

Assignment 2 Jen Ada

by Jennifer Ada

FILE	3017_JENNIFER_ADA_ASSIGNMENT_2_JEN_ADA_113617_1735634582.D OCX		
TIME SUBMITTED	04-MAY-2018 11:04PM (UTC+0100)	WORD COUNT	2116
SUBMISSION ID	88427332	CHARACTER COUNT	14441

Contents

Pertinent Literature	2
Moore's Law	2
Amdahl's Law	2
Gustafson-Barsis's Law	2
Sun & Ni's Law	2
C++ Threads	2
Message Passing Interface	2
CUDA	2
Diagnosis of problem	3
Profiling tool	3
1 st Experiment: Rough granularity, precise time measurements.	4
2 nd Experiment: More granularity, less precise time measurements.	5
Threading (C++/Boost).....	7
Message Passing Interface	Error! Bookmark not defined.
CUDA	Error! Bookmark not defined.
Conclusion	10
References	11

Pertinent Literature.

There are various laws which impose limits on the amount of the code that can be parallelised and how effective that parallelisation can be. The following laws are most relevant to parallelisation.

Moore's Law

As a consequence of transistor density increasing annually, processors now contain multiple "cores" [1]. Depending on how the cores are utilised, their use can be more efficient or less efficient than a single core processor. If all cores are not made use of concurrently then resources are wasted [2]. In regards to the other type of processor, general purpose graphics processing units (GPGPU), this is less of an issue according to Flynn's taxonomy[3] as a GPU generally will perform a single operation on a batch of data (data parallelism), therefore ideally maximizing the concurrent use of its resources.

Amdahl's Law

The proportion of the code that can be parallelised has a limit. Some code will only be able to run serially. This introduces a constraint on the optimisation of a serial program through parallelisation [4].

Gustafson-Barsis's Law

Whereas Amdahl's Law views time as a variable, in GB time taken is constant and it is the amount that parallelisation can scale in that time that is significant [5]. As Moore's law continues to provide more resources more work will be done in the same amount of time that less work could be done previously.

Sun & Ni's Law

Sun and Ni are relevant to parallelisation because of parallelisation's problems with inter-process (and associated constructs) communication [6]. Memory capacity is important because the more data that the cache can hold the less chance there is of a page faults where the program tries to access data that has not been loaded to memory[7]. Bottlenecks between processor and memory increase the risk of page faults further.

There are also various techniques, frameworks, libraries and API's which can be used to parallelise code. The following will be evaluated in terms of how appropriate they are to optimise the code provided.

C++ Threads

Threads in C++ 11[8] can be invoked using a lambda expression to create a closure. This stores the arguments of the thread in square brackets. Using the equality operator "[=]" will capture the arguments by value for the thread but other operators can pass values by reference or a combination of the two. The main program of a C++ program is inherently a thread attached to a process and data which is written or read by multiple threads can be protected with techniques such as the mutex which protects critical sections from race conditions. Threading is useful because threads can access and manipulate data in the same allocation of memory mitigating the need for inter-process communication such as in MPI[7].

Message Passing Interface

Cluster computing can be executed by using the high-performance computing (HPC) message passing interface [9]. There are two useful methods for sharing data which are more complicated to implement than threading because each node in a cluster is networked and usually physically separated unlike in threading where activity takes place on one machine. Consequently data must be shared between nodes either using "intra-node parallelism" (threading), "inter-node parallelism" (MPI) or both [7].

CUDA

CUDA leverages the abundant ALU's (arithmetic logic units) on a GPGPU to perform single instruction multiple data processing as opposed to threading which performs MIMD (multiple instructions). This allows the CUDA to perform operations on large vectors and matrices concurrently. There is a shared memory element which is cached data and units can access memory for read / write operations[7][10].

Diagnosis of problem.

1st Experiment: Initial run.

An initial run of the program indicates that "C and D" comprise approximately 50% of execution time (See figure 1).

```
Enter file name to read:
input_200.txt
Time taken to complete "File Read": 0 milliseconds.
Time taken to complete "Initialising variables": 0 milliseconds.
Time taken to complete "Setting A": 0 milliseconds.
Time taken to complete "Setting B": 0 milliseconds.
Time taken to complete "Inverting A": 14 milliseconds.
Time taken to complete "Inverting B": 14 milliseconds.
Time taken to complete "Setting ADash and BDash": 37 milliseconds.
Time taken to complete "Setting C and D": 146 milliseconds.
Time taken to complete "Setting E": 47 milliseconds.
Time taken to complete "Setting Result": 0 milliseconds.
Time taken to complete "Computation": 253 milliseconds.
Time taken to complete "File write": 8 milliseconds.
Time taken to complete "Deallocating variables": 0 milliseconds.
Time taken to complete "Entire Program": 277 milliseconds.
Complete. Press <Enter> to exit.
```

Figure 1 Initial run of code demonstrating majority of execution time is dedicated to "C and D", and "Computation".

2nd Experiment: Rough granularity, precise time measurements.

Using the profiler built into Microsoft Visual Studio 2017[11], the CPU usage of each function was measured to confirm the initial findings about “CD” necessitating further investigation. “Computation” was not ranked high for CPU usage and was omitted from investigation at this stage (See Table 1).

Table 1 Recorded CPU time for most expensive functions in experiment 2.

Function Name	Total CPU [ms, %]	Self CPU [ms, %]
ComputationEngine::SetCD	209 (55.00%)	207 (54.47%)
ComputationEngine::SetABDash	44 (11.58%)	0 (0.00%)
MatrixInverse::MatrixInversion	37 (9.74%)	37 (9.74%)
ScopeTimer::~ScopedTimer	20 (5.26%)	0 (0.00%)

3rd Experiment: More granularity, less precise time measurements.

The performance wizard was used to examine the code in more depth but at the sacrifice of precision in terms of time measurements. This is because it uses a technique called sampling[12]. Exclusive time is only that time spent explicitly in the initial scope of a function. Inclusive includes also the time spent in functions called by the function. Both are measurements of which function is at the top of the call stack at distributed points in time as opposed to measuring the actual time taken.

The “Hot Path” reading confirms that ComputationEngine::SetCD is the most resource intensive part of the program (See Table 2).

Table 2 Hot Path sampling confirming "CD" is most computationally expensive.

Hot Path		
Function Name	Inclusive Samples %	Exclusive Samples %
DCCAssignment.exe	100.00	0.00
sert common main seh	100.00	0.00
Main	100.00	22.13
ComputationEngine::SetCD	59.10	58.82

Again, when examining which functions do the most work using exclusive samples, ComputationEngine::SetCD comprises the majority of samples (See Table 3).

Table 3 Exclusive samples of functions doing the most work individually.

Functions Doing Most Individual Work	
Name	Exclusive Samples %
ComputationEngine::SetCD	58.82
Main	22.13
MatrixInverse::MatrixInversion	9.52
[msvcpl40.dll]	7.28
[ucrtbase.dll]	1.12

When examining ComputationEngine::SetCD itself in more detail (See Table 4), four lines are identified as comprising the most resource intensive parts and these will be the focus of optimisation as hypothetically, these will return the most gains for optimisation.

Table 4 Largest individual samples amongst lines in "CD".

Function Name	Exclusive Samples	Exclusive Samples %	Inclusive Samples	Inclusive Samples %	Source Line Begin
ComputationEngine::SetCD	131	36.69	131	36.69	172
ComputationEngine::SetCD	40	11.2	40	11.2	175
ComputationEngine::SetCD	34	9.52	34	9.52	174
ComputationEngine::SetCD	4	1.10%	4	1.10%	173

Here those high priority lines have been flagged with task markup in the following code. This is possibly $O(n^3)$ in terms of time complexity as this would mean that the complexity is the square of the input as there is three nested loops:

```

1. void ComputationEngine::SetCD(void)
2. {
3.     ScopedTimer *st = new ScopedTimer("Setting C and D");
4.     for (int L = 0; L < arrSize; L++)
5.     {
6.
7.         for (size_t I = 0; I<arrSize; I++)
8.         {
9.             size_t b = L*arrSize + I;
10.            nC[b] = 0.0;
11.            nD[b] = 0.0;
12.
13.            for (size_t J = 0; J<arrSize; J++)
14.            {
15.                size_t k1 = L*arrSize + J; //TODO: Analyze Priority: High
16.                size_t k2 = I + J*arrSize;
17.                nC[b] += nA[k1] * nB[k2]; //TODO: Analyze Priority: Medium
18.                nD[b] += nADash[k1] * nBDash[k2]; //TODO: Analyze Priority: Medium
19.            }
20.        }
21.    }
22.    delete(st);
23. }

```

Here are the individual lines most relevant and prioritised by severity.

```

172: size_t k1 = L*arrSize + J; //TODO: Analyze Priority: High
174: nC[b] += nA[k1] * nB[k2]; //TODO: Analyze Priority: Medium
175: nD[b] += nADash[k1] * nBDash[k2]; //TODO: Analyze Priority: Medium

```

The problematic code will be optimised by separating the method so that it loops through C's operations and D's operations separately adhering to the KISS principle of ensuring that a solution is not more complicated than the problem[13] illustrated by the printout below which shows approximately 30% faster performance for C and D functions (See Fig. 2).

```

Enter file name to read:
input_200.txt
Time taken to complete "File Read": 0 milliseconds.
Time taken to complete "Initialising variables": 0 milliseconds.
Time taken to complete "Setting A": 0 milliseconds.
Time taken to complete "Setting B": 0 milliseconds.
Time taken to complete "Inverting A": 11 milliseconds.
Time taken to complete "Inverting B": 13 milliseconds.
Time taken to complete "Setting ADash and BDash": 30 milliseconds.
Time taken to complete "Setting C": 53 milliseconds.
Time taken to complete "Setting D": 54 milliseconds.
Time taken to complete "Setting E": 53 milliseconds.
Time taken to complete "Setting Result": 0 milliseconds.
Time taken to complete "Computation": 223 milliseconds.
Time taken to complete "File write": 13 milliseconds.
Time taken to complete "Deallocating variables": 0 milliseconds.
Time taken to complete "Entire Program": 253 milliseconds.
Complete. Press <Enter> to exit.

```

Figure 2 Updated output for separate C and D functions.

Threading in C++.

Threading was implemented through iterative prototyping. The quantity of processors available is measured and the loop divided into subsections based on the measurement. These subsections should not introduce race conditions as each iteration of the loop addresses a different index in the target vector.

```

1. void ComputationEngine::ThreadedMethod(int start_number, int end_number)
2. {
3.     for (int loop_1 = start_number; loop_1 < arrSize + end_number; loop_1++)
4.     {
5.         for (size_t loop_2 = 0; loop_2 < arrSize ; loop_2++)
6.         {
7.             size_t b = loop_2 + (loop_1 * arrSize);
8.             nD[b] = 0.0;
9.             for (size_t loop_3 = 0; loop_3 < arrSize; loop_3++)
10.            {
11.                size_t k1 = loop_3 + (loop_1 * arrSize); //TODO: Analyze Priority: High
12.                size_t k2 = loop_2 + (loop_3 * arrSize);
13.                nD[b] += nADash[k1] * nBDash[k2]; //TODO: Analyze Priority: Medium
14.            }
15.        }
16.    }
17. }
18. void ComputationEngine::SetDPrototype(void)
19. {
20.     int processor_count = std::thread::hardware_concurrency();
21.     int subloop_size = arrSize / processor_count;
22.     int start_number = 0;
23.     int end_number = 0;
24.     ScopedTimer *st = new ScopedTimer("Setting D");
25.     for(int i = 0; i < processor_count; i++)
26.     {
27.         cout << "Setting thread " << i << endl;
28.         start_number = i * subloop_size;
29.         end_number = subloop_size + (i * subloop_size);
30.         thread *t = new thread([=] { ThreadedMethod( start_number, end_number); });
31.     }
32.     delete(st);
33. }

```

Taking measurements like those in Experiment 2's (Fig. 3) and 3's (Fig. 4), C and D disappeared from the most CPU intensive functions when the code was profiled).

Function Name	Total CPU [ms, %]	Self CPU [ms, %]
▲ DCCAssignmentExe (PID: 848)	327 (100.00%)	327 (100.00%)
main	312 (95.41%)	244 (74.62%)
__scrt_common_main_seh	312 (95.41%)	0 (0.00%)
ComputationEngine::SetABDash	42 (12.84%)	0 (0.00%)
MatrixInverse::MatrixInversion	37 (11.31%)	37 (11.31%)
ScopedTimer::~ScopedTimer	12 (3.67%)	0 (0.00%)
std::operator<< <std::char_traits<char> >	10 (3.06%)	0 (0.00%)
FileIO::WriteFile	5 (1.53%)	0 (0.00%)

Figure 3 Time measurements seem to imply that C and D are more efficient due to their absence.

Function Name	Inclusive Samples	Exclusive Samples ▼	Inclusive Samples %	Exclusive Samples %
main	294	234	98.99%	78.79%
MatrixInverse::MatrixInversion	36	36	12.12%	12.12%
[msvcrt140.dll]	20	17	6.73%	5.72%
[ucrtbase.dll]	7	7	2.36%	2.36%
ComputationEngine::SetA	1	1	0.34%	0.34%
ComputationEngine::SetB	1	1	0.34%	0.34%
std::basic_filebuf<char,std::char...	3	1	1.01%	0.34%

Figure 4 Stack sample measurements seem to also imply that C and D are more efficient due to their absence

On executing the optimised code, the program crashes after finishing computation. Multithreading C appears to be more inefficient than simply separating the code (See Fig. 5) this could not be verified with the profiler as the functions were not computationally expensive enough to be logged with the profiler.

```
Enter file name to read:
input_200.txt
Time taken to complete "File Read": 0 milliseconds.
Time taken to complete "Initialising variables": 0 milliseconds.
Time taken to complete "Setting A": 0 milliseconds.
Time taken to complete "Setting B": 0 milliseconds.
Time taken to complete "Inverting A": 18 milliseconds.
Time taken to complete "Inverting B": 19 milliseconds.
Time taken to complete "Setting ADash and BDash": 47 milliseconds.
Time taken to complete "Setting C": 77 milliseconds.
Setting thread 0
Setting thread 1
Setting thread 2
Setting thread 3
Time taken to complete "Setting D": 35 milliseconds.
Time taken to complete "Setting E": 82 milliseconds.
Time taken to complete "Setting Result": 0 milliseconds.
Time taken to complete "Computation": 365 milliseconds.
```

Figure 5 The extent to which the code will execute.

Conclusion

The best technique for optimising the code of this nature according to Bashford *et al.* [7] is threading due to the “linear increase in optimisation with every core that was utilised”, that they gained in their experiments. It was found that ‘dgetrf’ (which is assumed to be a factorisation operation on a matrix) was inefficient on both CUDA and MPI. Following this logic, threading was attempted first and MPI / CUDA was deprecated in priority. In theory splitting the outermost loop of the triple nested for loops should deliver an increase in performance but this could not be verified due to access violations occurring at the end of execution and the profiler being incapable of triangulating the measurements taken by the program itself. MPI and CUDA were not investigated in as much depth due to personal limitations in terms of scope prioritisation and time resource management which must be rectified in future endeavours.

References.

- [1] G. E. Moore, "Cramming more components onto integrated circuits.," *Electronics*, vol. 38, no. 8, 1965.
- [2] C. Martin, H. Augsburg, and F. ■ Rechnerarchitektur, "Post-Dennard Scaling and the final Years of Moore's Law Consequences for the Evolution of Multicore-Architectures Informatik und Interaktive Systeme Post-Dennard Scaling and the final Years of Moore's Law," 2014.
- [3] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.
- [4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, 1967.
- [5] J. L. Gustafson and J. L., "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.
- [6] X. H. Sun and Y. Chen, "Reevaluating Amdahl's law in the multicore era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, pp. 183–188, 2010.
- [7] 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.
- [8] B. Stroustrup, "Lecture: C++11 The Future is here." Swansea, 2013.
- [9] T. O. P. MPI, "Open MPI API Reference," 2017. [Online]. Available: <https://www.open-mpi.org/doc/current/man1/mpiun.1.php>. [Accessed: 01-Jan-2018].
- [10] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 2011.
- [11] Microsoft, "Beginners Guide to Performance Profiling." [Online]. Available: <https://msdn.microsoft.com/en-us/library/ms182372.aspx>. [Accessed: 04-May-2018].
- [12] Microsoft, "What are Exclusive and Inclusive?! – Visual Studio Profiler Team Blog," *Visual Studio Profiler Team Blog*, 2004. [Online]. Available: <https://blogs.msdn.microsoft.com/profiler/2004/06/09/what-are-exclusive-and-inclusive/>. [Accessed: 04-May-2018].
- [13] D. Periwal, "The KISS (Keep It Simple and Straightforward) Principles for OR-Mapping Products from Software Tree, Inc A White Paper."

Assignment 2 Jen Ada

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

52/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

GRADING FORM: SMU GENERIC_GJ

JENNIFER ADA

52

RESEARCH-INFORMED LITERATURE



An excellent number of sources correctly cited.
Unfortunately, I can't give marks for flattery, but I

4.5

can enjoy it all the same. :-)

KNOWLEDGE AND UNDERSTANDING



Very good evidence of knowledge and understanding - a well-written and thoroughly-considered report. A very logical process applied. Well done!

23

ANALYSIS



Good analysis - some results generated, though it'd be nice to see some concluding numbers and graphs. Suspect this was rather limited by the implementation.

8

PRACTICAL APPLICATION AND DEP.



Threading applied and some optimisation performed, though only for the one method - you've investigated the primary bottleneck.

12

SKILLS FOR PROFESSIONAL PRAC.



A well-written report, nicely presented.

4.5