ISSN:0975-9646

From Moore's Law to Amdahl's Law: The Pursuit of Computer Performance

Chakib Chraibi
School of Engineering and Technology,
Miami Dade College, Miami, Florida, USA

Abstract-As the increasing reliance on information technology and computer power, to make information systems more efficient or advanced research more possible, has become so pervasive, the underlying technology, based on Moore's law that made it possible, is coming to an end, at least in its present model. This paper explores alternative computer frameworks and architectures to increase the speed and power of computing, namely the multicore and the many-core models.

Keywords-Computer Performance, Parallel Architecture, Parallel Programming, Moore's Law, CUDA

INTRODUCTION

In the 1960s, Gordon Moore of Intel stated that the number of transistors, which can be manufactured on a single integrated-circuit die, will double every 18 months. This prediction, known now as Moore's law, has been validated by four decades of empirical results as shown in Figure 1 [1]. The growth is expected to slow down by 2013, after which density will only double every three years.
During this sustained period, software performance had essentially relied on guaranteed hardware capacity growth enshrined in Moore's law. This reliance on hardware density is predictably coming to an end.
The main current existing alternative is to use parallel architectures. Multi-core processors, consisting on two or more independent processors, are common in all types of computing devices. The effect on computer performance can only be harnessed by leveraging software algorithms and their implementation. Although near linear speed-up may be achieved in some cases such as embarrassingly parallel problems, most cases are limited by the fraction of the algorithm that can be parallelized as stated in Amdahl's law. This law describes the expected speed-up of parallelization as constrained by the serial portion of the program. In the case of parallelization, Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

For software developers, the focus now should be in developing software algorithms that would take an optimal advantage of multi-core architectures. The future of computer performance growth lies in the parallelization of software. In this paper, we run several algorithms of different complexities and discuss the advantage of parallel models over serial models.

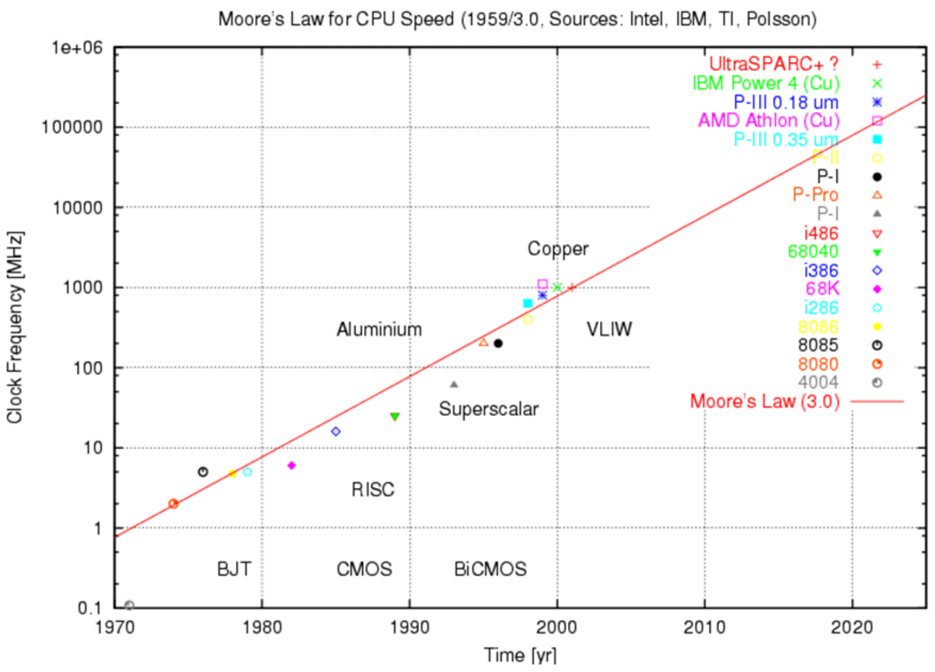

Figure 1. Moore's Law for CPU Speed

The goal of this study is to explore the two main approaches for designing multiprocessor architectures. The first trajectory is known as the *multicore* model. For instance, the Intel® Core i7 microprocessor has four processor cores, each implementing the fullx86 instruction set and focuses on the execution speed of sequential programs [2]. The other model is referred to as the *many-core* model and focuses on the execution throughput of parallel applications. An example is the NVIDIA® GeForce® GTX 280 graphics processing unit (GPU) with 240 cores.

PARALLEL COMPUTING USING A MULTICORE MODEL

In this study, we will run algorithms in the serial and parallel models and compare their performance with the goal of transforming the way we seek increased computer performance. In theory, serial algorithms have a slight reduction of overhead compared to parallel algorithms, because parallel algorithms must divide the execution among processors as they proceed, whereas serial algorithms do not. However, aside from that, parallel algorithms have the advantage of being able to take advantage of the parallelization of the workload.

In our experiments, we will use a quad-core machine and "parallel-friendly" algorithms. Theoretically, the maximum speedup is quadruple the amount of work as the serial algorithm if there is no dependency between computations. Now, since those multi-core machines are affordable and accessible, software solutions should be designed to fully exploit parallel hardware. Most applications developed today were intended to run on a single core and see no speed improvements when run on multi-core machines. On the other hand, overexposed parallelism can lead to slower performance by creating unnecessary synchronization and race condition. A systematic methodology should be developed to guide programmers in designing parallel prone software programs.

Recently, several programming environments have been extended to provide parallel programming support. For instance, Visual Studio 2010 and the .NET Framework 4 provide support for parallel programming across cores by providing a new runtime, new class library types, and new diagnostic tools [3, 4]. These features make it easier to do parallel development so that efficient, fine-grained, and scalable parallel code can be written without having to work directly with threads or use specialized languages or platforms such as OpenMP.

In what follows, we will contrast serial execution and parallel execution on a multicore machine using matrix multiplication algorithms. This algorithm is important because it exhibits a central programming structure to parallel programming, the parallel loop. The application is run in the Visual Studio 2012 environment.

EXAMPLE: MATRIX MULTIPLICATION ALGORITHM

The following algorithm implements the matrix multiplication operation. Matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. The matrix product of two matrices can be defined when the number of the columns of the first matrix matches the number of the rows of the second matrix. The product of an $m \times p$ matrix A with a $p \times n$ matrix B is an $m \times n$ matrix denoted AB whose entries are where $1 \le i \le m$ is the row index and $1 \le j \le n$ is the column index.

$$(AB)_{i,j} = \sum_{k=1}^{p} A_{ik} B_{kj},$$

As we can notice from the formula, most operations may be independently executed but then need to be synchronized to complete the final addition of the different factors. These operations are also repetitive and require a loop control construct.

We chose this algorithm because it can be characterized as embarrassingly parallel. In parallel computing, an embarrassingly parallel algorithm is a program that requires little effort to be modularized into parallel tasks because its many operations may be performed in relative independence, with few or no dependencies between these parallel tasks. A program fitting this pattern offers the best chance for efficient parallel execution, based on Amdahl's law.

Matrix multiplication has several applications, for instance, in business applications and supply management. The programs, used to test the serial and parallel implementation models of matrix multiplication, were written in Visual Basic. In the parallel algorithm, the loops were written using the parallel construct now supported in the .NET environment. In order to test performance between the two algorithms, the algorithms were run on increasing matrix sizes. The tests were performed on an Intel Core 2 Quad, running in a 32-bit operating system environment. This particular processor has four cores within one die, and is clocked at 2.4 GHz. processors. All tests were run on the same hardware, one after one another. Because of uncontrolled external factors (i.e.. other processes running on the system, all results should be considered with an error factor of 0±0.1 second).

Results of our simulations are shown in Table 1 and Figure 2.

Table 1. Matrix Multiplication Serial vs. Parallel

Matrix Size	Serial Time (sec.)	Parallel Time (sec.)	Gain/Loss (sec.)	Improvement
100	0.035	0.019	0.016	46%
200	0.288	0.068	0.22	76%
300	0.983	0.231	0.752	77%
400	2.815	0.528	2.287	81%
500	6.184	1.063	5.121	83%
1000	51.718	7.577	44.141	85%

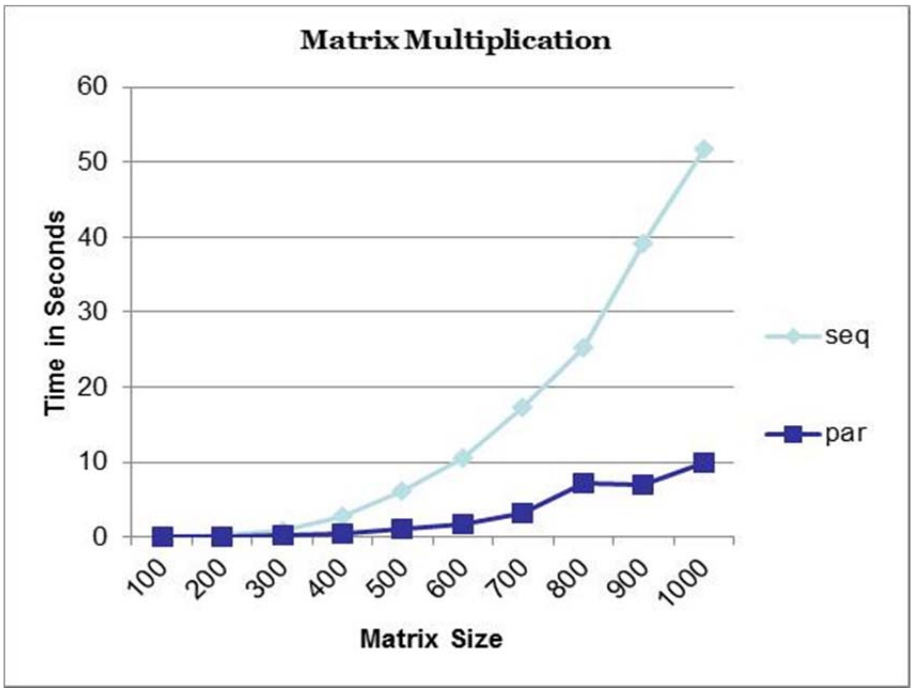

Figure 2. Matrix Multiplication: Serial vs. Parallel Model

The results of our simulations show that computer performance can be significantly improved on a regular machine by transforming the program from a serial model into a parallel model so it takes advantage of the parallel hardware infrastructure, especially when operations are repetitive as in matrix multiplication. While Moore's law provides a 58% improvement over two years, we were able to get over 80% improvements for moderately large matrices, in our example. In matrix multiplication, only a tweaking of the loop construct is required. A significant majority of the work in many applications and algorithms is done through loop control constructs.

PARALLEL COMPUTING USING A MANY-CORE MODEL

Graphic Processing Units or GPUs have been increasingly used to support applications in science and engineering that require large amounts of computer power while staying reasonably affordable. CUDA (Compute Unified Device Architecture) is one the most popular and convenient application programming interfaces used to harness the power of GPUs by efficiently and relatively easily launching multiple compute kernels or threads on the GPU. CUDA, invented by NVIDIA, is a parallel computing platform and programming model that draws its computer power through the production and implementation of GPUs, using programming languages such as C or C++.
The fast growing video industry has exerted a lot of influence on the design of the GPU programming model (Kirk 2010). Video game applications require the capability of executing a massive number of floating point calculations. The GPU-based model can sustain thousands of threads. The design philosophy of the GPU programming model is to build a large number of parallel computing units that are small, simple, and power efficient. The base hardware is heterogeneous combining two types of processors: the CPU and the GPU. The experiment shown below is run using CUDA. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model created by NVIDIA® and implemented by the graphics processing units (GPUs) that they produce. CUDA allows us to program both processors with one program. This model permits to harness the power of GPUs in our programs while the control is still initiated by the CPU. CUDA supports many languages, but in our example, we will be using the C language.

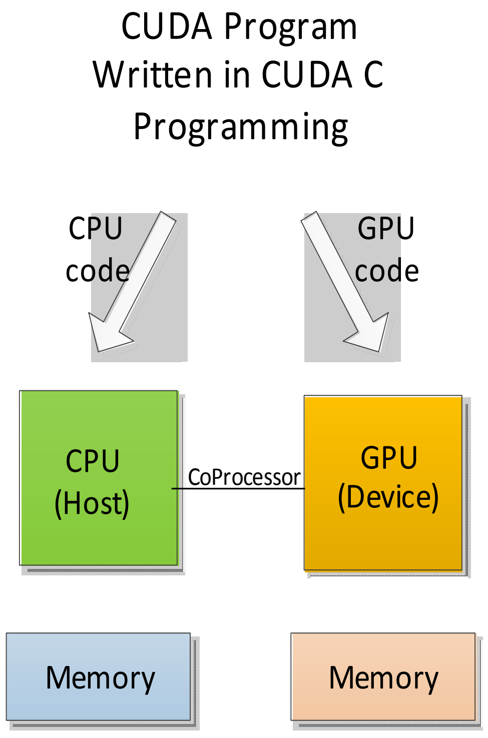

Figure 3. GPU Programming Model

Figure 3 illustrates the GPU programming model. The typical CUDA program will look like a regular program as if it runs on a single thread, but when it is invoked, the "CUDA" kernel will be executed through multiple threads. The CPU allocates storage on the GPU and copies some input data from the CPU to the GPU. Then, the CPU launches the threads to be executed on the GPU. The results are finally copied back from the GPU to the CPU.
Figure 4 shows the simulation results of the execution of the matrix multiplication example executed with a sequential C program vs. a CUDA C program.

Results of our simulations are shown in Table 2 and Figure 4. The results of our simulations show that in small size computation, the CUDA performance suffers from the operations required to set up the transfer between the CPU and GPU. However, eventually, as the matrix size grows, CUDA provides a significant improvement. We were able to reach a 76% improvement ratio that surpasses the 58% typically provided by Moore's law.

Table 2. Matrix Multiplication Serial vs. Parallel

Matrix Size	C Serial Time (sec.)	CUDA C Parallel Time (sec.)	Gain/Loss (sec.)	Improvement
100	0.001	0.81	0.016	-800000%
200	0.03	0.84	0.22	-2700%
300	0.11	0.93	0.752	-745%
400	0.33	0.98	2.287	-197%
500	0.59	1.17	5.121	-98%
1000	7.7	1.85	44.141	76%

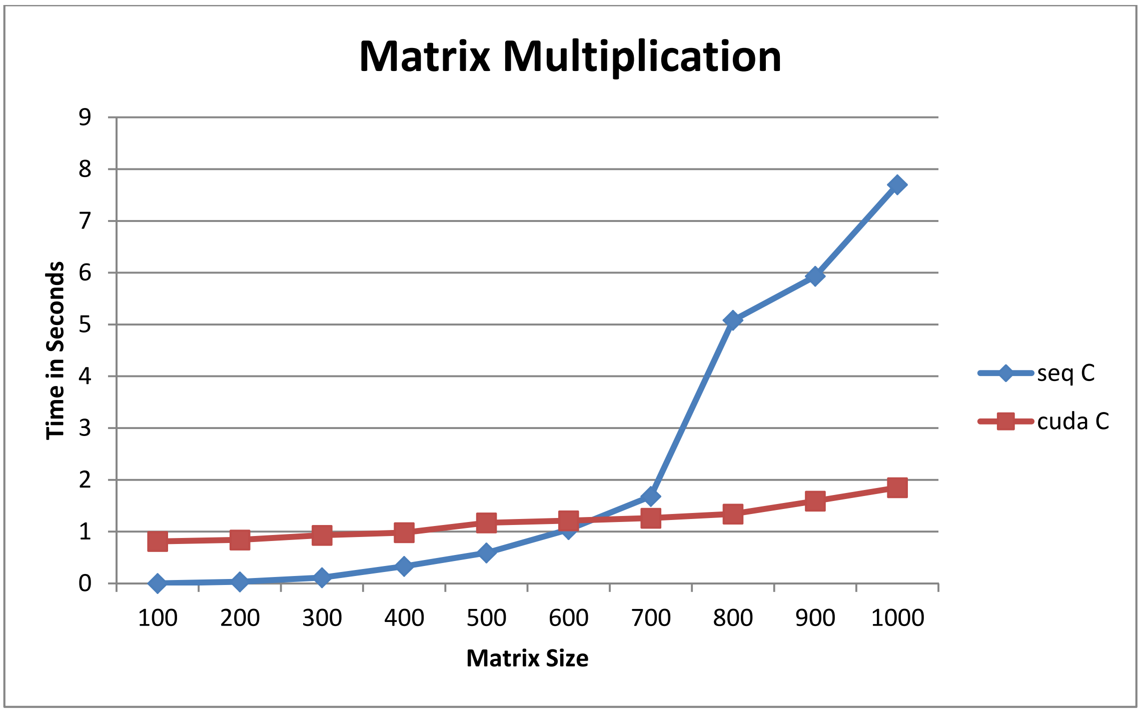

Figure 4. Matrix Multiplication: Serial vs. Parallel Model

Conclusion

Parallel programming is about optimizing the performance of applications executing on multiple cores by maximizing processor usage across all available cores. Microsoft Visual Studio development system now provides a development model that supports parallel programming through the Parallel Patterns Library (PPL) and the Asynchronous Agents Library. This simplifies tremendously the job because sophisticated algorithms that dynamically distribute computations on multicore architectures.

Based on our multicore and many-core programming models' simulation results, parallel programming provides clear improvement of computer performance. The speedups realized in multi-core equipped machines are directly correlated to the problem size. As the problem size gets larger, parallel execution overcomes the cost of splitting the code on two or more processors. This trend should hold for larger multicore and speedups will be more significant. The results of our simulation using a many core architecture based on the CUDA programming model comfort the power of parallel computing. However, applications consist of sequential and parallel parts. Speedup will depend on the parallel sub-structure that is suitable for parallel execution, as stated in Amdahl's law.

However, parallel programming is inherently more complicated that sequential programming because additional issues may arise such as synchronization, deadlock, and load balancing. The ideal situation is when program codes can be divided into threads that can be completely implemented in parallel, that is, one thread per core. This rarely happens because threads, during execution, may need to access data or wait for the execution of data, and thus synchronize with other threads. All these operations delay the proper execution and slow down the performance of the program.

High performance can only be achieved through proper, correct, smart modularization and programming. This should encourage software developers to learn more about parallel computation and programming issues, and work on patterns that can be automatically used to take full advantage of multi-core processors architectures. Several parallel programming patterns have been proposed to enhance and automatize the decomposition of serial programs into parallel programs. The focus should be on performance analyzers at both the design and run-time levels. These tools should be used to monitor the performance and identify any gap in optimizing the use of parallel execution. The adage "what the hardware gives, the software takes away" is fading away.

REFERENCES

[1] Dongarra, J. P., et al. (2003) "*Sourcebook of Parallel Computing"*, San Francisco, CA: Morgan Kaufmann.

[2] Kirk, D.B., and Hwu, W.W. (2010) "*Programming Massively Parallel Processors"*, Burlington, MA: Elsevier.

[3] Campell, C., and Miller, A. (2011) "*Parallel Programming with Microsoft Visual C++"*, Redmond, WA: Microsoft Press.

[4] Marshall, D. (2011) "*Parallel Programming with Microsoft Visual Studio 2010"*, Redmond, WA: Microsoft Press.

AUTHOR

Dr. Chakib Chraibi is currently a faculty in the School of Engineering and Information Technology at Miami Dade College in Miami, USA. Previously, he was the Associate Dean in the College of Engineering and Information Sciences at DeVry University and the Chair of the Department of Mathematics and Computer Science at Barry University. His research interests include real-time systems, enterprise computing, software engineering, computer networking and security, and computer performance.