

P145484 Assignment 1

by Cerys Bethan Nickels

FILE	2645_CERYS_BETHAN_NICKELS_P145484_ASSIGNMENT_1_94244_1835039266.DOCX (281.46K)		
TIME SUBMITTED	22-JAN-2018 04:32AM (UTC+0000)	WORD COUNT	3668
SUBMISSION ID	82180255	CHARACTER COUNT	24267

Assignment Report

Name: Cerys Bethan Nickels

Student Number: P145484

Course: Software Engineering

Module Name: Distributed &
Cluster Computing (Level 6)

Module Code: M6X01093

Lecturer: Dr. Tim Bashford

Assignment Number: 1

Table of Contents

Table of Contents

1.0 Introduction to Parallelism.....	3
1.1 Historical Context.....	5
2.0 Parallel Computing Paradigms & Tools.....	7
2.1 Parallel Threading.....	7
2.2 Cluster Processing using Message Passing Interface.....	8
2.3 General Purpose Computation on Graphics Processing Units (GPGPU).....	9
2.3.1 CUDA & OpenCL.....	10
2.3.2 Analysis of GPU Data-Level Parallelism.....	10
3.0 Limitations of Parallel Computing.....	12
4.0 References.....	13

Table of Figures

Figure 1: Horizontal Parallelism [3, p. 24]	3
Figure 2: Vertical Parallelism [2, p. 24]	3
Figure 3: The Three Levels of Cache [8]	4
Figure 4: Role of the Bridge Within a Network [10]	5
Figure 5: The Deadlock Cycle [32]	12

Table of Equations

Equation 1: Amdahl's Law [34]	12
-------------------------------------	----

1.0 Introduction to Parallelism

According to El-Rewini [1], parallel processors are: 'computer systems consisting of multiple processing units connected via some interconnection network plus the software needed to make the processing units work together'. These consist of the processing units (such as the ALU) and the interconnection network (such as MPI) [1]. According to Fish [2, p. 24], there are two types of parallelism:

1. Horizontal parallelism increases the number of units working in parallel for throughput. This can be seen below:

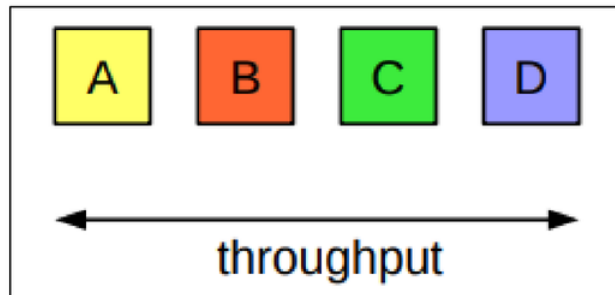


Figure 1: Horizontal Parallelism [3, p. 24]

2. Vertical parallelism for hiding latency. Pipelining keeps memory units busy when they are waiting for dependencies. This can be seen below:

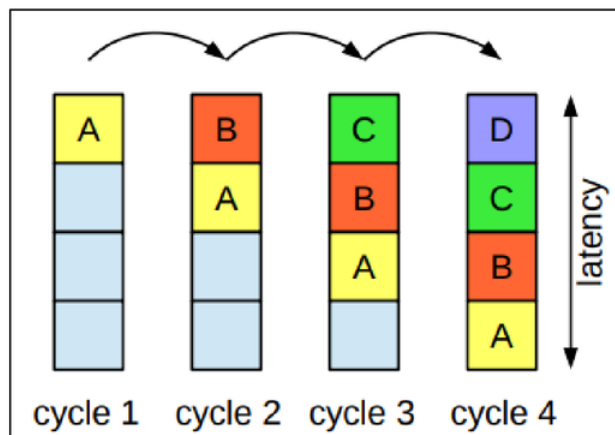


Figure 2: Vertical Parallelism [2, p. 24]

According to Collange [3, p. 22], the three sources of parallelism are:

- Instruction-Level Parallelism (ILP) – this occurs between independent instructions in sequential programs

- Thread-Level Parallelism (TLP) – this occurs between independent execution contexts or threads
- Data-Level Parallelism (DLP) this occurs between vector elements, with the same operation performed on several elements

Flynn's Taxonomy was published in 1966 [4], to categorise parallel computer architectures by the concurrency in processing sequences (or streams), data, and instructions. There are four classes:

- SISD (single instruction, single data)
- SIMD (single instruction, multiple data)
- MISD (multiple instruction, single data)
- MIMD (multiple instruction, multiple data) [1, Ch. 1.2], [5], [6, Ch. 3.2]

Cache is a specialised type of memory used to speed up the user's experience. This can be on the internet, known as browser cache, or for a program, where it is part of the code design, or part of the hardware, and is used to store recent data, or information gathered as part of a prediction of what might be requested, such as the next block of instructions [7, Ch. 1,2]. There are three levels:

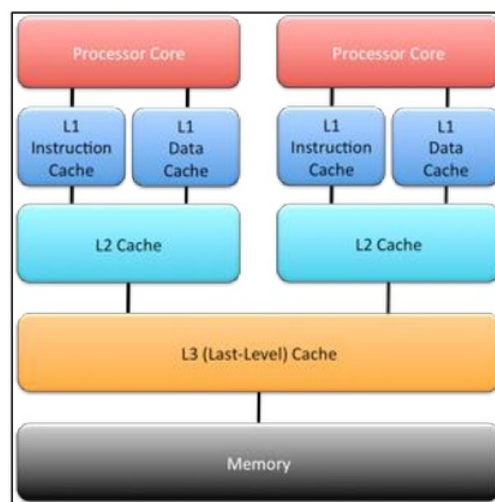


Figure 3: The Three Levels of Cache [8]

Bridges are used to allow communication through the joining of two separate computer networks, thus allowing them to work as a single network. They are used with Local Area Networks (LAN's), and inspect incoming network traffic, to decide whether to forward or discard it based upon its intended destination [9]. This can be seen below:

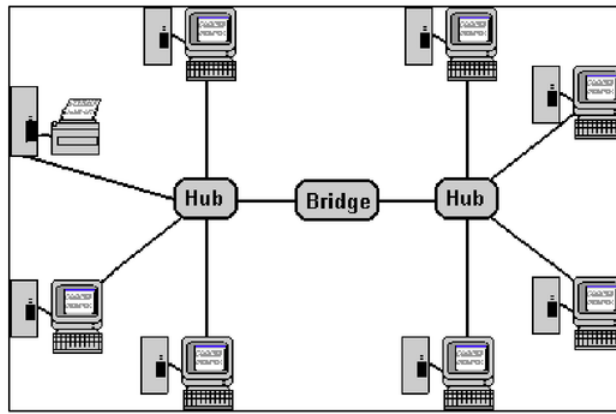


Figure 4: Role of the Bridge Within a Network [10]

1.1 Historical Context

According to [4], [11]–[13], some of the key dates are:

1842 – Charles Babbage invents the Analytical Engine, which incorporated an arithmetic logic unit, integrated memory, and controlled the flow using conditional branching and loops.

1955 – IBM launches the 704, with the principal architect being Gene Amdahl. It is the first commercial machine capable of performing floating-point arithmetic operations. It was capable of approximately 5 kFLOPS (floating-point operations per second).

1958 – The first large scale multiprocessor, the Illiac IV. In addition, the Gamma 60 is announced by Bull of France with multiple functional units. It is capable of fork & join operations.

1960's – The appearance of transistorised computers, which were capable of providing instruction-level parallelism at the machine-language level. Start of research into parallel computers.

1960 – Honeywell 800 is launched, with the capability of timesharing between eight programs.

1962 – The Atlas computer becomes operational, with the capability of approximately 200 kFLOPS. It is the first machine to use virtual memory and paging, pipelining the execution of instructions, with separate fixed and floating-point arithmetic units.

1964 – The CDC 6600 is launched as the first successful supercomputer, with one 60-bit CPU and 10 peripheral processing units (PPUs).

1968 – Adams coins the term “dataflow” whilst describing computational graphical models, and Dijkstra describes semaphores, introducing the dining philosopher’s problem.

1971 – Intel produced the world’s first single-chip CPU, the 4004 micro-processor.

1978 - The Communicating Sequential Processes (CSP) model is described by Hoare as the mixing of synchronous point-to-point communication with guarded commands. This becomes the basis for many parallel programming languages.



1982 – Wolfe released the optimizing compilers for supercomputers thesis, which contained a detailed, coherent account of program transformations for vectorization and shared-memory parallelisation.



Mid 1980's – Massively parallel processors (MPP) - The Caltech Concurrent Computation project was launched as a new kind of parallel computing. The supercomputer was built for scientific applications using 64 Intel 8086/8087 processors. This allowed for the development of cluster computing.

1993/4 – The MPI standard was introduced.

2.0 Parallel Computing Paradigms & Tools

There are many paradigms and tools used within parallel computing. These include the following:

2.1 Parallel Threading

A thread is a stream of executable code, which can be scheduled to perform a task. They are created by the kernel, managed by the OS, and are attached to processes, which have their main thread of execution. Multiple threads can access the same memory locations, therefore complex inter-process communications, such as TCP, semaphore, signals and MPI, become unnecessary [14]. Parallel threads are concurrent processes or threads that execute simultaneously. Additional threads can be branched off these as sub-threads, and can be dependent or independent of other parallel threads [15].

According to [15]–[17], the advantages of parallel threading are:

1. Improved performance and concurrency – These can be improved for certain applications through careful use of multithreading. These parallelised tasks allow for an improved use of system resources, allowing for simultaneous access to multiple applications through parallel execution of threads, speeding up the communication and processing of data.
2. Simplified coding of remote procedure calls and conversations – Potential to increase the ease of coding different remote procedure calls and conversations in separate threads, than managing them from the same thread. The creation and management of threads is easy, and can be managed by the OS or through its code design.
3. Reduced number of required servers – A server can create multiple service threads, thereby reducing the quantity of servers needed for an application. This is especially useful for increasing the availability of conversational servers, which would otherwise be dedicated to one client for the entire duration of the conversation.
4. Responsive – Especially in larger programs, there are multiple threads, each with their own task. If one thread stops working, then other threads within the program can keep the program running and responds to the user quickly. This is beneficial because threads within a program often share the same data, so no extra memory is required for each thread.
5. Resource utilization – Reducing CPU idle time due to multiple threads running in each program. This can help to improve cache usage by identifying what operation may come next, context switching by storing the current thread state for later execution, throughput and turnaround time.

According to [15]–[17], the disadvantages of parallel threading are:

1. Execution speed – The construction and synchronisation of threads is CPU/ memory intensive. Multi-threading is a heavy load on the processor, so a single, or even a dual-cored processor, it can overload the processor, causing programs to crash or blocking to occur.
2. Difficulty and complexity of writing code – Threading increases the complexity of these applications, making them harder to write. This requires programming experience to code these types of applications effectively. Bad design and programming of threads can lead to 'thread starvation', which occurs when a thread has not been allocated enough CPU time [18].
3. Debugging issues – There is a decrease in the ease of replicating, identifying, and verifying errors within the code source. This can be due to unpredictable results, which are often timing related, which are less common in single-threaded applications.
4. Managing concurrency – Managing thread concurrency is difficult, therefore it has the potential to introduce new problems into an application, such as deadlocks and synchronisation of shared resources, such as data and objects.
5. Porting existing code – Existing code often requires significant re-architecting to allow multithreading to exist. This includes the removing of static variables, and replacing any function calls, or other code, that is not thread-safe. These changes must be thoroughly tested, increasing the cost of the application.

2.2 Cluster Processing using Message Passing Interface

Message Passing Interface (MPI) is a cross-platform library which allows for communication (through the sending and receiving of messages) with distributed memory within parallel and cluster computing [14]. This standardised library specifies the names, calling sequences, and the results of subroutines from Fortran programs, and combines the functions from C programs, with the classes and methods from the MPI C++ library. Programs written using MPI can be compiled normally, and are linked with the MPI library. This is especially important to users which write and use parallel libraries, as these depend upon this library for communication [19, Ch. 2], [20].

According to Gropp et al [19, Ch. 2], the main features of MPI are:

- Application-orientated process structures
- Dynamically managed process groups
- Extensive set of collective operations
- Series of error codes to help determine MPI issues

There are over 430 functions used in MPI-3 [21], although the six main functions which are used are:

- MPI_Init – initialises MPI
- MPI_Comm_size – determines the number of processors
- MPI_Comm_rank – determines the type of process it is
- MPI_Send – sends a message
- MPI_Recv – receives a message
- MPI_Finalize – terminates MPI [19, Ch. 2]

According to [19, Ch. 1–2], [22], there are many benefits of MPI:

1. Can be implemented on a wide variety of hardware architectures
2. Reduced the need for MPI alternatives which use the message-passing model
3. Runs on shared and distributed memory architectures
4. No need to compromise on efficiency, portability and functionality
5. Portable as it takes advantage of the vendor provided hardware and software

However, there are several limitations to MPI:

1. Changes are required from serial to parallel versions of code
2. Can be harder to debug than other communication libraries
3. New versions are released, so need to ensure the user is up to date with the changes
4. Cluster computing and MPI require specialist hardware to implement
5. Performance is limited by the communication network between the nodes [14], [22]

2.3 General Purpose Computation on Graphics Processing Units (GPGPU)

Modern Graphics Processing Units (GPU's) consists of substantial numbers of Algorithmic Logic Units (ALU's) and threads. Using Flynn's taxonomy, GPU's are classified as single instruction multiple data (SIMD), as each thread has concurrent memory access, which allows the same operation to be performed on different parts of the data simultaneously. General Purpose computation on Graphics Processing Units (GPGPU) data is transferred using the system bus between the system and graphics card memory, as a method of distributing memory [14]. GPGPU can be achieved using existing programs, providing that the programmer can rewrite and restructure the code (using CUDA, OpenCL, or such tool or library), and can maintain both the CPU and GPU programs separately [23, p. 1]. This is a limitation as the GPU architecture is significantly different from the traditional CPU architecture [24]. This could be beneficial to use when there are

vector mathematics within the code, or when the program could benefit with threading. The main benefit of using these tools and libraries is the code is cleaner and easier to read than most graphics code. This makes the code more maintainable [25]. Another limitation is incorporating GPU hardware into systems adds expense in power consumption, heat production, and cost. Some of these tasks could be done more economically by maximising the number of CPU's that are available [24].

2.3.1 CUDA & OpenCL

Compute Unified Device Architecture (CUDA) was developed by NVIDIA in November 2006 to improve the productivity of GPU programmers. It is a C-like language and programming environment and improves heterogeneous computing and multifaceted parallelism [7, Ch. 4.4], [26]. According to Natoli [25]: "CUDA is an elegant solution to the problem of representing parallelism in algorithms, not all algorithms, but enough to matter". It allows for an easier and more natural method of task level parallelism with the way programmers think and code [25]. In CUDA 6.0, NVIDIA implemented algorithms which simulate shared memory, to create 'unified memory' [14]. Threads are created, managed, scheduled and executed in blocks of 32 threads, known as a warp, each with a consecutive thread ID [26], [27, p. 4].

OpenCL is developed by several companies to provide a vendor-independent language that can be used on multiple platforms [7, Ch. 4.4]. It is similar in how CUDA operates, and is an industry standard for task-parallel and data-parallel heterogeneous computing. It provides easy-to-use abstractions and a broad set of programming APIs by defining a set of core functions which can be used on all devices, with additional functions for higher-function devices. OpenCL can use a one-off compilation methodology, or can automatically take advantage of the latest hardware and software features of the device by using a run-time compilation methodology [28].

The thread is the lowest level of parallelism for these tools, with the compiler and hardware using thousands of these to operate different methods of parallelism within the GPU, such as multi-threading, MIMD, SIMD, and instruction-level parallelism [7, Ch. 4.4].

2.3.2 Analysis of GPU Data-Level Parallelism

There are many benefits of GPU data-level parallelism:

1. As GPU's are SIMT (Single Instruction, Multiple Threads), vectors are processed at a faster rate through the use of threads, than is possible with a single thread [7, Ch. 4].
2. As the GPU is separate from the CPU, the parallel thread of execution (PTX) exists, which allows greater flexibility for GPU architects to test new innovations and experiments. An

example of this is the Fermi architecture altered the hardware instruction set from being memory-oriented (such as x86) to being register-oriented (e.g. MIPS), and going from a 32-bit to a 64-bit address size, without changing the NVIDIA software stack [7, Ch. 4.8].

There are many limitations of GPU data-level parallelism:

1. As GPUs are coprocessors, they are separate from the main memory, and need to travel across the system bus. This increases system traffic and needs to be managed accordingly [7, Ch. 4.8].
2. Vector performance can be improved without accessing memory bandwidth, which is extremely important to all SIMD architectures [7, Ch. 4.8].
3. If the memory performance on GPUs is not sufficient, increase the thread count to minimise the latency to main memory. This is only possible if the memory accesses are not scattered and are correlated. There are limits to the number of manageable threads [7, Ch. 4.8].
4. Vector architectures concentrate on the peak clock speed performance and ignores the start-up overhead. In early processors (such as the TI ASC and the CDC STAR-100), the long start-up times meant that for some vector problems, long vectors were required for the vector code to be quicker than the scalar code. This is less of an issue now due to increased multi-threading, but it is still relevant [7, Ch. 4.8].
5. Increases in vector performance, without equivalent growth in scalar performance. This was an issue on many of the early vector processors, there they had comparatively slow scalar units, and high start-up overheads. This resulted in Seymour Cray (the Cray computer architect) rewriting the rules. Modern processors with low vector performance, but increased scalar performance, are capable of outperforming higher peak vector performance processors. Improved scalar performance reduces overhead costs (such as strip mining), latency and the impact of Amdahl's law [7, Ch. 4.8], [29].

3.0 Limitations of Parallel Computing

A race condition occurs when two or more threads access the same memory location, to change the shared data, and the final result is dependent on the thread order of execution [30]. The thread schedule algorithm is decided by the OS; therefore, the OS can swap between threads without the user's knowledge, and the user cannot control the order of how threads access the shared data. Race conditions can be detected during compile time, called static detection, and at during run-time, called dynamic detection [31].

Deadlocks occur when two or more threads (processes) are waiting for a resource, which is held by a mutex (mutual exclusion lock) [30]. Thus, preventing a thread from proceeding any further, causing the program to "hang" or stall. Debugging output and careful code design can be used to identify some of the sources of deadlocks within a program, which can change the timings and reduce the chance of this occurring. There are two types:

1. Global deadlocks occur if every thread in a program is blocked
2. Local deadlocks occur if one or more threads in a program are blocked [32]

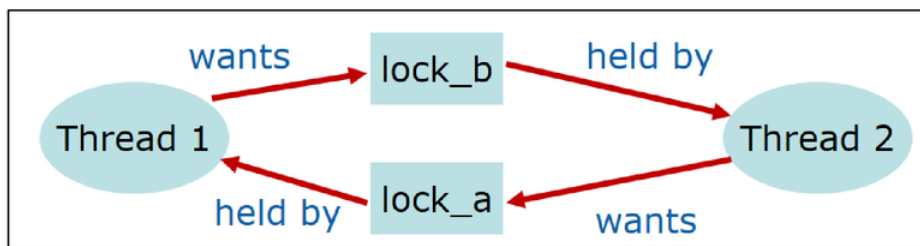


Figure 5: The Deadlock Cycle [32]

Amdahl's Law was published in 1967 [4], stating that there is an upper limit to the speed-up that can be achieved on a problem, independent of the number of processors [33]. It automatically eliminates many tasks that could be parallelised, such as if the code is incapable of being parallel, or if the serial portion is not much smaller than the code capable of being parallelised, it would not benefit from becoming parallelised, as it does not take into consideration the overhead it occurs. The equation below represents Amdahl's Law, and is usually expressed as an inequality, where $S(N)$ signifies the speed-up, $T(1)$ is the time, N is the number of processors, T_s is the serial portion and T_p is the parallel portion of the code [34], [35].

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p / N}$$

Equation 1: Amdahl's Law [34]

4.0 References

- [1] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Hoboken, New Jersey: John Wiley & Sons, Inc, 2005.
- [2] R. Fish, 'Future of computing - Part 3: The ILP Wall and pipelines', *EDN Network*, Feb-2012.
- [3] S. Collange, 'Parallel Programming: Introduction to GPU Architecture', 2015. [Online]. Available: http://www.irisa.fr/alf/downloads/collange/cours/ppar/ppar_gpu_1.pdf. [Accessed: 31-Dec-2017].
- [4] G. V Wilson, 'The History of the Development of Parallel Computing', 1994. [Online]. Available: https://parallel.ru/history/wilson_history.html. [Accessed: 21-Jan-2018].
- [5] M. Flynn, 'Flynn's Taxonomy', *Encyclopedia of Parallel Computing*. Springer US, Boston, MA, pp. 689–697, 2011.
- [6] B. Barney, 'Introduction to Parallel Computing', *Lawrence Livermore National Laboratory*, 2017. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/. [Accessed: 21-Jan-2018].
- [7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA: Elsevier, 2012.
- [8] Micro Kernel Dude, 'How to steal encryption keys: Your cloud is not as secure as you may think!', *Micro Kernel Dude*, 2015. [Online]. Available: <https://microkerneldude.wordpress.com/2015/04/27/how-to-steal-encryption-keys-your-cloud-is-not-as-secure-as-you-may-think/>. [Accessed: 21-Jan-2018].
- [9] B. Mitchell, 'Learn How to Use a Bridge to Expand Your Local Network', *Lifewire*, 2017. [Online]. Available: <https://www.lifewire.com/how-network-bridges-work-816357>. [Accessed: 21-Jan-2018].
- [10] Bucaro TechHelp, 'Computer Networking Devices', *Bucaro TechHelp*, 2001. [Online]. Available: <http://bucarotechhelp.com/networking/nwtech/87023001.asp>. [Accessed: 21-Jan-2018].
- [11] B. Ramakrishna Rau and J. A. Fisher, 'Instruction-Level Parallel Processing: History, Overview, and Perspective', *J. Supercomput.*, vol. 7, pp. 9–50, 1993.
- [12] J. L. Hennessy and D. A. Patterson, 'Historical Perspectives and References', in *Computer Architecture: A Quantitative Approach*, 5th ed., Waltham, MA: Elsevier, 2012, p. 84.

- [13] L. F. Menabrea and A. A. Lovelace, 'Sketch of The Analytical Engine', *Bibliothèque Univers. Genève*, no. 82, 1842.
- [14] T. Bashford, K. Donne, A. Marotin, and A. Al-Hussany, 'Parallelisation Techniques for the Dual Reciprocity and Time-Dependent Boundary Element Method Algorithms', *Int. J. Comput. Methods Exp. Meas.*, vol. 5, no. 3, pp. 395–403, 2017.
- [15] BEA Systems, 'Advantages and Disadvantages of a Multithreaded/Multicontexted Application', *BEA Systems*, 2000. [Online]. Available: https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm. [Accessed: 09-Jan-2018].
- [16] J. Rehman, 'Advantages and disadvantages of multi-threading systems - IT Release', *IT Release*, 2017. [Online]. Available: <http://www.itrelease.com/2017/12/advantages-disadvantages-multi-threading-systems/>. [Accessed: 20-Jan-2018].
- [17] J. Van Den Broecke, 'Multi-threading Advantages and Disadvantages', *Just Objects*. [Online]. Available: <http://justobjects.org/cowcatcher/browse/all/slides/java-essential/core/thread/slide.0.4.html>. [Accessed: 20-Jan-2018].
- [18] J. Jenkov, 'Java Concurrency / Multithreading Tutorial', *Jenkov.com*, 2014. [Online]. Available: <http://tutorials.jenkov.com/java-concurrency/index.html>. [Accessed: 20-Jan-2018].
- [19] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.
- [20] M. Samarah and R. Fatmi, 'Dynamic Self-assembling Petaflop Scale Clusters'.
- [21] B. Barney, 'Message Passing Interface (MPI)', *Lawrence Livermore National Laboratory*, 2017. [Online]. Available: <https://computing.llnl.gov/tutorials/mpi/>. [Accessed: 28-Dec-2017].
- [22] Dartmouth College, 'Pros and Cons of OpenMP and MPI', *Dartmouth College*, 2011. [Online]. Available: https://www.dartmouth.edu/~rc/classes/intro_mpi/parallel_prog_compare.html. [Accessed: 28-Dec-2017].
- [23] M. Wolfe, 'The PGI Accelerator Programming Model on NVIDIA GPUs Parts 1 and 2', Beaverton, Oregon, 2009.
- [24] Texas Advanced Computing Center, '8 Things You Should Know About GPGPU Technology', *Texas University*, 2011. [Online]. Available: <https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf>. [Accessed: 04-Jan-

2018].

- [25] V. Natoli, 'Kudos for CUDA', *HPC Wire*, Jul-2010. [Online]. Available: https://www.hpcwire.com/2010/07/06/kudos_for_cuda/. [Accessed: 28-Dec-2017].
- [26] J. Nickolls, I. Buck, and M. Garland, 'Scalable Parallel Programming', in *2008 IEEE Hot Chips 20 Symposium (HCS)*, 2008, no. March/ April, pp. 40–53.
- [27] G. Lei *et al.*, 'CPU-GPU hybrid accelerating the Zuker algorithm for RNA secondary structure prediction applications', in *Series on Advances in Bioinformatics and Computational Biology*, 2012, vol. 13, p. S14.
- [28] J. E. Stone, D. Gohara, and G. Shi, 'OpenCL: A parallel programming standard for heterogeneous computing systems', *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–72, May 2010.
- [29] Z. Chen and D. Kaeli, 'Balancing Scalar and Vector Execution on GPU Architectures', in *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, 2016, pp. 973–982.
- [30] F. Mckenna, 'Parallel Computing', *OpenSees Parallel Workshop*, 2010. [Online]. Available: <http://opensees.berkeley.edu/OpenSees/workshops/parallel/ParallelIntroduction.pdf>. [Accessed: 04-Jan-2018].
- [31] Y. Wang, 'Race Detection in Parallel Programming', San Jose, CA, 2015.
- [32] Intel Software, 'Deadlock Introduction to Parallel Programming', *Intel Software*, 2009. [Online]. Available: https://software.intel.com/sites/default/files/W07_Deadlock.pptx. [Accessed: 04-Jan-2018].
- [33] H. S. Morse, *Practical Parallel Computing*. California, USA: AP Professional, 1994.
- [34] R. G. Brown, 'Amdahl's Law & Parallel Speedup', *Duke University*, 2000. [Online]. Available: <http://webhome.phy.duke.edu/~rgb/brama/Resources/als/als/node3.html>. [Accessed: 04-Jan-2018].
- [35] C. Junghans, A. Agarwal, and L. Delle Site, 'Computational efficiency and Amdahl's law for the adaptive resolution simulation technique', *Comput. Phys. Commun.*, vol. 215, pp. 20–25, Jun. 2017.

P145484 Assignment 1

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

68/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5



Comment 1

No apostrophe - this is a plural, not a possessive.

PAGE 6

PAGE 7



Comment 2

This is too close to the source, and should either be better-paraphrased, or quoted.



Comment 3

Also direct from source.

PAGE 8

PAGE 9

PAGE 10

PAGE 11



Comment 4

Verbatim from source.

PAGE 12

**Comment 5**

Verbatim from source.

GRADING FORM: SMU GENERIC_GJ

CERY'S BETHAN NICKELS**68****RESEARCH-INFORMED LITERATURE**

An excellent number of sources used, though a few (from the web) of poor academic credibility. Good application of inline citations. This section, unfortunately, is slightly let down by a number of sections of the text which were taken directly from the source material, but were not quoted. You need to work on your paraphrasing to make sure this doesn't occur again, or quote this material if you feel you cannot paraphrase it.

6**KNOWLEDGE AND UNDERSTANDING**

Good evidence of knowledge and understanding. A well-researched, in-depth report.

20**ANALYSIS**



Some good synthesis of academic material. A lack of conclusions through the text, however - these would help your work to flow together a little better. Something to think about for the next assignment!

14

PRACTICAL APPLICATION AND DEP.



Good descriptions of underpinning technologies. An excellent amount of technical detail in many areas.

22

SKILLS FOR PROFESSIONAL PRAC.



Reasonably good spelling, punctuation and grammar. A few minor errors - be careful with apostrophes in particular. A nicely-presented report.

6