Design And Analysis Of A Plastic Parallel Programming System

Mark Jenkins

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh
2018

Abstract

This is the scond part of an MInf project which has spaned two years. In this project, we investigate the performance implications of plastic parallel programs which adapt to the current state of the host system, and explicitly cooperate to share and optimize the use of system resources.

In the first year of this project, we created an example of a parallel programming library which implemented these features, and gained some promising preliminary results into it's performance. In this second year, we develop upon this further by investigating a more insteresting parallel pattern...EXPAND

Contents

1 Background **3**

1.1 Introduction 3

1.2 Current Solutions 4

1.2.1 Contention Aware Scheduling 5

1.2.2 Plastic Programming 5

1.2.3 Skeleton Programming 5

1.3 Previous Work 5

2 Design **9**

2.1 Project Goal 9

2.2 Project Architecture 10

2.2.1 Parallel Pattern Selection 10

2.2.2 Performance Characteristics Investigation 11

2.2.3 Interesting Instances 12

2.2.4 Refined Investigation 13

2.2.5 Contention Aware Plasticity Analysis 13

2.2.6 Performance Characteristics of the Jacobi Pattern 14

2.3 Experiment Platform Design 14

2.4 Design of Experiment Platform 15

2.4.1 Basic Stencil Code 15

2.4.2 Adding Variable Configurations 15

2.4.3 Adding Plasticity 15

2.4.4 Experiment Scripts 15

2.5 Interesting Use Cases 16

2.5.1 Reasoning 16

2.5.2 Finding Interesting Use Cases 16

2.6 Selected Use Cases 18

3 Implementation **19**

3.1 Last Year's Implementation 19

3.2 Implementation Details 19

3.2.1 Issue - Producing a reliable parallel workload 20

3.2.2 Cold Caches 24

3.3 Testing Framework 24

3.3.1 Scripts! As opposed to last year! 24

4 Experimental Methodology And Program **25**

4.1 25

5 Results **26**

5.1 26

6 Future Work And Conclusions **27**

6.1 27

7 Bibliography **28**

Chapter 1

Background

In both the intro, and the chapter as a whole, we cover the current approaches to parallel programming. We will then explore the key ideas requisite to this project, such that we can discuss how they are combined and their implications, and give an overview. Then we will cover what was done in the previous phase of this project [6], and the intentions for this phase.

1.1 Introduction

Writing correct parallel programs is hard [7]. It introduces new difficulties such as race conditions and thread safety. Writing efficient parallel applications, where performance matters, is even harder [8]. The overhead of parallelism must be kept small enough such that the performance gains will outweigh it.

Current solutions range from exposing the programmer to all the low level primitives (POSIX Threads), to building on these primitives to provide a higher level language (OpenMP.) Other solutions focus on making parallel programming easier, usually employing skeleton code (Intel Threaded Building Blocks.)

In this project, we investigate plastic parallel programs with a focus on efficiency, which results in complex code. It is hard to ensure correctness, never mind performance. Since the ideas of this project will require the cooperation of multiple different programs, it is in our interest to make using these techniques as easy as possible, to encourage their use by programmers. Thus, to abstract this complexity away from the programmer, we employ skeletal programming. It also has the beneficial side effect of dividing the challenge into a pattern-by-pattern basis.

The main idea in this project is dynamic contention aware scheduling and optimisation. Contention aware scheduling is a means to address a particular problem. That problem is that programs running simultaneously on the same machine will interfere with each other, as they must share the resources available. This interference is not uniform, and, in fact, is dependent upon which programs are running, and the overall state of the system. Contention aware scheduling is the practice of taking into account this interference (contention) when scheduling programs, such that programs which work well together are scheduled to run on the same socket, thus reducing the interference (contention) between these programs.

It has been shown that this problem is an important factor in multiprogramming systems, and that it has performance implications [4]. The intention of this project is to investigate if we can exploit plasticity to further mitigate this contention, and what benefits this could bring.

In the first phase of this project [6], we investigated a basic parallel programming pattern, map-array. We incorporated this into a feasible estimate of a contention aware plastic parallel programming library, and measured the performance. The main conclusion was that such a system did not add a significant amount of overhead, and performance gains were feasible, but more investigation was necessary. To build on these findings, in this phase, we focus on the analysis of a more complex parallel pattern, without integrating it into a complete contention aware plastic parallel programming library.

1.2 Current Solutions

PThreads, MPI, OpenMP

Intel TBB, SkePu

1.2.1 Contention Aware Scheduling

1.2.2 Plastic Programming

1.2.3 Skeleton Programming

1.3 Previous Work

In the previous phase of this project, we designed, built, and tested a contention aware plastic parallel programming library. We analysed the potential gains from combining contention aware scheduling with plasticity, and assessed the overhead of using the library.

Design

For the implementation of our ideas, we need to make two modifications to a typical parallel program. We need to add plasticity to the program itself, and to introduce some from of communication and synchronisation with other programs currently running in the system.

- Plasticity

  To implement plasticity, we added the ability to vary three key aspects of the implementation of a single instance of the map-array skeleton:

  - Thread count - The number of threads we split the tasks between
  - Thread pinnings - The particular CPU core each thread runs on
  - Schedule - How to divide tasks between threads

  These are set before any worker threads are spawned. This lets us produce several different implementations by altering these parameters.

  The simplest method of implementing plasticity is to gracefully kill all worker threads, and restart them with the new parameters from where they left off. We chose this method to keep it simple. The cost incurred (closing and restarting each thread) would be proportional to how often we want to change parameters, and as such, we could hide it by making the workload arbitrarily large.

- Communication

  To implement communication and achieve synchronisation between instances of our library, we decided upon the use of a central controller program, which would communicate with each instance and instruct them what parameters to use.

When a new instance of our library starts, it registers with the controller program. The controller program then continually monitors the system for any changes, and reacts accordingly sending new parameters to it's registered programs. For our purposes, the controller program acted according to a predefined set of instructions, telling it what parameters to send and when. This was good enough for our tests, rather than the intended functionality.

In order for instances of our library to communicate with the controller program, we added a main thread, which would handle communication operations, and would update the worker threads with new parameters. This communication structure is illustrated in figure 1.1.

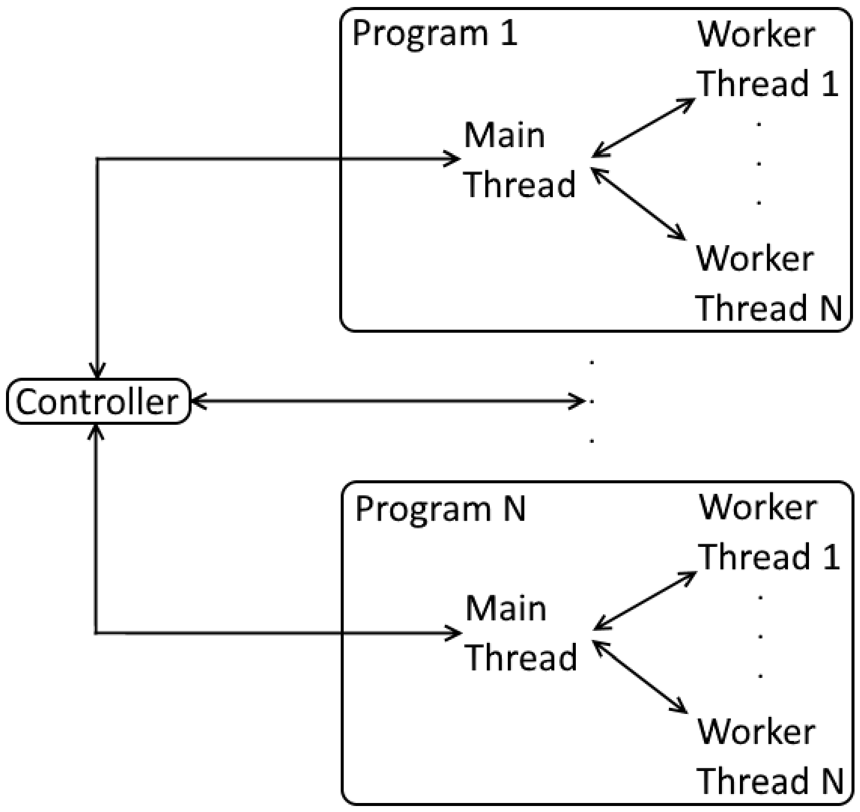


Figure 1.1: High level communication model of the system, with an arbitrary number of programs, with an arbitrary number of threads. Two way communication occurs between the controller and each main thread, and then between each main thread and its worker threads.

Results

The two main

Chapter 2

Design

In this section, we cover the design of the project as a whole, leaving the details of the system produced for chapter 3, which covers the implementation. We present a high level description of the plan for this project, which consists of several distinct steps. For this to be completely clear, we first describe the aims of this project, so that they can be kept in mind when reading the plan.

*** UPDATE *** We then detail the justification of and process of finding specific stencil codes with which to assess our system.

2.1 Project Goal

The main goal of this project is to investigate if we can gain some extra performance in parallel multi-programming systems by exploiting contention aware scheduling and plastic programming. In the last phase, we investigated the design and performance of a complete plastic parallel programming library, and created some prototype software for such a system. The conclusion was that such a system would be viable, as the overhead introduced could be easily overcome / was minimal.

This phase, our focus will just be on whether we can realise that extra performance. Therefore, we do not need to design and implement a complete programming library (as was done in the last phase,) we just need some stripped down software which would let us carry out our experiments.

2.2 Project Architecture

Here we present an itemised list of steps, established at the start of this phase of the project, which form the high level plan for this phase of the project. We go on to expand upon each point with explanations and justifications for the choices made.

- Select parallel pattern
- Investigate performance characteristics
- Find interesting instances of pattern
- Refined investigation of interesting instances
- Perform contention aware plasticity analysis

2.2.1 Parallel Pattern Selection

In the previous phase, we focused on the map-array parallel pattern. This pattern is fairly straightforward in terms of complexity and performance characteristics, in that providing more cores and corresponding threads to the computation will generally improve performance. Because of this, each program will effectively be able to saturate all the cores allocated to it. This is the ideal for any efficiency focused application, and as a result we have little room for improvement. Therefore, this year, we decided to focus on a more interesting parallel pattern which wouldn't necessarily improve with extra cores.

There are many options for us to choose from [1] [2], (and other options could be explored in future research,) including pipelines, recursive splitting, and geometric decomposition. Ideally we want a pattern which is widely used in real applications, providing many interesting use cases, and possessing more interesting performance characteristics than map-array. Accordingly, we selected stencil codes as the parallel pattern we would focus on.

Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern, called a stencil. Typically, after one pass of the array, some sort of convergence test is ran to check if the computation is complete. They are used in computational fluid dynamics, in linear algebra for solving partial differential equations [10] [3] [9], in image processing, and cellular automata. In general, any algorithm which operates on finite grids can be formulated as a stencil code.

Stencil codes should have more interesting performance characteristics since they have inherent data sharing properties. The array is divided between threads, and when a thread updates a particular point in the array, it will also access it's neighbours, and these neighbours may be in another thread's portion of the array.

For selecting interesting instances, we can vary the size of the array, the computation performed at each point, and the number of iterations (passes over the array) before the computation is complete. In a real application, this would vary, however we want a reproducible, predictable instance. So we simulate a convergence test and end the computation after N iterations.

2.2.2 Performance Characteristics Investigation

The intention behind this part of the project is to get a feel for the performance characteristics of our chosen parallel pattern. To do this, we plan to create a basic implementation of the parallel pattern, and generate a basic, synthetic, problem to run our pattern on. We will add the ability to vary characteristics of both the problem and the implementation, allowing us to perform this basic investigation. In particular, we would like to vary the following parameters:

Problem parameters

- Grid size
- Computation kernel
- Number of iterations

Implementation parameters

- Number of workers
- Thread pinnings

An implementation which could vary these parameters would give us a good starting point, and provide us with a platform to perform this performance charactreristics investigation.

Talk about:

- Barrier investigation
- Convergence test
- Different sized grids

- General: Size of work affects performance curve. - EXPECTATION! Actual result in results section!
- Could vary more parameters in implementation, e.g. use different data structures

2.2.3 Interesting Instances

In selecting interesting instances, we want to see some instances with different performance graphs, and some instances which utilise different resources. The reasoning behind this is that they should make for valuable test cases for our contention aware plasticity experiments. The differing performance curves because partitioning resources may be asymmetrical, and the differing resource utilisation because these programs may be an example of programs that work well together, i.e. they have less contention since they require different resources.

This section will be informed by the results of the previous one, the performance characteristics investigation. But we are hoping that by varying the total amount of work to be done, (grid size and computation kernel complexity,) we can produce at least two distinct performance curves (one with a small amount of work, meaning parallelisation overhead will be more significant, so more threads may not necessarily be a good thing, and one with a large amount of work, which can always use more threads than we have.)

In addition to varying the work size, we can change the computation kernel to utilise different resources, e.g. CPU and RAM. This will give us at least two more instances per resource.

Finally, we can add a random component to the computation kernel. Many stencil does do not have an equally distributed amount of work across the grid. Randomising the amount of work to be done at each point in the grid models this situation. This will double our number of instances, one with a random component, and one without.

There is a motivation for keeping our number of test instances down, and in fact a reason why we cannot exhaustively test every instance. Each instance will require a significant amount of experiments, which can take hours or even days. Increasing the number of test cases will increase the number of combinations of pairs for the pairwise contention aware plasticity experiments, making a large number of test instances simply impractical. Just finding the optimal number of threads for a given number of CPU cores requires $(C + 12) * C$ experiments, where C is the number of cores.

2.2.4 Refined Investigation

Here we perform a refined investigation of each chosen instance. The aim is to establish performance characteristics so that they can be used as a baseline in future experiments. Since performance may vary from machine to machine, we will run each experiment on multiple machines, giving us a comparison across machines.

To reduce the complexity of our experiments, we restricted our implementation parameters to just the number of workers and what CPU cores they are pinned to. Again, since our experiments are so time intensive, to reduce complexity we will focus on finding the optimal number of threads for a given number of provided CPU cores. Each thread will have access to any of the cores provided, meaning that this is akin to just limiting the CPU cores of a program, and as such this represents a typical implementation without any plasticity or contention aware scheduling, giving us the baseline performance.

2.2.5 Contention Aware Plasticity Analysis

In order to test if extending contention aware scheduling with plasticity is worth it, we need to experiment with programs running simultaneously, as in a real multiprogramming system. If we have N instances, we will have $\binom{n}{2}$ experiments, since this is the number of unique combinations. Therefore, we may again need to select only the combinations that we think will be interesting.

Here we present four experiments, which will be run on each pair of instances:

- Both programs running simultaneously, using as many cores as they can, unpinned. This configuration mimics that of a current multiprogramming system, giving us our standard baseline of the instances running in contention, and compared with the results of the previous experiments, we will know how much this contention costs.
- Similar to the above experiment, only each instance will use the optimal number of threads. Each instance will still have access to all the CPU cores. This is again similar to a current multiprogramming system, only in this case, we assume that the programmer or user has performed some sort of performance analysis and has determined the optimal number of threads to use.
- As the above experiment, only with plasticity. We take no consideration to contention here, we are simply augmenting the previous case with plasticity to see what effect this has. This is more like a situation where the programmer is using some sort of plastic library, and either the library or

the programmer has performed some sort of performance analysis to determine the optimal configuration. To execute this experiment, we will be setting the parameters manually, unlike the situation we are simulating.

- As the above experiment, with plasticity, and enhancing this with contention awareness. In this experiment, we partition resources between programs, taking into account their contention. We will also exploit the plasticity of our programs, manually selecting the relevant parameters. This represents the situation targeted by this project, a programmer using a plastic, contention aware, parallel programming library. This should be the most interesting experiment, whose results wil determine the outcome of this project.

****

OLD CHAPTER HERE

****

2.2.6 Performance Characteristics of the Jacobi Pattern

2.3 Experiment Platform Design

Run time of optimal threads experiments: N*N, runtime of simultaneous program experiments?

Two array thing -¿ Double buffering

Since the purpose of this year of the project is to analyse the performance characteristics of stencil codes, we do not need the full capabilities of a plastic parallel programming system. We just need enough functionality to assess the performance.

Previous Phase Analysis

Looking at the report produced last year, we came to the conclusion that too much work was put into creating the system rather than performing experiments. As a result, we did not have as detailed analysis as we would have liked.

2.4 Design of Experiment Platform

2.4.1 Basic Stencil Code

The first step was to implement a basic stencil code.

Two arrays are used, one to store the values from the previous iteration, and one to contain the newly computed values. After an iteration, the roles of these arrays are reversed. This technique of using two arrays is done to minimise the necessary synchronisation overhead.

Arrays are partitioned between threads, with a row level granularity. Shared data occurs at the edges of a particular thread's chunk.

Parallel code implemented using the c++ threads library (Based upon pthreads.)

2.4.2 Adding Variable Configurations

variables numthreads etc kernels [5]

1. Number of iterations
2. Thread pinning
3. Combinations of multiple different workloads
4. Grid size
5. Number of workers

2.4.3 Adding Plasticity

plasticity

2.4.4 Experiment Scripts

experiment scripts

2.5 Interesting Use Cases

2.5.1 Reasoning

In order to be able to do our investigations, we need to select a collection of interesting use cases of stencil code.

The potential for experiments is massive, due to the large amount of variables we have access to. In our investigation, we have access to the following:

EXPAND UPON THESE

*** DIFFERENTIATE INSTANCE PARAMETERS AND PROGRAM PARAMETERS ***

1. Number of iterations
2. Thread pinning
3. Combinations of multiple different workloads
4. Grid size
5. Number of workers

All of these can be changed from stage to stage *** WHAT'S A STAGE? HOW PROGRAM PARAMS CAN BE CHANGED! IDEAL - GRACEFUL, BUT NOT NECCECARY ***. Beyond these parameters, we can have many different combinations of programs running simultaneously.

For the purposes of this project, we will select a few interesting use cases of the stencil pattern, and focus on testing these in a few different configurations.

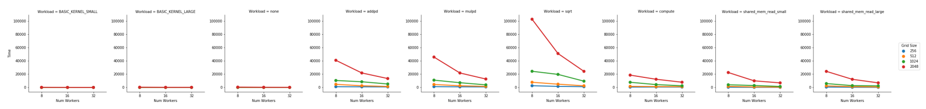
2.5.2 Finding Interesting Use Cases

In order to find interesting use cases, we performed an initial foray into the experiment space we have access to.

*** SEPARATE INTERESTING USE CASES INTO USE CASES & SYSTEM PARAMETERS

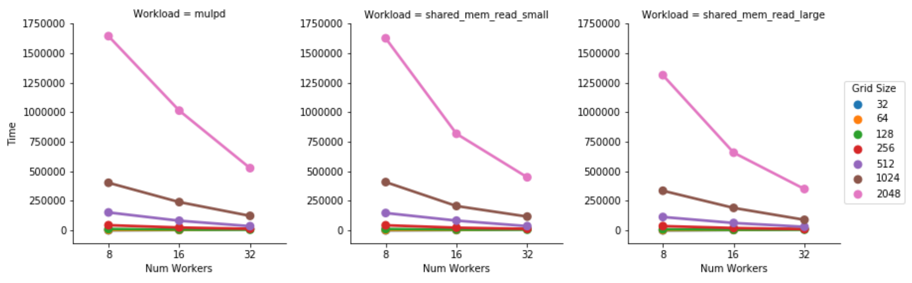
First

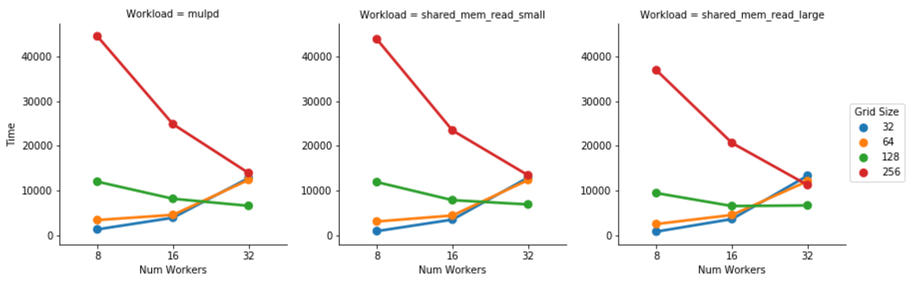
Firstly we compared the performance characteristics of each of our different workloads, and how they performed with different numbers of threads.



Second

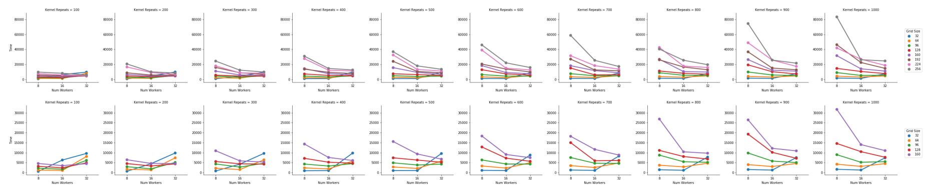
Next we picked some interesting kernels form the previous experiment, and investigated them further, with some smaller grid sizes. This was because we were interested in a use case where at a certain point, adding more threads is no longer optimal.





Third

Next, we focused on the mulpd kernel, and and tested a variety of kernel repeats, and some more fine grained grid sizes. This is so that we could identify the point at which, as the grid size shrinks, increasing the number of threads will be sub-optimal.



As you can see from the graphs, for gridsizes 32-128, going beyond 16 threads did not increase performance in all cases.

Another interesting take from the graphs is that we can see that as the grid size increases, the runtime curves 'flip' at 32 threads such that smaller grid sizes take longer. However, as the number of kernel repeats increases, this flipping gradually lessens. This is because the increased independent workload per thread starts to

2.6 Selected Use Cases

List selected patterns

Chapter 3

Implementation

system that was produced for this purpose, and the design decisions behind it.

****

- Pick relevant points, e.g. Jacobi code from web

- Test harness/scripts etc

- Graphing

- ...

****

3.1 Last Year's Implementation

Controller etc...

3.2 Implementation Details

In setting up the most basic instance of our pattern, our program will create an array of a given size, calculate row allocations for a given number of threads, and setup data structures for other features such as the conversion test. This is the setup phase of the program.

In the next phase, our program will perform the performance test. It spawns worker threads, and performs the computation, recording the time it takes. Each worker will

3.2.1 Issue - Producing a reliable parallel workload

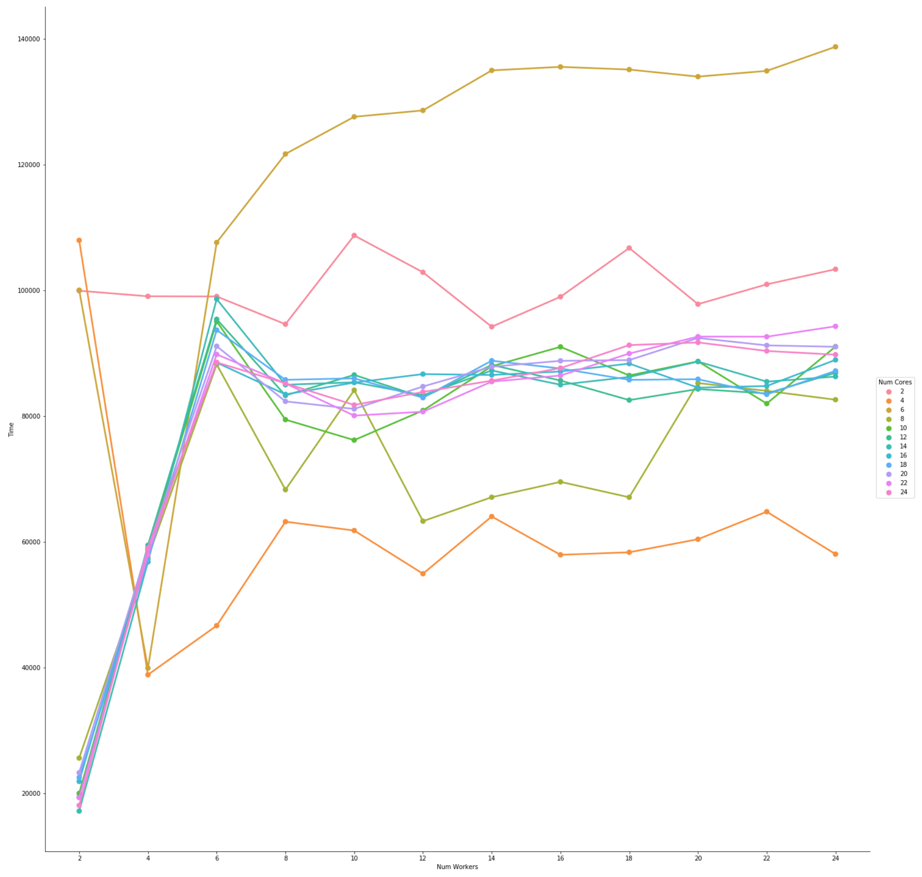
Power measurement techniques on standard compute nodes: A quantitative comparison DOI: 10.1109/ISPASS.2013.6557170

https://people.seas.harvard.edu/ apw/stress/

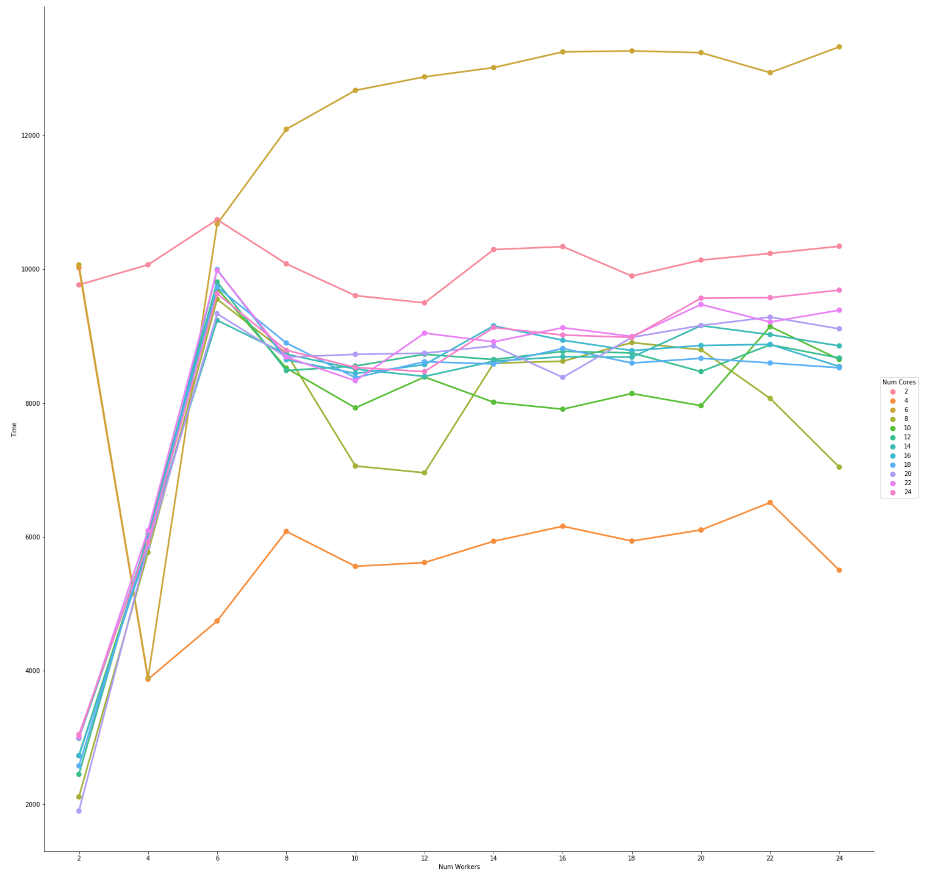
Issue - Compiler optimises code removing redundant loops. We need redundant loops for workload generation.

Tried - Allocating memory, creating values, reading values for computation - Introduces many memory reads, creating of values either slows computation or must be done outside worker thread beforehand (Finicky, annoying to do).

Tried - Using rand() for sqrt() seed. Not thread safe, produced confusing graphs. e.g.



This took a while to diagnose, first we thought this was due to cold cache issues, as the first graph was produced with one run and only one iteration (e.g. first convergence test was successful.) So we ran the experiment again with 10 repeats, and obtained the following graph:



Note these graphs were produced with num workers increasing in stages of two.

In diagnosing this, I discovered strange behaviour. Running a computation with one thread resulted in times of 500ms, and with two threads it would take 20,000ms. Not what we would expect at all! We suspected that the strangeness causing this was also causing the strange graphs, so we focused on investigating this since it could be ran quickly (as opposed to making complete graphs.)

This behaviour continued with using multiple threads, but pinning them to the same CPU (500ms). However, restricting more threads to just two CPUs resulted in the same behaviour.

Function being used was sqrt(rand()). sin(rand()) also tried, same result.

Must be rand! So looked for other methods to prevent compiler optimisation, used volatile with rand(constant). Behaviour corrected. Strangeness down to rand() not being thread safe! This made sense with our behaviour, since one thread ran much faster, and four threads pinned to one core was the same.

This was because, with them all running on the same core, they would never be running simultaneously, so calls to rand would never interfere with each other.

Tried - Using inline assembly so we know what is going on - Still needs function wrapper, preallocated memory, difficult to debug/understand (Usual issue), unwieldy. (Used volatile keyword?) Tried - Computing sum of sqrt(loop iterator) - Cannot have correct return function Tried - Using volatile keyword. Not used in most programs!

Quoting from the C++ Standard ($7.1.5.1/8)

[..] volatile is a hint to the implementation to avoid aggressive optimisation involving the object because the value of the object might be changed by means undetectable by an implementation.[...]

Volatile was successful!

But this means volatile value not written to cache, is written straight to memory. Is this a performance problem? Using a volatile pointer might solve this?

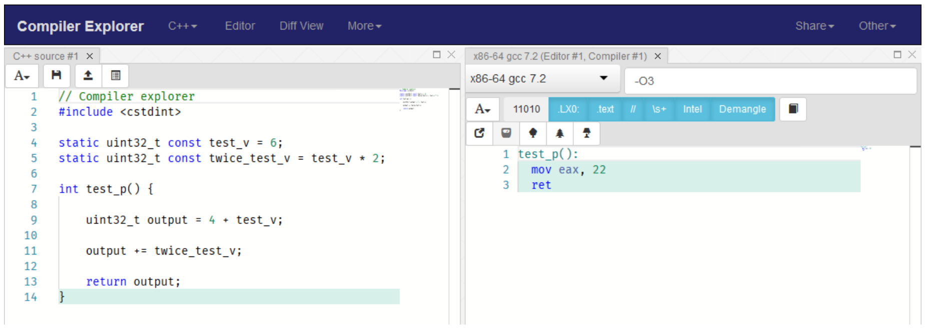
Tool used to diagnose compiler optimisation - https://godbolt.org/ Compiler explorer. INCLUDE SCREENSHOTS!

This is an example of why parallel programming is hard! Especially for performance!

(My implementation of a counter barrier is another example! Mention when discussing)

Other difficulties:

Which is better? declaring a variable in #define or using a static uint32_t const var? Avoiding the preprocessor would be better from a programming perspective (type safety etc,) but would it be as fast? (Using compiler explorer to confirm, yes, compiler optimises this.)



x86-64 gcc 7.2 -O3

```
// Compiler explorer
#include <cstdint>

static uint32_t const test_v = 6;
static uint32_t const twice_test_v = test_v * 2;

int test_p() {

    uint32_t output = 4 + test_v;

    output += twice_test_v;

    return output;
}
```

```
test_p():
  mov eax, 22
  ret
```

3.2.2 Cold Caches

3.3 Testing Framework

3.3.1 Scripts! As opposed to last year!

Complex testing and graphing framework, allows machines to be added and experiments to be set up easily. Experiments can be easily duplicated accross machines.

Chapter 4

Experimental Methodology And Program

4.1

Chapter 5

Results

****

- Results of each of the previous chapters

****

5.1

Chapter 6

Future Work And Conclusions

6.1

Chapter 7

Bibliography

[1] 2017.

[2] 2017.

[3] Anzt, H., Dongarra, J., and Quintana-Ort, E. S. Adaptive precision solvers for sparse linear systems. *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing - E2SC '15* (2015).

[4] Collins, A., Harris, T., Cole, M., and Fensch, C. Lira. *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '15* (2015).

[5] Hackenberg, D., Ilsche, T., Schone, R., Molka, D., Schmidt, M., and Nagel, W. E. Power measurement techniques on standard compute nodes: A quantitative comparison. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013).

[6] Jenkins, M. Design and analysis of a plastic parallel programming system. 1–60.

[7] McKenney, P., Gupta, M., Michael, M., Howard, P., Triplett, J., and Walpole, J. Is parallel programming hard, and if so, why?

[8] Sadowski, C., and Shewmaker, A. The last mile. *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10* (2010).

[9] Wang, H. A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Transactions on Graphics 34*, 6 (2015), 1–9.

[10] Yang, X. I., and Mittal, R. Acceleration of the jacobi iterative method by factors exceeding 100 using scheduled relaxation. *Journal of Computational Physics 274* (2014), 695–708.