model. MapReduce paradigm is a good choice for big data processing because MapReduce handles data record by record without loading whole data into memory and in addition the program is executed in parallel over a cluster [8]. It is very convenient to develop big data handling programs using MapReduce because Hadoop provides everything needed for distributed and parallel processing behind the scene which program does not need to know.

D. CUDA

The CUDA model is designed to develop applications scaling transparently with the increasing number of processor cores provided by the GPUs [23], [24]. CUDA provides a software environment that allows developers to use C as high-level programming language. For CUDA, a parallel system consists of a host (i.e.,CPU) and a computation resource or device (i.e., GPU). The computation of tasks is done in the GPU by a set of threads running in parallel. The GPU threads architecture consists in a two-level hierarchy, namely the block and the grid, see Fig. 1.

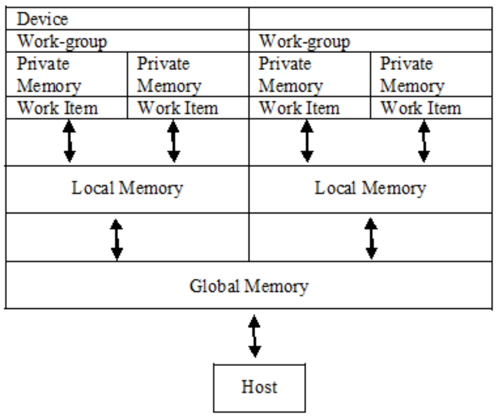

Fig. 1. CUDA (OpenCL) architecture and memory model.

The block is a set of tightly coupled threads, each identified by a thread ID. On the other hand, the grid is a set of loosely coupled blocks with similar size and dimension. There is no synchronization at all between the blocks, and an entire grid is handled by a single GPU. The GPU is organized as a collection of multiprocessors, with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple multiprocessors. Threads within a block can cooperate by sharing data through some shared memory, and by synchronizing their execution to coordinate memory accesses. More detailed information can be found in [22], [25]. Moreover, there is a best practices guide that can be useful to programmers [26]. CUDA is well suited for implementing the SPMD parallel design pattern [27]. Worker management in CUDA is done implicitly. That is, programmers do not manage thread creations and destructions. They just need to specify the dimension of the grid and block required to process a certain task. Workload partitioning and worker mapping in CUDA is done explicitly. Programmers have to define the workload to be run in parallel by using the function "Global Function" and specifying the dimension and size of the grid and of each block. The CUDA memory model is shown in Fig. 1. At the bottom of the figure, we see the global and constant memories. These are the memories that the host code can write to and read from. Constant memory allows read-only access by the device. Inside a block, we have the shared memory and the registers or local memory. The shared memory can be accessed by all threads in a block. The registers are independent for each thread.

IV. COMPARATIVE ANALYSIS

Here, parallel programming models are considered using a pure shared or distributed memory approach. As such, we consider the threads, shared memory OpenMP, and distributed memory message passing models. Table I collects the characteristics of the usual implementations of these models [28].

Table I: Pure Parallel Programming Models and Implementations

Operation	OpenMP	MPI
System Architecture	Shared Memory	Distributed and Shared Memory
Communication Model	Shared Address	Message Passing or Shared Address
Programming Model	Shared Memory	Message Passing
Synchronization	Implicit	Implicit or Explicit
Implementation	Compiler	Library
Granularity	Fine	Course or Fine

Now, for the all-pairs-shortest-path problem, three sample graphs were randomly generated of which the numbers of nodes were 10, 100, and 1000, respectively. When they were generated, each node was set to be linked to half of the other nodes. When final results are written, there can exist a bottleneck if a single file is used as the output. Hence, for the fair comparisons, each process or thread is allowed to write its output into its own out-file. Table II shows the execution time obtained in the all-pairs shortest-path problem experiments. For this computation intensive problem, the OpenMP program gave the best performance where 10 threads were used. The MPI program was executed on a single machine and on the cluster of 5 machines with total of 10 processes. Due to the computational overhead, the cluster showed poor performance for the MPI program [28]. In the experiment setting, the communication bottleneck was severe even though the machines were connected with a 1Gbps switching hub. The performance of MPI on a single machine is not comparable to OpenMP, because OpenMP threads share the global address space but MPI processes communicate using the message passing protocol. If some application can be run on a high-end single machine, OpenMP is preferred to MPI. MapReduce is not a choice for computational-intensive and iterative computation problems like the all-pairs-shortest-path problem [28].

Table II: Execution Times for the all-pair-shortest-path problem

Structure				
Node size	MapReduce	MPI		OpenMP
		Cluster	Solo Machine	
10	140s	0.32s	0.34s	0.1s
100	1010s	0.44s	0.41s	0.25s
1000	14440s	284s	24.14s	8.03s

Table III: Execution Time for the Join Problem

Problem	structure		
	MapReduce	MPI	OpenMP
The join problem	1455s	488040s	335640s

Table III shows the experiment results for the join problem. The execution time varies depending on the execution context like network bandwidth and resource management of operating systems, hence the same experiments have been conducted three times for each setting [28]. The MapReduce based program was the best one among the three models. MapReduce is the best choice for data-intensive processing of big volume of data [28]. Below are the results of tests performed in sequential and in the parallel languages for the three applications; Nqueens, Matrix multiplication and Mandelbrot [30]. These application were implemented sequentially in C and later in Cuda, OpenACC and OpenMP in parallelized form [30].

V. CONCLUSION

OpenMP, MPI, CUDA and MapReduce are the most widely recognized parallel or distributed programming frameworks. Each one is said to be the de facto standard on its computing model. If a problem is small enough to be accommodated and the computing resources such as cores and memory are sufficient, OpenMP is a good choice. When data size is moderate and the problem is computation-intensive, MPI can be considered the framework. When data size is large and the tasks do not require iterative processing, MapReduce can be an excellent framework. OpenMP is the easiest to use because there is no special attention needed to be paid because we just need to place some directives in the sequential code. MapReduce is relatively easy to use once we can abstract an application into Map and Reduce steps. The programmers do not have to consider workload partitioning and synchronization. MapReduce programs, however, take considerable time for the problems requiring much iteration, like all-pairs shortest-path problem. MPI allows more flexible control structures than MapReduce; hence MPI is a good choice when a program is needed to be executed in parallel and distributed manner with complicated coordination among processes. Generally it was found that CUDA was convenient to work with; in fact, across all the applications, able to offload work to the GPU responsible for 95.8%–99.7% of the applications' original, single-threaded execution time excluding disk I/O. It was also found CUDA far easier than traditional rendering-based GPGPU approaches using OpenGL or DirectX. CUDA's focus on available parallelism, the availability of local (perblock) shared memory, and the kernel-domain abstraction made these applications vastly easier to implement than traditional SPMD/thread-based approaches. In the case of k-means, CUDA was probably a bit more difficult than OpenMP, chiefly due to the need to explicitly move data and deal with the GPU's heterogeneous memory model. In HotSpot, with the pyramidal implementation, CUDA's ''grid-of-blocks'' paradigm was a natural fit and probably simplified implementation compared to OpenMP.

REFERENCES

[1] OpenMP Architecture Review Board, "OpenMP Application Program Interface," 2008, http://www.openmp.org/mp-documents/spec30.pdf.

[2] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, 2007, https://computing.llnl.gov/tutorials/parallel_comp/.

[3] J. Diaz, C.Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no.8, pp.1369–1386, 2012.

[4] W. Gropp, S. Huss-Lederman, A. Lumsdaine et al., *MPI: The Complete Reference, the MPI-2 Extensions*, vol. 2, The MIT Press, 1998.

[5] G. Jost, H. Jin, D. Mey, and F. Hatay, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," in *Proceedings of the 5th European workshop on OpenMP (EWOMP'03)*, 2003.

[6] J.Dean and S.Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 29–43, October 2003.

[8] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis,"Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA'07)*, pp. 13–24, Scottsdale, AZ, USA, February 2007.

[9] S. J. Plimpton and K. D. Devine,"MapReduce in MPI for largescale graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp.610–632, 2011.

[10]. NVIDIA, "CUDA C Programming Guide," no. July. NVIDIA Corporation, 2013.

[11]. Wikipedia, "General-purpose computing on graphics processing units," 2013.

[12]. Y.Charlie Hu, Honghui Lu, Alan L Cox and Willy Zwaenepoel, "OpenMp for Networks of SMPs," *Parallel Processing* , 13th International and 10th Symposium on Parallel and Distributed Processing, pp. 302-310, 1999.

[13]. John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson and Carl D. Offner, "Extending OpenMP For NUMA Machines," SC `00 Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Article no. 48, 2000.

[14]. Zaid Abdi Alkareem Alyasseri , Kadhim Al-Attar, Mazin Nasser and Ismail, "Parallelize Bubble and Merge Sort Algorithms Using Message Passing Interface (MPI)," Publication eprint arXiv:1411.5283, 2014.

[15]. Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, Rajeev Thakur and Jesper Larsson Traff, "MPI on Millions of Core," *Parallel Proceesing Letter*, vol. 21, issue 01, 2011.

[16]. Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski and Christos Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA `07), pp. 13-24, Scottsdale, AZ, USA, February 2007.

[17]. Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao and D. Stott Parker,"Map-reduce-merge: Simplified Relational Data Processing on Large Clusters," in Proceedings ACM SIGMOD international Conference on Management of Data, pp. 1029-1040, 2007.

[18]. Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink, "Accelerating Wavelet Lifting on Graphics Hardware Using CUDA," *IEEE transactions on Parallel and Distributed,* vol. 22, issue 01, pp. 132-146, 2010.

[19]. Jedrzej Kowalczuk, Eric T. Psota, Lance C. Pérez, "Real-time Stereo Matching on CUDA using an Iterative Refinement Method for Adaptive Support-Weight Correspondences," *IEEE transactions on Circuits and System for Video Technologies,* vol. 23,isuue 01, pp. 94-104,2012.

[20]. The OpenMp Forum. OpenMp Fortran Application Program Interface, Version 1.0, http:/www.openmp.org, Oct 1997.

[21]. The OpenMp Forum. OpenMp C and C++ Application Program Interface, Version 1.0, http:/www.openmp.org, Oct 1998.

[22]. CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html, Oct. 2011.

[23]. Nvidia Developer Zone, http://developer.nvidia.com, Oct. 2011.

[24]. D. Kirk and W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
[25]. Nvidia Company. Nvidia CUDA Programming Guide, v3.0, 2010.
[26]. Nvidia Company. Nvidia CUDA C Programming Best Practices Guide, Version 3.0, 2010.
[27]. T.G. Mattson, B.A. Sanders, and B. Massingill, Patterns for Parallel Programming. Addison-Wesley Professional, 2005.
[28]. Sol Ji Kang, Sang Yeon Lee, and KeonMyung Lee, "Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems, Advances in Multimedia, Research Article, 2014.
[29]. Shuai Che_, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," Published in *J. Parallel Distrib. Comput.,vol. 68,* pp. 1370-1380, 2008.
[30]. Cleverson Lopes Ledur, Carlos M. D. Zeve, Julio C. S. dos Anjos, "Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms," 11th Workshop on Parallel and Distributed Processing (WSPPD), Universidade Luterana do Brasil, Information Systems, BR 116, n. 5.724, Moradas da Colina – Guaba/RS.