

Report

by Eyad Ahmed N. Algmesh

FILE	2934_EYAD_AHMED_N._ALGMESH_REPORT_113617_266019742.DOCX		
TIME SUBMITTED	24-MAY-2018 12:24PM (UTC+0100)	WORD COUNT	2117
SUBMISSION ID	89843564	CHARACTER COUNT	12554



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

M6X01093
DISTRIBUTED AND CLUSTER COMPUTING

DR.TIM BASHFORD

By: Eyad Ahmed N. Algmesh-P142639
23-05-18



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

RESULT

School of Applied Computing ASSIGNMENT SPECIFICATION

Student name:	Eyad Ahmed N. Algmesh	Student P number:	P142639
Programme:	Software Engineering		
Lecturer:	Tim Bashford	Internal Verifier:	
Assignment Title:	Multi-mode parallelism of a scientific code		
Hand Out Date:	19/02/2018	Submission deadline:	04/05/2018

Referencing:	In the main body of your submission you must give credit to authors on whose research your work is based. Append to your submission a reference list that indicates the books, articles, etc. that you have read or quoted in order to complete this assignment (e.g. for books: surname of author and initials, year of publication, <u>title of book</u> , edition, publisher: place of publication).		
Disclosure:	I declare that this assignment is all my own work and that I have acknowledged all materials used from the published or unpublished works of other people. All references have been duly cited.		
Student's Signature:	<i>Hardcopy assignments will not be accepted without a signature here</i> Eyad	Date:	23-05-18

Late submission will result in a late penalty mark, as follows:

Up to one week late, maximum mark of 40% for first attempts. No late submission for resit work.

(Academic Quality Handbook 2016/17 7.5 (5))

Only the Extenuating Circumstances Committee may grant an extension.

Learning Outcomes tested (from module syllabus)

Critically appraise the major concepts, principles and practice to design distributed software components for use in enterprise services.

Demonstrate a comprehensive knowledge and understanding of complex client/server concepts, principles and tools to build, test, debug and critically evaluate software components in scalable, distributed applications.

Table of Contents

Introduction	4
Terminology	5
Potential Issues in Threading	6
Advancements	6
User Interface	7
Ideal approach	7
Exception Handling	8
REFERENCES	9

Introduction

The best choice for the code to enhance performance is Threading, which improves the throughput of the code and it is discussed in detail throughout the document. A message passing interface (MPI) is used to make the code run in parallel on two nodes. The updated Main.cpp file contains the parallel code. The improvement of performance of a code depends upon how we are defining the word performance. Here we are defining the performance by improving the time the code will take to show results, when a huge amount of work is given for processing. We can call this as throughput and we do threading in our coding to improve its throughput. For the given code, multithreading is used to improve the overall throughput performance. Multithreading allows more than one computer to do work on a shared problem. For the concept of multithreading to work properly, two things can help, and this includes the programming experience in C++. The higher the experience the probability of being comfortable with Win32 strings or threads increases. Apart from the graphical user interface (GUI) programming, the main issue in writing code in C++ is to do the threading in a proper way.

Threading in C++ implies diverse functions and procedures are executed simultaneously on various centers or processors. It is a specific type of multitasking. Multitasking permits to run at least two projects simultaneously. Multithreading makes a procedure that comprises different strings of execution(thread). It is utilized when the parallel execution of a few functions prompts more effective utilization of the resources that is part of the framework. Threading can be connected to one procedure to empower parallel execution on a multiprocessing framework [1].

The concepts of threading are a generally lightweight approach to implement numerous ways of execution within an application. At the framework level, programs run one next to the other, with the framework doling out execution time to each program in light of its needs and the necessities of different projects. Inside each program, there exists at least one thread of execution, which can be utilized to perform distinctive functions at the same time or in an about concurrent way. The framework itself really deals with these threads of execution, booking them to keep running on the accessible centers and similarly intruding on them as expected to enable different threads to run. From a programming point of view, a single thread is a mixture of the internal kernel level and application-level information structures expected to deal with

the execution of code. The kernel structures arrange the dispatching of programmer parts to the thread and the preemptive booking of the thread on one of the accessible centers [2].

The following code structure will be used to make the loops run in parallel.

```
struct __omp_func1_data
{
    int start;
    int end;
    int *b;
    int *c;
    int a;
};

void *__omp_func1(void *data)
{
    struct __omp_func1_data *d = data;
    int i;

    d->a = 0;
    for (i = d->start; i < d->end; ++i)
        d->a += d->b[i] + d->c[i];

    return NULL;
}

...
struct master_data md = {
    .start = 0,
    .end = 100
    .b = b,
    .c = c
};

__omp_func1(&md);
a += md.a;
```

Terminology

In an application which is based on non-concurrency, there is just a single thread of execution. That thread begins and shutdown with your application's primary routine and branches one-by-one to various techniques or capacities to execute the application's general behavior. An application that backs concurrency-based system, begins with only one thread and adds more as expected to make extra execution ways. Each new way has its own particular custom begin schedule that runs autonomously of the code in the application's fundamental schedule. Having various strings in an application gives two critical potential points of interest:

- 1- Various threads can enhance an application's responsiveness behavior.
- 2- Numerous threads can enhance an application's constant execution on multicore frameworks.

There can be a case that your application has just a single thread, and that thread is responsible for doing everything. It must react to all the events in the program, refresh your application's windows, and play out the greater part of the calculations expected to execute your application's conduct. The issue with having only one thread is that it can just complete one thing at once. So, what happens when one of your calculations takes more than usual time to wrap up? While your code is calculating all of the values it can need, your application quits reacting to client events and refreshing its windows. There is a chance that this conduct proceeds sufficiently long, a client may think your application is hung and ask you to stop it forcefully. In the event that you moved your custom calculations onto a different thread, nonetheless, your application's fundamental thread would be allowed to react to client communications in a quicker way [3].

Potential Issues in Threading

Obviously, threads are not a cure for settling an application's execution issues. Alongside the advantages offered by threads come the potential issues. Having numerous ways of execution in an application can include a lot of unpredictability to your code. Each string needs to arrange its activities with different strings to keep it from defiling the application's state data. Since strings in a solitary application share a similar memory space, they approach the greater part of similar information structures. There is a chance that two strings try to control similar information simultaneously, one string may overwrite another's which may result in the destruction of the whole database. Indeed, even with appropriate securities set up, despite everything you need to keep an eye out for compiler improvements that present un wanted bugs into your code [4].

Advancements

If you have a code that is using the concept of threads, OS X and iOS give a few advances to making threads in your applications. Likewise, the two frameworks additionally offer help for overseeing and synchronizing the work that should be done on those strings. At the application level, all of the threads show their behavior as they do on other platforms. When to start a string, the string keeps running in one of three principle states: running, prepared, or blocked. In the event that a string isn't as of now running, it is either blocked or in the queue for the input waiting for its turn. The thread keeps moving forward and backward among these states until the point that it at long last exits and moves to the ended state. When you make a new thread in your programme, you should determine an entry section (or a section point strategy

on account of Cocoa threads) for that thread. This passage point work establishes the code you need to keep running on the thread. At the point when the main function returns, or when you end the string expressly, the string stops for all time and is recovered by the framework. Since threads are moderately costly to make as far as memory and time is the concern, that's why it is recommended that your entrance section for input complete a lot of work [5].

User Interface

In the event that your application has a graphical UI, it is suggested that you get client related occasions and start interface refreshes from your application's primary thread. This approach maintains a strategic distance from synchronization issues related to taking care of client occasions and drawing window content. A few systems, for example, Cocoa, by and large, require this conduct, however for those that don't, keeping this conduct on the primary string has the benefit of rearranging the rationale for dealing with your UI. There are a couple of prominent exemptions where it is beneficial to perform graphical tasks from different threads. For instance, you can utilize optional threads to make and process pictures and perform other picture related counts. Utilizing optional strings for these tasks can significantly expand execution. On the off chance that you don't know about a specific graphical task, however, anticipate doing it from your primary string [6].

Ideal approach

The ideal approach for a thread to exit in a code is to exist naturally, by giving it a chance to achieve the finish of its primary passage point schedule. In spite of the fact that there are behaviors to end strings quickly, those behaviors ought to be utilized just if all else fails. Ending a thread before it has achieved its characteristic endpoint keeps the thread from tidying up after itself. However, if the string has allotted memory, opened a document, or gained different kinds of assets, your code might be not able to recover those assets, bringing about memory releases or other potential issues.

Despite the fact that an application designer has control about whether an application executes with numerous threads, library designers don't. When creating libraries, you should expect that the calling application is multithreaded or could change to being multithreaded. Therefore, you ought to dependably utilize locks for basic areas of code. For library designers, it is inappropriate to make locks just when an application moves toward becoming multithreaded. If you have to bolt your code eventually, make the lock ahead of schedule in the utilization of your library, ideally in some kind of definite call to declare the library. In spite of the fact that

you could likewise utilize a static library introduction declaration to make such lock attempt to do as such just when there is no other way [6].

CUDA based compilation

For CUDA based parallel implementation to work, a proper programming environment needs to be set up. The CUDA environment is used in the VISUAL studio, and the following header and parameter values are used for successful compilation of code.

```
1: #pragma once
2: #pragma warning(push)
3: #pragma warning(disable: 4996)           // Thrust's use of strerror
4: #pragma warning(disable: 4251)         // STL class exports
5: #include "thrust/host_vector.h"
6: #include "thrust/device_vector.h"
7: #pragma warning(pop)
```

Exception Handling

Neglecting to get a special case in an optional string is the same as neglecting to get an exemption in your primary string. In the case that you have to inform another thread, of an uncommon circumstance in the present thread, you should get the exemption and essentially make an impression on the other thread showing what happened [7]. In conclusion, we have used multithreading to increase the throughput of the code, which introduces parallel structure in the program. The best compilation time for the modified code is achieved using MPI code and a thread-based parallelization of loops. A CUDA based implementation can get better results if run on a graphical processing unit, where the capability of the GPU will define the final throughput optimization.

REFERENCES

- 1- Holowczak.com Tutorials. (2018). Introduction to Multithreading in C++. [online] Available at: <http://holowczak.com/introduction-to-multithreading-in-cplusplus/> [Accessed 9 May 2018].
- 2- J. A. Joines and S. D. Roberts, "An Introduction To Object-oriented Simulation In C++," *Winter Simulation Conference Proceedings*,, Atlanta, Georgia, USA, 1997, pp. 78-85.
- 3- C. Sanderson and R. Curtin, "An open source C++ implementation of multi-threaded Gaussian mixture models, k-means and expectation maximisation," *2017 11th International Conference on Signal Processing and Communication Systems (ICSPCS)*, Surfers Paradise, QLD, 2017, pp. 1-8.
- 4- Austin G. Walters. (2018). Multithreading: Common Pitfalls - Austin G. Walters. [online] Available at: <https://austingwalters.com/multithreading-common-pitfalls/> [Accessed 9 May 2018].
- 5- D. Hutchins, A. Ballman and D. Sutherland, "C/C++ Thread Safety Analysis," 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, 2014, pp. 41-46
- 6- Codeguru.com. (2018). UI-Thread Demonstration. [online] Available at: <https://www.codeguru.com/cpp/misc/misc/threadsprocesses/article.php/c3761/UIThread-Demonstration.htm> [Accessed 9 May 2018].
- 7- www.tutorialspoint.com. (2018). C++ Exception Handling. [online] Available at: https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm [Accessed 9 May 2018].

Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

31 /100

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

GRADING FORM: SMU GENERIC_GJ

EYAD AHMED N. ALGMESH

31

RESEARCH-INFORMED LITERATURE



A modest number of sources, though limited academic credibility in several cases.

4

KNOWLEDGE AND UNDERSTANDING

	Reasonable evidence of knowledge and understanding through the report.	10
---	--	-----------

ANALYSIS

	Some limited analysis - it'd be nice to see some more concluding remarks through the report.	4
---	--	----------

PRACTICAL APPLICATION AND DEP.

	Implementation, unfortunately, is a very long way from finished - a rough attempt made at parallelism, but neither complete nor partially working.	11
---	--	-----------

SKILLS FOR PROFESSIONAL PRAC.

	A few errors throughout the report - be sure to proof-read carefully.	2
---	---	----------