

Jen Ada Distributed and Cluster Computing Assignment 1

by Jennifer Ada

FILE	3017_JENNIFER_ADA_JEN_ADA_DISTRIBUTED_AND_CLUSTER_COMPUTING_ASSIGNMENT_1_94244_319360563.DOCX (70.61K)		
TIME SUBMITTED	23-JAN-2018 11:43PM (UTC+0000)	WORD COUNT	2144
SUBMISSION ID	82294697	CHARACTER COUNT	14056

DISTRIBUTED AND CLUSTER COMPUTING ASSIGNMENT 1 – JEN ADA

Contents

The historical context behind the use of parallelism.....	3
Multicore processors, and the implication this has on other hardware.	4
Cache.....	4
Northbridge and Southbridge	5
Description of current parallel computing paradigms and tools available.....	6
Threading	6
Cluster processing and MPI.....	6
GPGPU and CUDA	7
Benefits and limitations of current parallel computing paradigms and tools.	7
Threading	Error! Bookmark not defined.
The potential pitfalls and limitations of parallel computing.....	8
Race conditions	8
Deadlocks	Error! Bookmark not defined.
Amdahl's law.....	8
References.	9
Appendix:	10

The historical context behind the use of parallelism.

As processors have become faster they are heat dissipation capabilities which limit clock speed. This has required increasingly complex components which can operate in parallel sometimes on specialised processes such as vector math and new approaches to designing software to take advantage of the innovations in hardware design.

Until 2006 the speed of processors followed Moore's Law and was exponentially increasing. Approximately every 18 months processors speed doubled[1] a slower rate of progress than in the 80's. Continuously rising CPU clock speeds made sequential software perform faster with each new hardware generation but higher clock speeds require more power and generate more heat[2]. By 2005[2] clock speeds reached the thermal dissipation limits of a few square centimetres of silicon. CPU architects have not significantly increased clock speeds since 2005, but the number of transistors that can be placed on a chip has continued to rise[2].

By 2005, the focus on miniaturisation and integration of components led to multi-core processors emerging as a response to energy consumption and heat dissipation limits being reached. Initially with two or four cores and constantly increasing. The growth of processor speed continued but the reason behind the increased performance was parallelism taking advantage of additional processor cores[1].

Multicore processors, and the implication this has on other hardware.

Architects increased parallelism by putting more CPU cores on each chip[2]. Power and frequency limitations observed on single core implementations have paved the way for multi-core technology and there is considerable amount of research going on in the field to utilize multi-core processors more efficiently[3].

CPU performance is increasing rapidly[4], the number of cores on the chip increases at each release of a new generation of a CPU[5] and multicore CPU production is expected to continue to increase to meet future demand.

The next iteration of multicore processors is an asymmetric multicore processor(AMP). It uses cores of different types in the same processor and a prioritisation of heterogeneity. Different cores in an AMP may be optimized to save energy or maximise performance. It can also use different application-domains or for exploiting instruction-level parallelism, thread-level parallelism or memory-level parallelism.

The new problem introduced by AMP is that as the magnitude of heterogeneity increases, either due to large number of core types or applications, diverse instruction set architectures or the range of parallelism they seek to exploit, the complexity of managing them escalates at an exponential rate. For this reason, conventional symmetric multicore processor optimization techniques may not work well in an AMP[6].

The industry has continued to innovate on multicore processor design and this has implications for the way both software and hardware are designed.

Cache

The main implication multicore processors have had on the surrounding architecture of the processor is the implementation of ever increasing levels of cache to ensure the processor is fed with enough work, data from storage and memory, to justify increased performance capabilities. Multicore architecture necessitates multiple levels of cache[7] because although Moore's Law has been beneficial to the performance of processors and memory, the connections between the two have not kept pace resulting in a bottleneck as the processors fetch and push data to and from memory. Multicore processors can have varying levels of shared cache which further improves performance where the data needs to be accessed by more than one core[7].

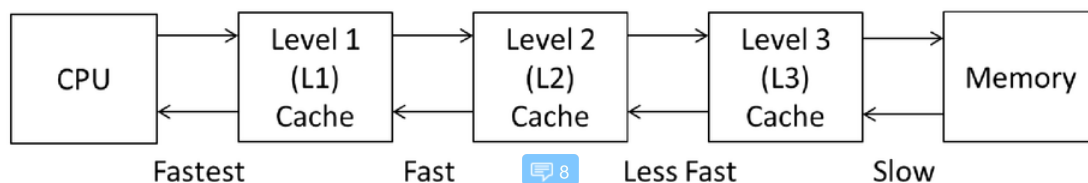


FIGURE 1 THREE-LEVEL CACHE ORGANIZATION.

Northbridge and Southbridge

The bridge was introduced to increase performance between the processors and memory (North), and Northbridge and bus to external devices (South). This is hierarchical like the cache so that the further away from the CPU a component is, the slower the clock speed.

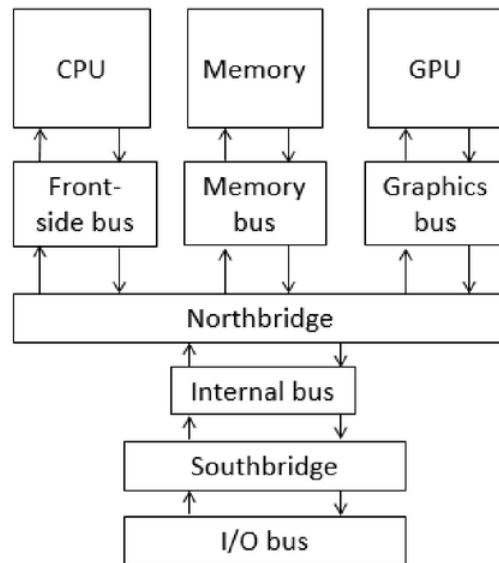


FIGURE 2 HIGH-LEVEL EXAMPLE OF CHIPSET.

Description of current parallel computing paradigms and tools available.

Threading and Multicore Processors

All threads are attached to a process and conversely all processes have a main thread of execution. The development of multiple cores on a single CPU meant that multithreading became an important method of improving program performance[8].

An example that demonstrates different resolutions of threading can be found in the problem domain of computer games. The game developer Valve created the Source game engine which used three levels of granularity for threading.

Coarse threading was designated to anything that would be assigned to individual processors and all coarse threads are synchronised by a timeline thread necessary due to the time sensitive nature of games. Fine-grain threading was allocated to tasks which were the same and occurred often such as loops which could be split up and distributed among multiple cores. The engine was referred to as using a hybrid threading system where tasks such as sound which required little interaction were allocated single threads whereas rendering was allocated multiple threads as the user interacted with that element more frequently[9].

An example of a thread in C# from the API reference can be found in the appendix[10].

Unlike other parallelism methods threading has the advantage of having every thread able to access to the same memory, meaning that complex inter-process communication as seen in Clusters and MPI is usually unnecessary[8].

Cluster processing and MPI

A cluster refers to multiple pieces of hardware connected as nodes in a network[8][7].

There are several benefits to using a cluster computer for parallelisation[11]. To a certain extent the cluster can continue to grow by simply adding more nodes to the cluster. This makes upgrades to performance cheap. Clusters have been built using video game consoles for this exact reason. Redundancy also becomes less of a concern because MPI can be used to take advantage of API's such as MPI[12] which mean that if a node fails such as in a Beowulf cluster, the master node can allocate the work to a different slave node.

GPGPU and CUDA

NVIDIA and AMD created parallel processing devices with many cores. Initially they were used mainly for graphics applications and called Graphics Processing Units (GPU). Later they became useful for general purpose applications (GPGPU) beyond graphics and visualisation due to them sharing the same operations for example vector and matrix operations[1].

Whereas a CPU is a general purpose processor, GPUs processing concerns itself in reference to Flynn's Taxonomy[4][8][7], to Single Instruction Multiple Data processing (SIMD), in contrast a single processor which would handle one instruction at a time be classified under the taxonomy as SISD. A multicore processor deals with multiple instructions and is therefore referred to mostly as a MIMD processor standing for multiple instruction multiple data due to its general-purpose nature.

Modern GPUs include a very large number of Algorithmic Logic Units (ALUs) and run many threads, each with concurrent memory access to perform the same operation on different parts of the data simultaneously hence its SIMD designation[8].

10

Unlike previous generations that partitioned computing resources into vertex and pixel shaders, the CUDA Architecture included a unified shader pipeline, allowing every arithmetic logic unit (ALU) on the chip to be organized by a program intending to perform general-purpose computations.

Because NVIDIA designed these graphics processors for general purpose computing, these ALUs were built to comply with floating point arithmetic standards from the IEEE and contained a general instruction set not solely dedicated to graphics processing. The execution units on the GPU are allowed arbitrary read and write access to memory as well as access to a software-managed cache known as shared memory.

All of these features of the CUDA Architecture were added in order to create a GPU that would excel at computation as well as being backwards compatible and allowing traditional graphics tasks to be performed also by using a block to parallelise kernels[13]. CUDA kernels are program which are run on threads. These threads are run in blocks[13]. This architecture has several benefits including memory optimisation.

11

12

The potential pitfalls and limitations of parallel computing.

Race conditions and Deadlocks

A race condition occurs when two instructions which should function separately manage to influence both or one instructions operation. To put it another way multiple threads accessing shared data simultaneously may lead to a timing dependent error known as a race condition.

If a second core accessing data in memory before the first core finishes updating memory, the secondary core will have the wrong data and so the instruction will produce the wrong result. The part of the code which can induce a race condition is referred to as a critical section and can be protected using techniques which restrict access to the memory to protect which is demonstrated by the mutex in the C# System.Threading namespace.

Conversely When two threads wait for each other to access memory and are therefore stuck not executing, a deadlock is created. These can be mitigated by checking at intervals that this has not occurred.

Amdahl's law

A limitation on how much of a program can have its performance increased through parallelisation is summed up in Amdahl's Law. For example, doubling the number of cores that a multithreaded application can run on would double the speed with which it is executed. But if only 10% of the program can be threaded then doubling the number of cores leads to only a 5% increase in speed. Taking latency from memory access into account this hypothetical program could see less than a 5% increase in speed from investing in a new processor with double the cores. Techniques which mitigate Race Conditions and Deadlocks can also contribute to the margin in performance.

References.

- [1] J. Kowalik, P. Arlukowicz, and E. Parsons, "Speeding Up Computers," Mar. 2016.
- [2] and N. Z. Austin T. Clements, M. Frans Kaashoek, Eddie Kohler, Robert T. Morris, "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors," *Commun. ACM.*, vol. 60, no. 8, pp. 84–90, 2017.
- [3] B. Venu, "Multi-core processors – An overview," p. 6, 2011.
- [4] I. R. Chhibber and R. B. Garg, "Multicore Processor, Parallelism and Their Performance Analysis," *Int. J. Adv. Res. Comput. Sci. Technol.*, vol. 31, no. 3, 2014.
- [5] G. E. Moore, "Cramming more components onto integrated circuits.," *Electronics*, vol. 38, no. 8, 1965.
- [6] S. Mittal and Sparsh, "A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–38, Feb. 2016.
- [7] W. Stallings, *Computer Organization and Architecture Designing for Performance*. Pearson, 2013.
- [8] T. Bashford and K. Donne, "PARALLELISATION TECHNIQUES FOR THE DUAL RECIPROCITY AND TIME-DEPENDENT BOUNDARY ELEMENT METHOD ALGORITHMS," *Int. J. Comp. Meth. Exp. Meas*, vol. 5, no. 3, pp. 395–403, 2017.
- [9] W. Harris, "Multi-Core in the Source Engine," *bit-tech.net Tech. Pap.*, 2006.
- [10] Microsoft, "C# API Reference." [Online]. Available: <https://docs.microsoft.com/en-gb/dotnet/api/system.threading.thread?view=netframework-4.7.1>. [Accessed: 17-Jan-2018].
- [11] E. Brewer, "Clustering: Multiply and Conquer," *Data Commun.*, 1997.
- [12] T. O. P. MPI, "Open MPI API Reference," 2017. [Online]. Available: <https://www.open-mpi.org/doc/current/man1/mpirun.1.php>.
- [13] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2011.

Appendix:

Threading example from Microsoft C# API Reference:

```
1. using System;
2. using System.Threading;
3.
4. // Simple threading scenario: Start a static method running
5. // on a second thread.
6. public class ThreadExample {
7.     // The ThreadProc method is called when the thread starts.
8.     // It loops ten times, writing to the console and yielding
9.     // the rest of its time slice each time, and then ends.
10.    public static void ThreadProc() {
11.        for (int i = 0; i < 10; i++) {
12.            Console.WriteLine("ThreadProc: {0}", i);
13.            // Yield the rest of the time slice.
14.            Thread.Sleep(0);
15.        }
16.    }
17.
18.    public static void Main() {
19.        Console.WriteLine("Main thread: Start a second thread.");
20.        // The constructor for the Thread class requires a ThreadStart
21.        // delegate that represents the method to be executed on the
22.        // thread. C# simplifies the creation of this delegate.
23.        Thread t = new Thread(new ThreadStart(ThreadProc));
24.
25.        // Start ThreadProc. Note that on a uniprocessor, the new
26.        // thread does not get any processor time until the main thread
27.        // is preempted or yields. Uncomment the Thread.Sleep that
28.        // follows t.Start() to see the difference.
29.        t.Start();
30.        //Thread.Sleep(0);
31.
32.        for (int i = 0; i < 4; i++) {
33.            Console.WriteLine("Main thread: Do some work.");
34.            Thread.Sleep(0);
35.        }
36.
37.        Console.WriteLine("Main thread: Call Join(), to wait until ThreadProc ends.");
38.        t.Join();
39.        Console.WriteLine("Main thread: ThreadProc.Join has returned. Press Enter to end program.");
40.        Console.ReadLine();
41.    }
42. }
```

Jen Ada Distributed and Cluster Computing Assignment 1

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

55/100

PAGE 1

PAGE 2



Comment 1

Very minor, but take care to re-index your contents section before uploading.

PAGE 3



Awk.

Awkward:

The expression or construction is cumbersome or difficult to read. Consider rewriting.



Comment 2

While cited, this entire section is extremely close to the original source, and has to be paraphrased far better or quoted (though the former would be preferable).

PAGE 4



Comment 3

This taken directly from another source, and should be quoted.



Comment 4

Verbatim from source.



Comment 5

Several sentences verbatim from source.



Comment 6

This huge section is verbatim from the source. You must be considerably more careful with this in future!



Comment 7

Really like this description.



Comment 8

It'd be helpful to see "fast", "slow" etc. quantified.

PAGE 5

PAGE 6



Comment 9

Bar two words, straight from source.

PAGE 7



Comment 10

Bar a few words, all straight from their respective sources.



Comment 11

Direct from source.



Comment 12

You mention AMD, but not a hardware-compliant framework, e.g. OpenCL - would be nice to see an alternative to CUDA discussed.

PAGE 8

PAGE 9

PAGE 10

GRADING FORM: SMU GENERIC_GJ

JENNIFER ADA

55

RESEARCH-INFORMED LITERATURE



You've used a good number of academically credible sources, but there's a major, pervasive issue throughout this work. In several cases, you've taken material directly from source, and either inserted it directly, or changed one or two words. You must either quote this material and show that you've taken it verbatim, or ideally paraphrase sources far better in future.

4

KNOWLEDGE AND UNDERSTANDING



Some good knowledge and understanding evidenced. If anything, a little brief in places - as much as I'm not a fan of word counts, you're a good bit shy of the 2500 recommendation, and you're resultantly a little light on detail in some areas. Still, I have little doubt you have a robust understanding of the work.

13

ANALYSIS



Some reasonable evidence of analysis. Try to include conclusions throughout the report - this'll both summarise your work, and tie your sections together a little better.

12

PRACTICAL APPLICATION AND DEP.



Some good descriptions of practical implementations, using C# as an example. Some additional breadth would be good - e.g. OpenCL is conspicuously absent from your discussion of GPGPU techniques.

19

SKILLS FOR PROFESSIONAL PRAC.



You write well, with only a few minor errors in places. A nicely-presented document.

7