On Benchmarking the Matrix Multiplication Algorithm using OpenMP, MPI and CUDA Programming Languages

Muhammed Al-Mulhem*, Abdulah AlDhamin**, and Raed Al-Shaikh**

*Information & Computer Science Department, King Fahd University of Petroleum & Minerals, mulhem@kfupm.edu.sa

**Saudi Aramco,raed.shaikh@aramco.com, abdulah.dhamin@aramco.com

ABSTRACT

Parallel programming languages represent a common theme in the evolution of high performance computing (HPC) systems. There are several parallel programming languages that are directly associated with different HPC systems. In this paper, we compare the performance of three commonly used parallel programming languages, namely: OpenMP, MPI and CUDA. Our performance evaluation of these languages is based on the implementation of matrix multiplication algorithms. Matrix multiplication is chosen because of its wide application in many scientific and engineering problems such as bioinformatics, linear algebra, and computer graphics. Our results show that CUDA programming delivers up to 15 fold speed acceleration relative to OpenMP and MPI Programming. However, CUDA programming may prove comparatively more challenging to programmers.

Keywords

Bioinformatics, HPC, OpenMP, MPI, CUDA, GPGPU, Infiniband.

1. INTRODUCTION

In recent years, we have witnessed a growing interest in optimizing the parallel and distributed computing solutions using scaled-out hardware designs and scalable parallel programming languages. This interest is driven by the fact that single CPU-chips are reaching their physical limits in terms of heat dissipation and power consumption. Therefore and as a continuation to Moore's law, recent trends in high performance and grid computing have shown that future increases in performance can only be achieved through increases in systems scale using a larger number of components, which are supported by scalable parallel programming languages. Accordingly, scaled-out computing is clearly becoming the trend.

On the parallel programming level, MPI OpenMP and CUDA have become the de facto standard to express parallelism in a program. OpenMP supports fork-and-join execution technique, where a program starts execution as a single process or thread. This thread runs sequentially until a parallelization directive for a parallel region is detected. At this stage, the thread creates a group of threads and elects itself as the master thread of the new group. All threads execute the program until the end of this parallel segment.

Another program parallelization can be achieved through the message passing interface (MPI) [1] programming, which can be employed within and across several nodes. MPI is a widely accepted standard for writing message passing programs. It provides the user with a programming model where processes communicate with other processes by calling library routines to send and receive messages.

The new GPU designs and CUDA are geared towards generic-programming methodologies, which was not possible in older generations of GPU architectures. What characterizes GPU systems is that they are fast and inexpensive when compared with conventional CPU-based clusters. For example, the newest nVidia GTX 580 card is able to deliver a theoretical 1.5 Tflops at around $500 [4].

Our objective in this paper is to evaluate the three commonly used parallel programming languages, by implementing the parallel matrix multiplication algorithm and benchmarking the performance and scalability of the three implementations. We choose matrix multiplication because of its wide use in bioinformatics applications such as sequence alignment [6], analysis of microarray data [7], phylogenetic [8], MHC binding [9], motif finding [10], DNA protein binding [11]. To the best of our knowledge, this is the first paper that systematically benchmarks the three programming languages on the latest Intel's Westmere technology and Infiniband QDR interconnect.

The rest of the paper is organized as follows: In section 2, we show the performance evaluation and results. In section 3 we present the programming models used in this study. The last section states our conclusion and future work.

2. PERFORMANCE EVALUATION AND RESULTS

In our performance evaluation for OpenMP and MPI programming, we used a DELL cluster powered by PowerEdge M610 Blade Servers. The cluster consisted of 32 nodes with 2x sockets and Intel Hex-Core (Westmere) 2.93GHz processors. Compute nodes were running RedHat Enterprise Linux Server 5.3 as the cluster operating system. Each node contained an Infiniband Host Channel Adapter (HCA) supporting 4x Quad Data Rate (QDR) connections with the speed of 32Gbps. Each node also had 24 GB (6 x 4GB) DDR3 1333Mhz of memory, thus the total amount of memory the system had was around 786GB.

For the CUDA programming we used the GPGPU system equipped with GeForce GTX460 OC XLR8 black box with Nividia GF104 Fermi graphics processor. It has 1 GB memory, 336 cores and 256 bit memory bus. It is installed on HP Z800 workstation with Intel Xeon six-core processors X5680, 8GB 1333 MHz DDR3 ECC unbuffered RAM and 2 TB SATA/300 N6Q hard disk.

In our experiments, we used three versions of matrix multiplication codes [2, 3, 5] to benchmark the three programming languages. Although, it is computationally intensive with $O(n^3)$ iterations, we chose the matrix multiplication since it is an integral part of many numerical linear algebra and bioinformatics applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries.

Figure 1 shows the OpenMP performance benchmark for multiplying 4000x4000 and 5000x5000 size matrices. Initially, all runs were significantly improved when adding more cores, while their improvement slowed down when reaching 6 cores.

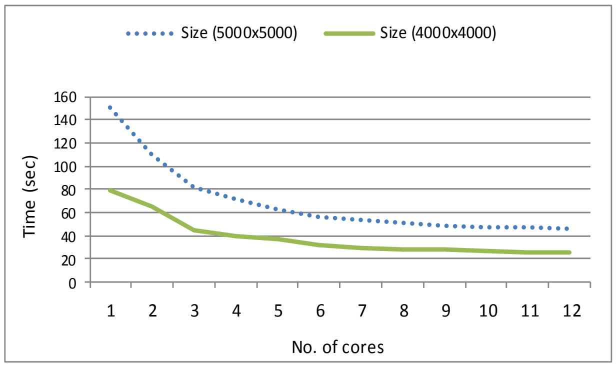

Figure 1: OpenMP implementation.

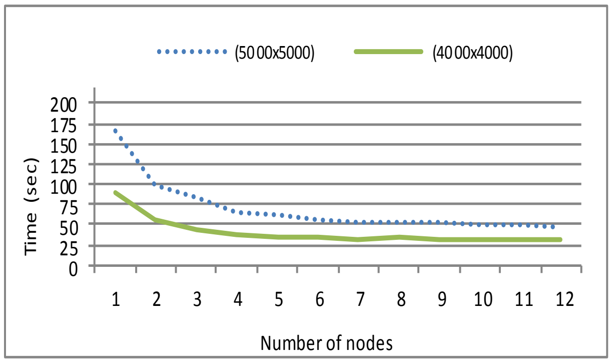

Figure 2: MPI implementation.

Figure 2 shows the matrix multiplication performance using MPI programming. In this test, the code was compiled using MVAPICH MPI libraries. It is noticeable that the code on a single node/core took around 165 seconds, whereas it took only 150 seconds when running OpenMP on a single core. This is due to the fact that the MPI-based matrix multiplication C code has more routines and functions to call, making the code more complex, and thus more time to run. Another observation is the slight increase in the run time of the MPI compared with OpenMP when multiplying the 4000x4000 size matrices on 11 and 12 cores. This increase is related to the additional communication overhead with respect to the computation time. This communication is lessened in the 5000x5000 multiplication as the computation time gets larger with respect to the communication overhead.

To magnify the effect of MPI communication overhead with respect to computation time, we extended the MPI matrix multiplication benchmark runs to 32 nodes. Figure 3 shows the effect of this communication overhead as the number of nodes increases.

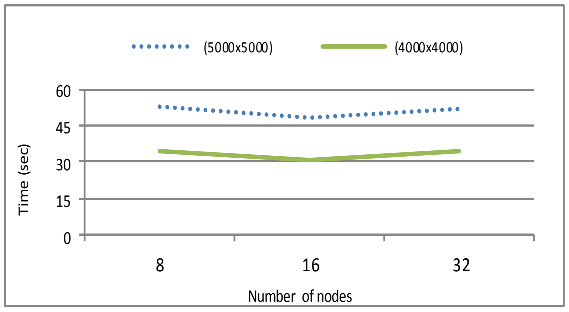

Figure 3: MPI scalability in 5000x5000 and 4000x4000 matrix multiplication using up to 32 nodes.

Table 1 shows the CUDA matrix multiplication runs on the GPGPU hardware. It shows that CUDA programming delivers up to 15 fold speed acceleration relative to OpenMP and MPI programming. In comparison with the CPU threads in OpenMP, threads in CUDA are grouped into blocks, while blocks are grouped into grids. A kernel is executed as a grid of blocks of threads, given that threads in a block may cooperate by having shared memory accesses and also sharing the results.

Table 1: CUDA implementation.

Matrix size	Blocks/Grid	Threads/Block	Time (seconds)
4000	16	16	1.732
5000	8	16	4.911

3. THE PROGRAMMING STRUCTURE

In the field of HPC, the current hardware trend is to build clusters of complex programming environments and use parallel libraries, such as OpenMP, MPI and GPGPU programming. In this section, we compare the programming structure and complexity of the mentioned programming paradigms.

1. OpenMP

Open Multi-Processing programming, commonly known as OpenMP, is one of the major and widely used application program interfaces (APIs) to develop task-parallel applications, but can also be used to achieve data-parallelism through invoking work-sharing constructs. OpenMP supports both shared memory multi-core systems and distributed shared memory systems, though it is not meant for distributed-memory systems by design.

Unlike pthreads, which is also designed to support shared-memory architecture, OpenMP is considered as a high-level extension to the standard ANSI C. It is composed of three API components: compiler directives, runtime routines, and environment variables. The directives define the portion of the code in the program that the developer wants to parallelize. Then, the compiler uses runtime library routines to explicitly tell the machine to perform the tasks. The tasks of thread creation, memory management, communication between threads and synchronization are all being taken care of by the compiler [2]. Hence, it is possible to have a C compiler that cannot compile OpenMP programs into parallel programs.

The primary advantage of using OpenMP in parallelization is the fact that it allows developers to incrementally parallelize serial programs rather than re-writing them from scratch.

2. MPI

Message Passing Interface (MPI) is a set of parallel programming APIs, designed to develop parallel programs using high-level programming languages, most commonly used is the standard ANSI C, for distributed-memory systems. Because of the nature of this environment, the interconnection networks plays a critical role on the performance of the program, which is the core medium for transferring messages between threads [3]. That said, the bandwidth and latency of the interconnects are very critical measures of how well the parallel application will perform with MPI.

The key concept of the MPI is to provide a methodology of sharing data between processes or threads that do not share physical memory space, whereas the basic functions of the MPI is the send and receive methods between processes. Nevertheless, distributed-memory systems can also programmed to allow threads access memory regions belonging to another thread. This approach is called one-sided communication. A key element of the MPI model is a communicator, which is a set of processes that can send and receive messages between each other. MPI Collective Communication modes consist of: Tree-structured, Collective communication, Point-to-point communication, Broadcast and Data distribution.

3. GPGPU

General-purpose computing on graphics processing units (GPGPU) is a programming model meant to use the graphics processing units (GPUs) as a co-processor to speed up the CPU performance for general-purpose scientific and engineering computing [5]. Specifically, GPUs takes custody of compute-intensive and time-consuming portions of the code off the CPU, while leaving the CPU to compute the rest of the application. This programming model is a typical data-parallelism approach specifically designed to support the Single Instruction Multiple Data, or SIMD, model.

The fact that GPUs are designed specifically for graphics implies a certain restrictions on what operations and programming they provide. To get the best out of this approach, only computing problems that can be solved through stream processing will benefit the most. Stream processing is the programming paradigm that allows applications to exploit the parallelism by using other computational units, other than the CPU. In GPGPU paradigm, the application is divided into chunks of streams and kernels. Kernels are the functions applied to each element of the stream. In addition, because GPUs process elements in each stream independently, there is no support of shared or static data. On the other hand, the GPGPU model is differentiated from OpenMP and MPI, in that it utilizes a second computation unit that can scale to larger number of threads.

The SIMD nature of the GPUs suggests some common characteristic about GPGPU applications is that they have large datasets, high parallelism with no or very minimal branching, and minimal data dependency between elements.

The difference in the performance gain in GPU compared to CPU is based on the design philosophies between multi-core CPUs and the GPUs. While the former is designed to optimize the performance of sequential code in which cache memories play essential part to reduce instruction and data access latencies, the latter is optimized for parallel execution. In addition, GPUs deliver huge memory bandwidth, up to 10 times the bandwidth of CPU chips, as well as a significant communication bandwidth with the CPU, which is expected to increase as the CPU bus bandwidth grows in the future.

In addition, the memory hierarchy in the GPGPU environment is different from the CPU computing, with three distinct layers:

1. Global memory; to serve the grid I/O.
2. Shared memory; to serve thread collaboration/communication.
3. Registers; dedicated for thread space.

There are several programming platforms for the GPGPU, most commonly used are CUDA by NVIDIA, DirectX by Microsoft, and the open source OpenGL. All of which share the same high level structure of program with differences in the specifications.

4. CONCLUSION

New multi-core processors represent one of the biggest advances in parallel computing. This advancement is driven by the fact that the microchip technology is gradually reaching its physical limitations in terms of heat dissipation and power consumption. In this paper, we compare the performance of three commonly used parallel programming languages used in HPC systems, namely: OpenMP, MPI and CUDA programming. The performance evaluation is based on running different sizes of the conventional matrix multiplication algorithm. Our results show that while GPU programming delivers unparalleled levels of performance, it poses a significant challenge for developers, in the way that it is still complex and error-prone, compared to programming on general-purpose CPUs and using parallel programming models such as OpenMP.

Acknowledgment
The authors would like to thank King Fahd University of Petroleum & Minerals and Saudi Aramco for their support.

REFERENCES

[1] Liu J., et al. "Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics", Supercomputing, ACM/IEEE, pages 58- 58, Nov. 2003.

[2] Lawrence Livermore National Laboratory – OpenMP tutorial. Available at: https://computing.llnl.gov/tutorials/openMP/

[3] Simple matrix multiplication on MPI. Available at: http://sushpa.wordpress.com/2008/05/20/simple-matrix-multiplication-on-mpi/

[4] Ryoo S., et. al., "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA", Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 73-82, Feb. 20-23, 2008. Salt Lake City, Utah, USA.

[5] GPU Nvidia, Available at: http://www.nvidia.com/object/what-is-gpu-computing.html

[6] Russo L., "Monge properties of sequence alignment", Theoretical computer science 423 (2012), pages 30-49.

[7] Moloshok T., et.al., "Application of bayesian decompostion for analysing microarray data", Bioinformatics, vol. 18, no. 4, 2002, pages 566-575.

[8] Suchard M., et. Al., "Many-core algorithms for statistical phylogenetics", Bioinformatics, vol. 25, no. 1, 2009, pages 1370-1376.

[9] Kim Y., et. Al., "Derivation of an amino acid similarity matrix for peptide: MHC binding and its application as a Bayesian prior", BMC Bioinformatics 2009, 10:394, pages 1-11.

[10] Ribeca P. and Raineri E., "Faster exact Markovian probability function for motif

occurrences: a DFA-only approach", Bioinformatics, vol. 24, no. 24, 2008, pages 2839-2848.

[11] Teif V., "General transfer matrix fomalism to calculate DNA-protein-drug binding in gene regulation: application to OR operator of phage A", Nucleic Acides Research, vol. 35, no. 11, 2007, pages 1-18.