Towards A Plastic Parallel Programming System

Mark Jenkins

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

Abstract

In this project, we investigate the performance possibilities of plastic parallel programs, which adapt to the current state of the host system and explicitly cooperate to share and optimise 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 for a single parallel programming pattern, that of map-array. We investigated the overhead incurred by such a system, concluding that it was insignificant, and we produced some promising preliminary results into its performance.

In this second year of the project, we develop upon this by investigating a less predictable parallel pattern, that of stencil codes, with a focus on potential performance to be gained from plasticity when run in contention with other programs. We designed and implemented a synthetic test stencil program, along with testing scripts and data analysis applications. We then evaluated the synthetic test program with two variants running in contention in a variety of scenarios, designed ultimately to reveal the potential for performance gains attainable by a contention aware plastic parallel programming library over and above the normal performance benefits attained by parallelism. The results of these experiments showed potential speedups from 1.03 to 2.44, compared to a non contention-aware system. We then investigated some more complex scenarios, involving three variants running in contention, which showed speedups from 1.13 to 2.43, compared to a non contention-aware system.

Contents

Chapter 1

Background

In this chapter, we cover the current approaches to parallel programming. We then explore the key ideas requisite to this project, in order to discuss how they are combined and their implications. Finally, we review what was done in the previous year of this project [20], and some lessons we carried forward to this year.

1.1 Introduction

The three pertinent ideas to this project are those of plasticity, contention aware scheduling, and skeleton programming. We briefly cover these ideas here, and provide more detailed explanations later.

A plastic program is a program which can adapt to its current environment. For example, a program may utilise plasticity in order to alter its operations to suit a particular architecture. In our case, the program will alter its implementation in response to the current state of the system, with a focus on which other programs may be active.

Contention aware scheduling is the idea of taking into account possible contention when scheduling programs, and is particularly relevant when considering machines with multiple sockets. We may know that certain programs run well together, and so we would schedule them to run together on the same socket. Similarly, we would try to keep more contentious programs apart, running on separate sockets.

Skeleton programming involves the use of an algorithmic skeleton. The skeleton provides the outline of the algorithm, and the user supplies some code to pro-

duce a complete program. With skeletons for common programming patterns, these are particularly useful for parallel programming, as they abstract away complexity, making producing correct and efficient parallel programs easy.

The overall aim of this project is to investigate if we can gain performance by utilising all of these ideas with parallel programming.

1.2 Parallel Programming

Writing correct parallel programs is hard [23]. It introduces new difficulties such as race conditions and thread safety. Writing efficient parallel applications, where performance matters, is even harder [28]. The overhead of parallelism must be kept small enough such that any performance gains will outweigh it.

Current solutions range from providing the programmer with low level primitives (POSIX Threads [4]), to abstracting away these primitives, resulting in a higher level language (OpenMP [7].) Other solutions focus on making parallel programming easier, usually employing skeleton code (Intel Threaded Building Blocks [19].)

In this project, we are combining the ideas of plasticity and contention aware scheduling with parallel programming, with a focus on efficiency. This, along with the difficulty of parallel programming, makes it hard to ensure correctness, and particularly efficiency. Since the goals of this project will require the cooperation of several 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.

1.2.1 Current Solutions

Since we are investigating the feasibility of a new parallel programming library, it is worth giving a quick overview of some current parallel programming solutions. These include:

- Pthreads (POSIX Threads [4])
- MPI (Message Passing Interface [34])
- OpenMP (Open Multi-Processing [15] [7])
- Intel TBB (Intel Threading Building Blocks [19] [9])

- SkePU ([12] [13])

Pthreads and MPI represent the most direct methods of parallel programming, whereas OpenMP, Intel TBB, and SkePU focus on making parallel programming simpler.

Pthreads is a set of interfaces (functions, header files) for multi-threaded programming. A process can contain multiple threads, all of which collaborate on the same program. It is targeted towards shared memory architectures, so threads share global memory (data and heap), but each thread has its own stack (automatic variables). The low level nature and the portability of Pthreads makes it suitable for building other parallel programming mechanisms.

MPI is a message passing standard, and as such it describes how processes can communicate. The message passing model is attractive because it provides portability, and it can be used for communication in both distributed or shared memory systems, networks of computers, or any combination of these. Its versatility and efficiency allows for portable and scalable parallel programs, and as a result, it is the dominant model used in the high performance computing industry today [34].

OpenMP is an API which supports shared memory multiprocessing programming in C, C++, and Fortran. It consists of compiler directives, library routines, and environment variables. Like Pthreads, OpenMP is intended for shared memory architecture, however it abstracts away many of the primitives which are present in Pthreads. As such it provides a simple and flexible interface, whilst also being portable and scalable.

The approach taken with some computer clusters is a mix of both OpenMP and MPI, such that OpenMP is used for parallelism within a node while MPI is used for parallelism between nodes. The main benefit of this is reduced communication bandwidth [32].

Intel TBB is a C++ template library developed by Intel for parallel programming on multi-core processors. It breaks down computations into tasks, constructing dependency graphs according to algorithmic skeletons (described in section 1.5). The library then schedules threads to execute these tasks, respecting the dependencies.

Designed to make parallel programs easier to write, it has the advantage of low-overhead polymorphism, since templates are a compile-time construct, meaning there is little loss in performance compared to hand written programs, provided your application can be described using the templates.

SkePU is an open-source skeleton programming framework for multicore CPUs and multi-GPU systems. It is a C++ template library with data-parallel

and task-parallel skeletons, and support for execution on multi-GPU systems both with CUDA and OpenCL.

Similar to Intel TBB, it attempts to simplify parallel programming through the use of templates (here called skeletons.) However, SkePU adds features such as multiple back ends, including OpenMP, OpenCL, CUDA, and multi-GPU OpenCL and CUDA. It also provides a tuning framework for context-aware implementation selection, allowing it to select the expected fastest implementation at each skeleton call. This differs from our system in that it performs static analysis before the program is run, and as such it cannot react to changes in the system at runtime. It also does not utilise plasticity at a more fine grained level than which language (or hardware) to use.

Parallel programming introduces a whole host of complications ranging from race conditions to limited scalability. These new problems can certainly be overwhelming to a traditionally sequential application programmer, so much so that there are entire books dedicated to the use of parallel programming methods. Given that this is already so hard, and we require plasticity on top of this, we must simplify what we present to the programmer, otherwise it would be too formidable. Like Intel TBB and SkePU, we do this by utilising skeletons/templates.

1.3 What Is Contention Aware Scheduling?

We use contention aware scheduling as a means to address a particular problem. In our case, that problem is that programs running simultaneously on the same machine will interfere with each other, as they must compete for resources. This interference is not uniform (as can be seen in figure 1.1), 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 overall contention.

LIRA [8] shows that throughput gains of 3-7% can be seen with this technique. Extending this with plasticity would mean that we can control which program runs where, and the implementation each program uses, allowing us more scope to mitigate contention.

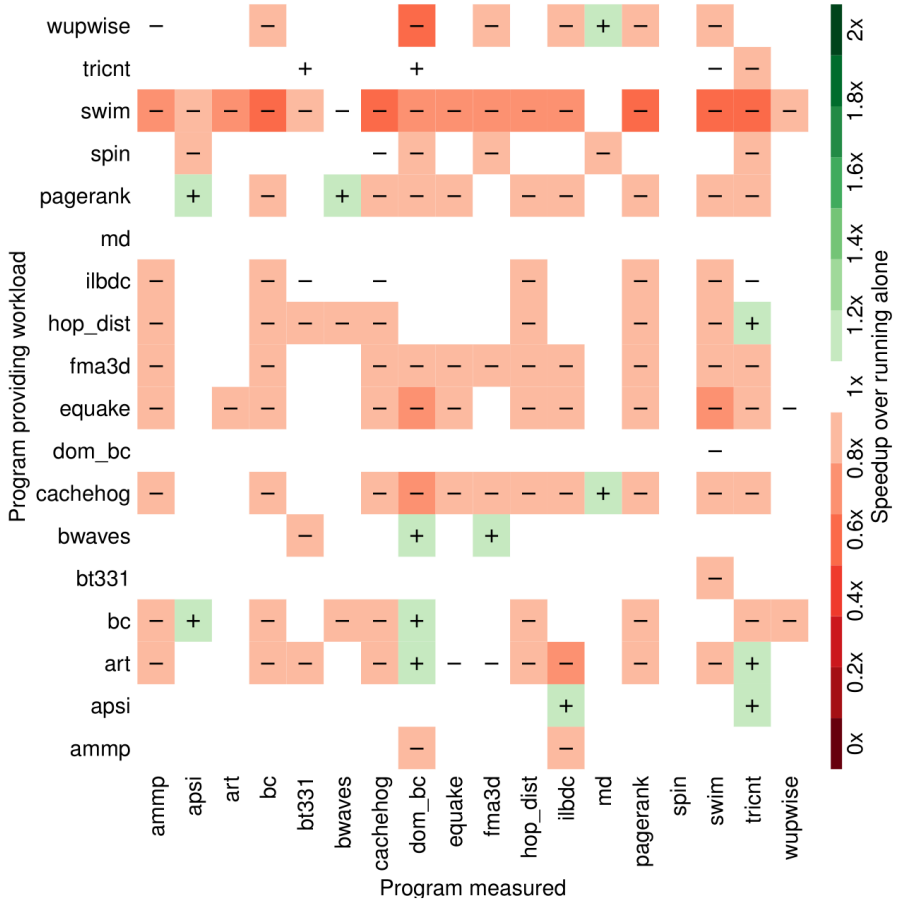

Figure 1.1: Taken from the LIRA paper [8]: Pair-wise speedup of programs, comparing sharing a socket to running in isolation. Boxes annotated with a − indicates where performance decreased when sharing a socket, and + shows increases. For example, looking at the programs bc and swim, we see a speedup of around 0.6x, whereas for the programs apsi and pagerank, we see a speedup of around 1.2x.

1.4 What Is Plastic Programming?

When programming an algorithm, there are often choices which can greatly affect performance, and the optimal choice depends on the circumstances. As an example, for a sorting problem with a large input size, radix sort would perform best. For a small input size, insertion sort would be better. Parallel programs

tend to have more choices, particularly with a view to keeping overhead reasonable.

Plastic programming is the idea of a program which includes multiple implementations, which can be chosen between dynamically. The mechanism for selecting the implementation can vary, back with our sorting example, we could assess the input size, and run radix sort until the problem is small enough to use insertion sort. Figure 1.2 illustrates such a situation.

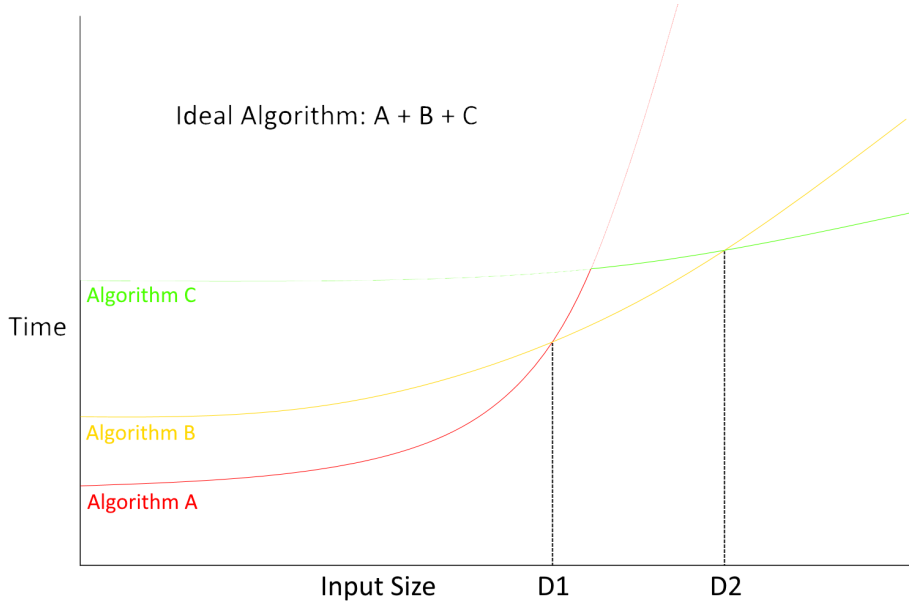

Figure 1.2: A graph showing three algorithms with different runtime curves, which depend upon the array size. Combining these algorithms would provide an improved algorithm, with D1 and D2 showing optimal decision points where a plastic programming system should switch algorithms

1.5 What Is Skeleton Programming?

Skeleton programming is a high-level programming model, which will allow us to abstract away the details of parallel programming, plastic programming, and contention aware scheduling. The idea is that the skeleton provides the core structure of an algorithm, with the user providing some code, producing a correct and hopefully efficient program [6]. Multiple skeletons can be combined to produce a more complex program, for example map and reduce skeletons are commonly combined. This makes skeletons powerful tools for easily creating

clean, complex programs.

This makes using our library much easier, and means that we can assess the program's complexity, since we know the algorithmic details of the skeleton.

1.6 Review of Previous Year

The main goal of this two year project is to investigate if we can gain some extra performance in parallel multi-programming systems by exploiting contention aware scheduling and plastic programming. There has been no published previous work on runtime dynamic plastic parallel skeletons, as described in this project.

In the first year of this project [20], 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, with a focus on understanding and minimising overhead. In this section, we cover the design of this library, and the information gained from its analysis.

Design

In order to make such a library, we needed to make two modifications to a typical parallel program. We needed to add plasticity to the program itself, and to introduce some form of communication and synchronisation with other programs running in the system. This synchronisation could be distributed, or centralised. For the initial investigation, we decided that centralised was more appropriate to keep things simple.

- 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 allowed us to generate several different implementations by altering these parameters.

  However, this alone did not allow us to dynamically switch implementations during runtime. The simplest way to make this dynamic would be to gracefully kill all worker threads, and restart them with the new

parameters. We chose this method to, again, 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, its overall impact depended upon the frequency of change. We also investigated the impact this method had, finding it to be negligible.

- Communication

  As discussed previously, we decided upon the use of a centralised controller program, which would communicate with each application running an instance of our library and instruct them what parameters to use.

  When a new instance of the library starts, it registered with the controller program. The controller then continually monitored the system for any changes, and reacted accordingly sending new parameters to each registered program. For our purposes, the controller acted according to a predefined set of instructions, telling it what parameters to send and when. In a more developed system, the controller would monitor the system and send updates programmatically.

  To allow instances of our library to communicate with the controller, we added a "main" thread, to handle communication operations, and update the worker threads with new parameters. This communication structure is illustrated in figure 1.3.

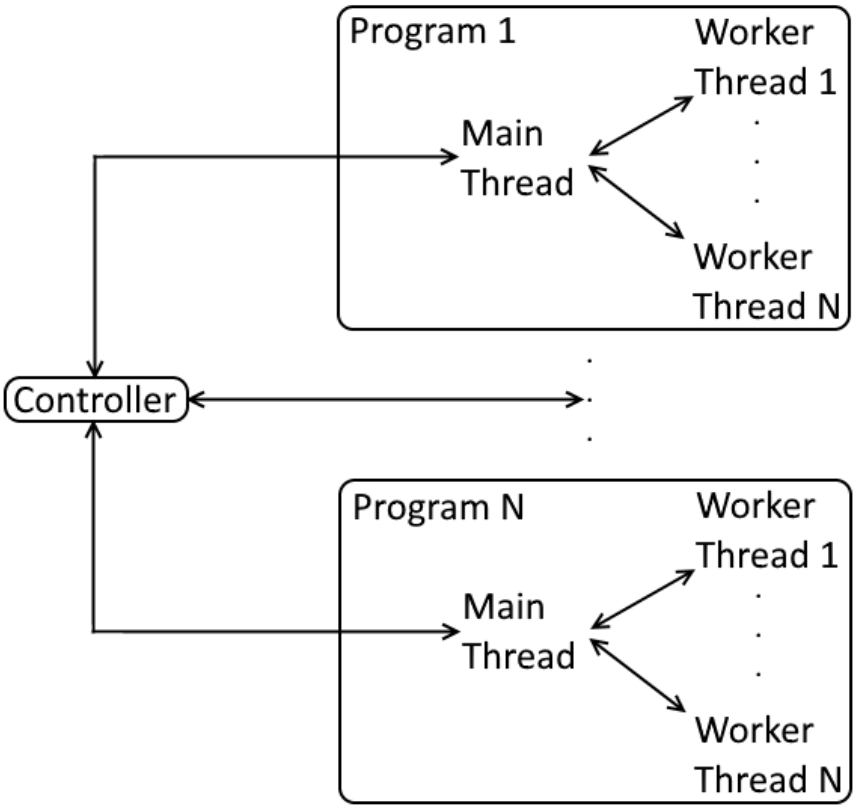

Figure 1.3: High level communication model of the system, with an arbitrary number of programs, each with an arbitrary number of threads. Two way communication occurs between the controller and each main thread, and then between each main thread and its worker threads.

Results

The two aims of the previous year were:

- Assess the feasibility and the overhead introduced by a contention aware plastic parallel programming library
- Assess the potential performance gains to be had by extending contention aware scheduling with plasticity

Regarding the first item, we found that while programming the library was com-

plex, using it was straight forward, as expected. As for the overhead, compared to running the programs with hand coded solutions, we found it acceptable. Thus, it is feasible if we see reasonable gains from adding plasticity to contention aware scheduling, that the overhead could be amortised.

For the second item, we found that whilst the results were promising, the map-array pattern was limited in the performance gains to be had from plasticity.

Lessons to be Carried Forward

The performance results of the previous year showed promise, but further exploration was required, which is our focus this year. We saw some gain in certain circumstances, but we concluded that the map array pattern used was too predictable, and doesn't have much scope for plastic improvement. However, it would still be reasonable to use our library for map array patterns, since other programs on the system may benefit.

This issue is why we decided to focus on the stencil code pattern this year, as it is much more complex to parallelise, with more scope for our decisions to have an impact.

Chapter 2

Design

In this chapter, we cover the overall design of the project, leaving the details of the systems produced for chapter 3, which covers the implementation. We first describe the aim of this project, and our goals for this year. Next we present a high level plan for this year of the project.

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 previous year, 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 was minimal.

This year, our focus will be on what, if any, performance we can gain from the use of such a system. Therefore, we do not need to design and implement a complete programming library, (as was done in the previous year,) we just need some stripped down software which would let us carry out our plastic performance experiments. Packaging this into a complete system could be work for a future student, this is discussed further in section 6.2.

2.2 Project Architecture

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

1. Select parallel pattern
2. Investigate performance characteristics
3. Find interesting instances of pattern
4. Perform refined investigation of interesting instances
5. Perform contention aware plasticity analysis

2.2.1 Parallel Pattern Selection

In the previous year, we focused on the map-array parallel pattern. This pattern is fairly predictable 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 [24] [25], which 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 interesting use cases, and possessing less predictable 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 the elements of an array, (we refer to this as the data grid,) according to some fixed pattern, called a stencil. Typically, after one pass of the array, some sort of convergence test is run to check if the computation is complete. Stencil codes are used in computational fluid dynamics, linear algebra for solving partial differential equations [37] [2] [35], image processing, and in cellular automata. In general, many algorithms which operate 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 data grid is divided between

threads, and when a thread updates a particular point in the grid, it will access the point's neighbours, and these neighbours may be in another thread's portion of the grid.

For selecting interesting instances, we can vary the size of the data grid, the computation performed at each point, and the number of iterations (passes over the array) before the computation is complete. In a real application, the number of iterations would vary, however we want a reproducible, predictable instance. To this end, we simulate convergence tests, and manually 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 need to create an implementation of the parallel pattern, and generate a synthetic problem to run our pattern on. We 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

A synthetic stencil with these features would give us a good starting point, and enable us to perform our experiments. From here, we can begin to explore the performance characteristics of such a stencil.

Between iterations, stencil codes use a barrier, in order to synchronise threads. An interesting experiment would be to investigate what effect this has on performance, since it requires communication and synchronisation between threads. The cost of this operation is dependent upon the number of threads, and the work that they do, both aspects of the program we wish to vary for later experiments. This is the first of the two experiments we plan for the performance characteristics investigation.

Another mechanism which stencil codes use between iterations is some sort of convergence test. A convergence test essentially is a test for whether a computation is complete. After a successful test, we end the computation. Similar to barriers, they require communication and synchronisation between threads, making their performance impact variable. Unlike barriers however, they are not always essential to the computation, and are application dependent. An investigation into the impact of the convergence test would be a valuable insight. This is the second of the two experiments we plan for the performance characteristics investigation.

Here, we discuss a prediction of how the stencil codes will behave. An artefact of a large grid size is that it is less likely that threads would interfere, as the border between their section of data and their neighbours has the same size, but their overall area has increased, making it less likely that two threads are on their respective borders simultaneously. This is illustrated in figure 2.1

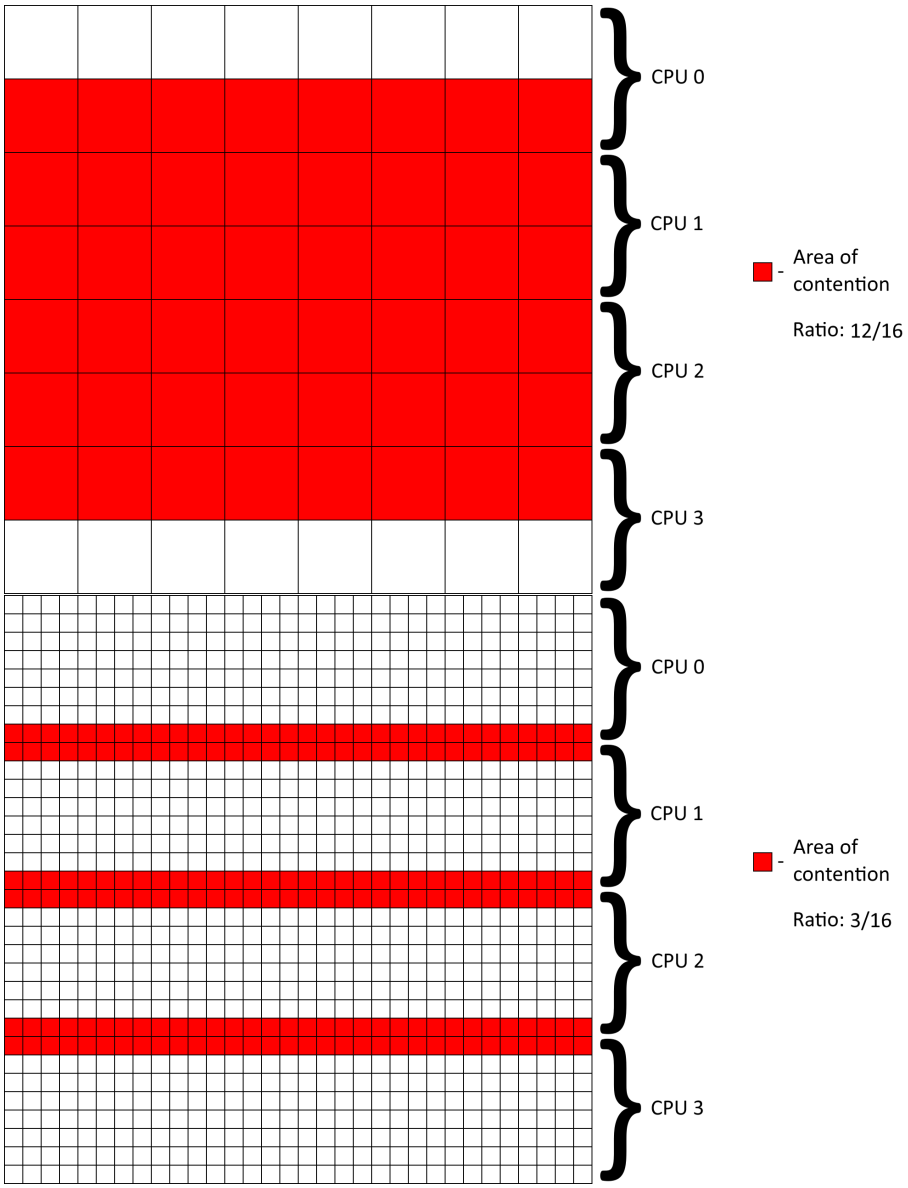

Figure 2.1: A visualisation of how the areas of contention change when we increase the gridsize. Different threads can only encounter contention along the edges of their designated area, so as we increase the size of the grid, the ratio of area where a thread may experience contention compared to the area where it cannot shinks.

Comparably, with a more complex, time consuming kernel, threads are less

likely to interfere, for similar reasons. Instead of being more distant in space, they are more distant in time, making it less likely that two threads are on their respective borders simultaneously.

The point of all this is that the interference between threads should decrease as the grid size or kernel complexity increases, and as such, we should expect better scaling of performance with large grid sizes and complex kernels. This is not only because of the reduced interference, but also because the proportion of useful work to overhead is larger, meaning that the overhead costs can be more easily "hidden."

2.2.3 Finding Interesting Instances

In this section, and in this report as a whole, when we discuss performance curves, we refer to the curve on a graph of the runtime of an instance, against the number of threads used. Examples of real performance curves can be seen in section 5.1, with figures 5.1 and 5.3.

In selecting interesting instances, we want to find instances with different characteristic performance curves, and some instances which utilise different resources. The rationale behind this is that they should make 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. 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, if time allows, we could produce additional instances by adding a random component to the computation kernel. Many stencils 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, and as such, it may result in an impractical number of experiments.

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 could 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 $(C + T) * C$ experiments, where C is the number of cores, and T is how much we "overshoot" the number of cores. We do this because we may gain some interesting results from having more threads than cores.

2.2.4 Finding Optimal Thread Counts

Here we will perform a refined investigation of each chosen instance. The aim is to establish the performance characteristics of each interesting instance we find, such that they can be used as a baseline for comparison (one of the program running in isolation.) In previous experiments, we were interested in the features of the stencil code itself, (e.g. barrier synchronisation,) however now we are interested in the overall performance, and since performance may vary from machine to machine, we will repeat each experiment on a variety of machines, to give us a comparison.

To reduce the complexity of our experiments, we will restrict our implementation parameters to just the number of workers and what CPU cores they are pinned to. Again, since our experiments will be so time intensive, to reduce complexity we will focus on finding the optimal number of threads for every possible 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 Experiments

In order to test if extending contention aware scheduling with plasticity is worth it, we need to experiment with programs running simultaneously, on a real multiprogramming system. We focus on two programs running simultaneously, to keep the number of possible combinations in the parameter space practical.

With two programs, we have four parameters, that is, the number of workers and the number of CPU cores assigned to each program. We wish to test the performance of each program when run simultaneously for each combination of these parameters, in order to find the optimal configuration. Taking the results of the previous experiment into account, (where we should find the best

parameters for each program in isolation,) we can investigate the performance of the configuration where we use these parameters. The difference between this value and the value given by the optimal configuration will tell us if we can gain any performance from contention aware scheduling, completing the main goal of this project.

So that we can compare our results to the previous experiment, again each thread will have access to any of the cores provided. Both programs will start simultaneously, and we will measure overall performance in terms of throughput, by summing the runtime of each program. Also similar to the previous experiment, we will endeavour to run these experiments across multiple machines.

The size of the parameter space of two programs running simultaneously is $O(N^4)$. This is because both programs have two parameters which can be varied, which are the number of cores to use, and the number of threads to use. Exhaustively testing this space is likely to be unrealistic due to time constraints, therefore, we may need to reduce the number of experiments as time allows. This will be discussed further in chapter 4.

2.3 Additional Experiments

With the above set of experiments, we will have answered the main question posed for this project. That is, can we gain any significant performance by utilising both contention aware scheduling and plastic parallel programming. In this section, we present some additional experiments which should provide some further insight into the system, which were designed and performed after as time allowed.

2.3.1 Extended Contention Experiments

We build upon our contention experiments by performing additional ones with three programs running in contention. This provides an interesting insight to a more complex scenario. However, with three programs, we have six parameters to vary, (the number of threads and the number of cores assigned to each program.) With six parameters, the size of the parameter space is $O(N^6)$, where N is the number of options. Because of this, we have to drastically reduce the number of tests that we perform.

2.3.2 Plastic Experiments

With our contention experiments, will have hopefully demonstrated that there is potential for clear, tangible gain to be had with our system. We intended to further this by demonstrating our system plastically switching implementations, and showing an improvement in the runtime.

To carry this out, we would select one of our contention experiments, and design an experiment where our program starts with a sub-optimal configuration, and then plastically changes its parameters to optimal ones. We would then compare this to the case where we did not plastically change.

We designed and programmed this experiment, however, unfortunately we did not have the time to run it.

Chapter 3

Implementation

In this chapter we summarise the implementation of the system produced last year, to get an idea of how a complete contention aware plastic parallel programming library could be produced. We then go on to cover the implementation details of the synthetic test program produced for this year of the project. Finally, we note a parallel programming pitfall that we encountered, which provides some insight as to why parallel programming is a challenge.

3.1 Last Year's Implementation

In this section, we give an overview of the implementation of the system produced last year. The purpose of this is to get a feel of how a contention aware plastic parallel programming library could be implemented, to provide context for this year's investigation.

We start with our implementation of a basic parallel skeleton, then we cover adding plasticity, and contention aware scheduling. C++ was used for speed, and the parallel backend used was Pthreads due to its wide availability, and fine level of control.

3.1.1 Skeleton Foundation

The skeleton we chose to implement was a map-array skeleton, as the map-array pattern is a widely used pattern with lots of applications. It consists of the common map pattern, where a given operation is applied to all elements of a sequence, with the addition of providing the operation with an additional

array of supplementary data which can be accessed at any time.

This was done in C++ using a templated function which takes another function (supplied by the user) as an argument. The interface of this skeleton is given in figure 3.1, and a usage example in figure 3.2. These figures have been taken from the work done in the previous year [20].

```
template <typename in1, typename in2, typename out>
void map_array(deque<in1>& input1,
               deque<in2>& input2,
               out (*user_function) (in1, deque<in2>),
               deque<out>& output,
               string output_filename = "",
               parameters params = parameters())
```

Figure 3.1: Interface of the map_array skeleton. The first four variables are the two input arrays, the function to apply, and the output array respectively. The output_filename variable is the filename to record the metrics output in, and params sets up the initial parameters we will use. These last two are optional. This figure has been taken from the report of the previous year.

```
int user_function(int in1, deque<int> in2) {
   return in1 + in2[in1];
}

int main() {
   // Inputs.
   deque<int> input1(ARRAY_SIZE);
   deque<int> input2(ARRAY_SIZE_2);

   // Put data in inputs.

   // Output.
   deque<int> output(ARRAY_SIZE);

   // Start mapArray.
   map_array(input1, input2, user_function, output);

   // Record output.
}
```

Figure 3.2: A usage example of map_array, here we apply our user_function to each element of input1. The size of our two input arrays need not match, but the size of the input1 and output arrays must. This figure has been taken from the report of the previous year.

3.1.2 Adding Plasticity

To make this skeleton plastic at runtime, we stop all computation, terminate worker threads, update the parameters, and spawn new worker threads. To facilitate this, we utilised a bag of tasks, to allow for tasks to be put back into the bag, amongst other conveniences including providing common semaphores for inter-thread communication.

3.1.3 Contention Aware Scheduling

In order for all the programs to synchronise and agree upon the parameters to use, we use a separate controller application, from which each program receives its parameters, and subsequent updates. To achieve this, along with a separate controller program, we reuse the original thread which spawns the worker threads to communicate with the main controller and update parameters when necessary. This system is illustrated at a high level in figure 3.4, which expands upon figure 3.3. These figures have been taken from the work done in the previous year [20].

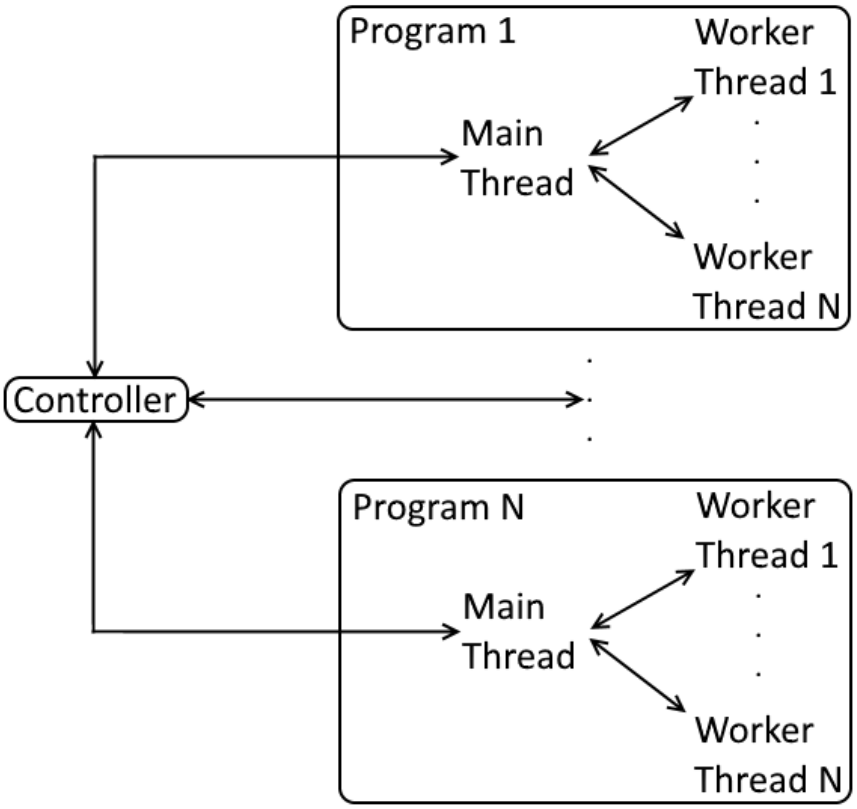

Figure 3.3: High level communication model of the system, with an arbitrary number of programs, with an arbitrary number of threads. Two way communication occurs between the controller and each main thread, and then between each main thread and its worker threads. This figure has been taken from the report of the previous year.

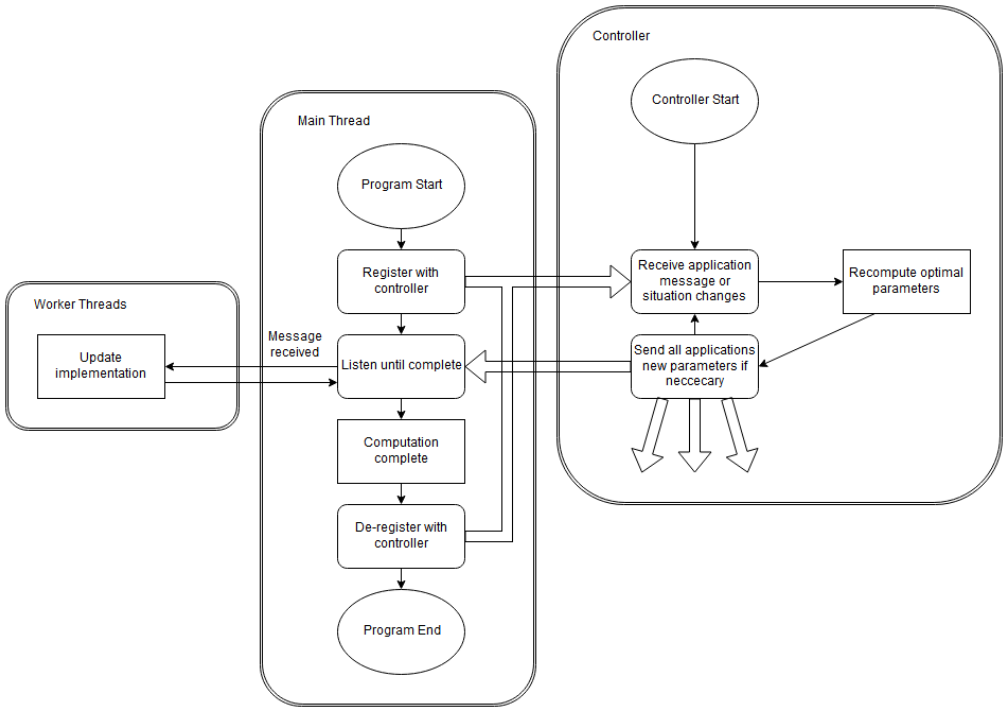

Figure 3.4: Communication model for communications between applications and the controller. Thin lines represent program flow, thick lines represent inter-process messages. This figure has been taken from the report of the previous year.

When this switching is done and what the new parameters should be is another matter. In this project, we demonstrate how a contention aware plastic parallel programming system might be implemented, that the overhead incurred is acceptable, and we show the potential performance gains. A complete system would calculate when to switch parameters and what they would be, based upon the current state of the machine. However, for last year's investigation, we manually control these things in a synthetic environment, and communicate them to the main thread using the controller application, as this was adequate for our investigation.

3.2 Implementation of the Synthetic Test Program

In this section we cover the implementation details of the system we produced this year. The main focus is our synthetic test program, which is designed to perform a test of a stencil code, with given parameters and for a given number of repeats. It does not handle changing parameters for different tests, this is left to our experiment scripts. In the next chapter, chapter 4, we give details of other software that we produced, as well as modifications made to our synthetic test program in the context of specific experiments.

3.2.1 High Level Plan

In contrast to the contention aware plastic parallel programming library that we built last year, this year we do not require a controller application. This is because we are not interested in the overhead incurred with the implementation, but rather with assessing the possible gains that could be had, and overall if a contention aware plastic parallel programming library would be valuable.

This changes our focus, meaning that there will be less work to be done on communication and synchronisation, but more care needs to be taken with regards to parallel performance and producing a suitable synthetic stencil workload, particularly with this more complex parallel pattern.

We intend our synthetic test program to be a part of a larger experimental framework. The outline of how this should operate is as follows:

A complete experiment consists of multiple tests of our synthetic test program with varying parameters. Accordingly, we utilise an experiment script, written in bash, which will handle changing the parameters for each test, and niceties such as reporting the progress of the experiment. It will compile and run our synthetic test program, supplying it with appropriate parameters. The synthetic test program will save its output in an appropriate directory, ready for later analysis with a data analysis program tailored to the current experiment.

Our synthetic test program will have two distinct phases, a setup phase and a processing phase. The high level plan for how our synthetic test program should operate is as follows:

Setup Phase

- Read configuration and set appropriate parameters

- Initialise the data grid with given size
- Calculate row allocations to divide said grid between the given number of threads
- Setup data structures for additional features, such arrays for the convergence test, and semaphores for barrier synchronisation (These features are discussed in section 2.2.2.)

Processing Phase

- Setup timekeeping functions
- Spawn worker threads
- Perform computation of the stencil code, recording the time taken
- Format and save output data for later analysis

3.2.2 Implementation

In this section we give the specifics of the implementation of our synthetic test program. We start by describing some general features of the program, then we break it down into the setup phase and the processing phase, as described in the high level plan, in section 3.2.1.

The synthetic test program is implemented in C++. We chose C++ for its speed, and advanced language features such as templates. The parallel code was implemented using the C++ std::threads library, which provides an improved interface using pthreads as its back end.

Many aspects of the program are tuned for performance, such as minimising function calls the use of appropriate data structures. Also in the name of performance, and of correctness, for features which are switched on and off between tests in an experiment, (such as barrier synchronisation,) we use the C++ preprocessor to include or remove the actual code at compile time, according to compiler flags. This is so that, if a feature is switched off, its code is not even included in the binary file, meaning it has no effect on our performance. An example of how this is done can be found in appendix A, which shows the preprocessor commands and how they switch features in our worker function.

Another consideration which can affect our performance is a cold CPU cache. The CPU cache is a structure which stores data for faster lookup, and is particularly relevant to multiprogramming workloads, as we may encounter synchronisation issues [1]. However, When we first start our program, the cache will be

cold, as it contains no useful data. So our first run by our synthetic test program will be affected by the cold cache, as it loads data into the cache. Subsequent runs would not be affected, as the cache would have "warmed up". To combat this, we perform one extra run at the start of our computation, and we disregard its execution time when we are analysing the data from the experiment.

Setup Phase

The first thing to be done by our synthetic test program is to read the configuration given. This is provided as a set of parameters in a text file, which control each aspect of the test. In short, these aspects are: the number of repeats, the size of the data grid, the number of iterations (passes) over said grid, the computation to perform at each point, the number of workers to use and which virtual CPU core(s) they should use. This covers the features we described in section 2.2.2. The exact details of the configuration file are given in section 4.2.

Reading this configuration file was done using a custom implementation, rather than utilising an existing library, for precise control over the configuration file format.

Next, we initialise the data grid, which uses double precision floating point values, as this is a common format in scientific computing. So much so that work has been done to facilitate its use for scientific computing [27].

Along with the data grid, we must initialise additional grids for the worker threads to store intermediate values between iterations. Two arrays are used, one to store the values from the previous iteration, and one to contain newly computed values. After an iteration, the roles of these arrays are reversed. This technique of using two arrays, named double buffering, is done to minimise the necessary synchronisation overhead, and is a common technique in performance conscious applications [10, 31].

In the final part of the setup phase, we divide the data grid to allocate between the given number of threads, and setup our other data structures.

Processing Phase

At the start of the processing phase, we record the current time. We go on to start the given number of worker threads, which will perform the main computation (the code for these worker threads can be seen in appendix A.)

The worker threads then iterate over the grid, performing the basic jacobi computation at each point (that is, compute and record the mean of the points neighbours.) However, we wish to add an additional computation at each point,

one that we can control, which can stress different resources and perform a set amount of work. This workload must be predictable (for repeatability,) scalable (for varying performance characteristics,) and efficient with regards to parallelism (for validity.) This is the main section of the processing phase, and is an important part of our synthetic test program. The rest of this section is dedicated to how we went about creating these workloads.

We want a set of workloads with different performance characteristics, which stress different components. As discussed in 2.2.3, this should make for valuable test cases. Accordingly, we start with workloads which stress different aspects of a system. These include the CPU, virtual memory, hard disk, and low-level I/O. We chose to focus on the most common performance critical resources, the CPU and the memory. We looked at existing parallel performance tests to inform our decisions [17, 36].

Again, as discussed in section 2.2.3, we target two scenarios for each synthetic stencil workload. One with a small amount of work, meaning the 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. We can do this simply by changing the grid size, but for more fine grained control, we also wish to vary the complexity of the calculations performed at each point in the grid.

To achieve this, all of our synthetic stencil workloads consist of some sort of work, contained in a loop. This loop serves two purposes, it beefs up our workload functions such that they can have a significant effect on runtime, and they provide linear control over the weight of our workload, allowing us to vary it for different experiments.

For example, one of our final workloads designed to stress the CPU consisted of a loop performing one square root operation, using the loop iterator as input. The square root function was used as it is a commonly used, reasonably expensive operation. This is the workload we use to describe the issues we had with compiler optimisation in the next section.

3.3 A Parallel Programming Pitfall

In this section, we present an aside describing a problem encountered during the development of the software required for this project. It illustrates the challenges of parallel programming.

Our workload is synthetic with no actual effect. As such, it is vulnerable to compiler optimisation. Since we are carrying out performance critical experiments, we compile our application at GCC optimisation level 3, which can be quite

smart in removing redundant loops amongst other optimisations. Our workload entirely consists of redundant loops, and so was consistently optimised away, as shown in figure 3.5. As it transpires, this can be a particularly tricky problem to overcome.

The first attempt at a fix was to vary the input to sqrt function. If done in an unpredictable way, the compiler wouldn't be able to optimise our workload away. To do this, we thought of storing a random sequence of numbers, and iterating over this to provide input for sqrt. However, this would introduce more memory reads, slowing our computation and corrupting our workload from a purely CPU dependant workload. Also, the compiler may have still optimised away our workload, since we are using precomputed inputs, and we do nothing with the resulting value.

Our second idea was to generate random numbers to use as input to the workload. Moving the random number generation into the workload lets us circumvent any memory use. We can use rand() with a constant seed, since we are not concerned with true randomness, and we want repeatable results.

In validating this, we found that the workload was not being optimised out, and that we could vary the workload and the time taken would vary accordingly. However, with a more detailed analysis, we came across some confusing results, presented in figure 3.6.

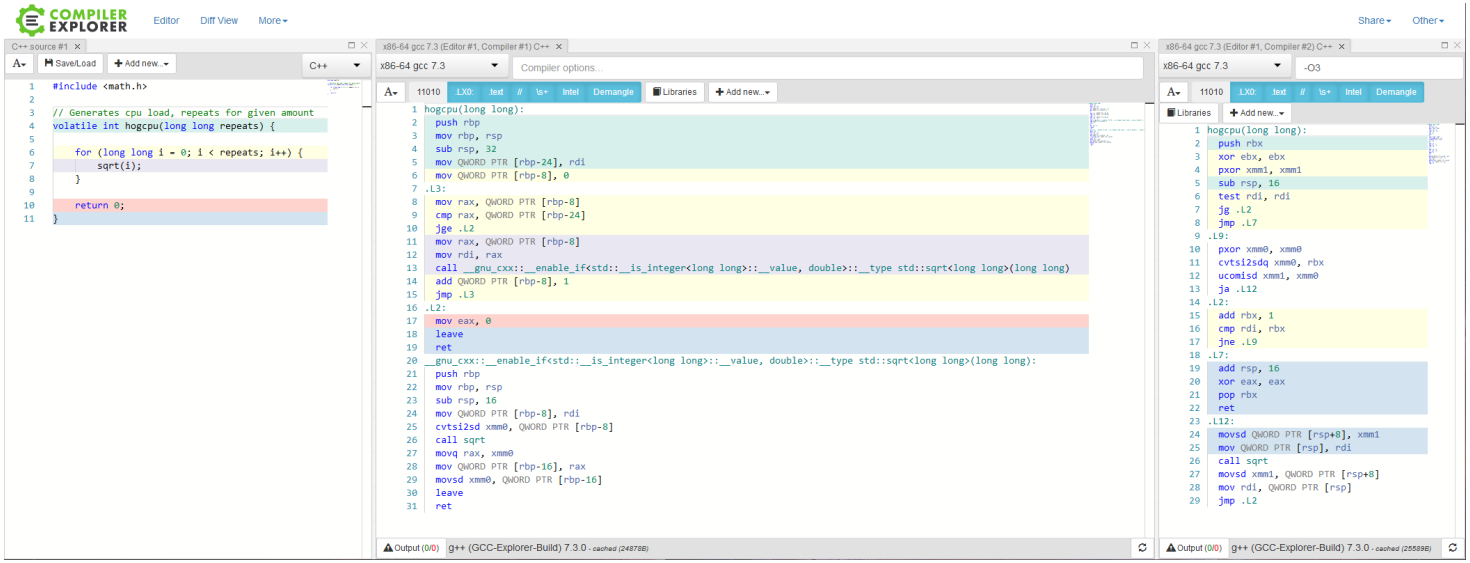

Figure 3.5: This figure shows that our workload function (sqrt()) is optimised away at optimisation level 3. Note - Colour coded regions show where the C++ code matches the machine code.
Left: Original C++ code, Middle: Machine code produced by GCC at optimisation level 0, Right: Machine code produced by GCC at optimisation level 3
This tool [16] proved to be useful during diagnosis of this problem.

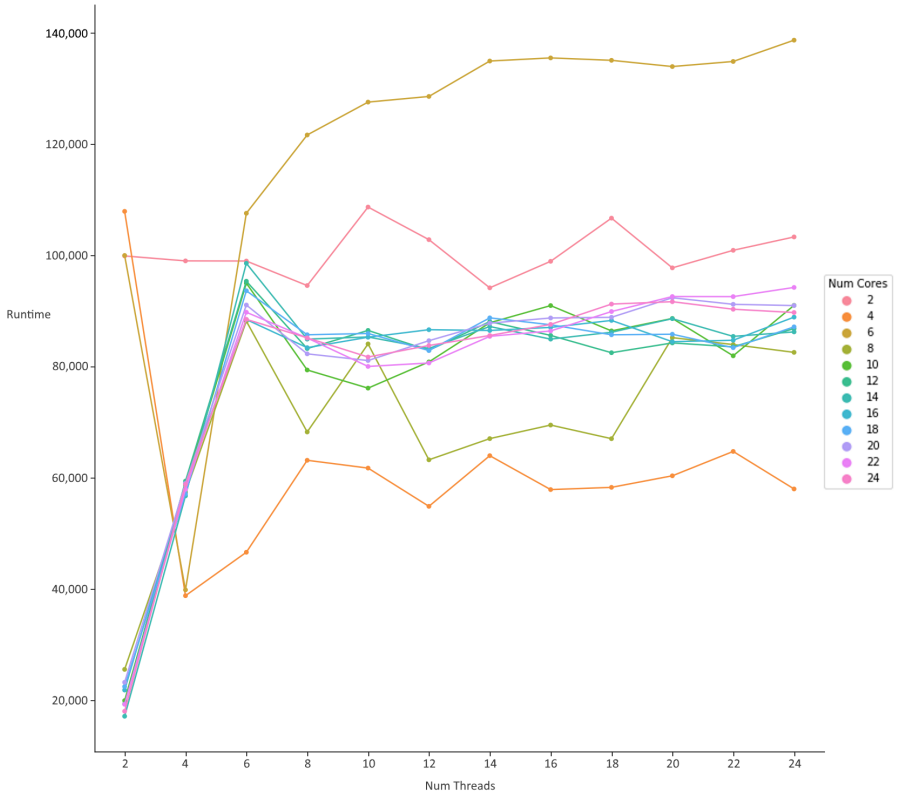

Figure 3.6: In this experiment, we provided the program with up to N CPU cores, and used up to N threads. Each number of threads was evaluated with each number of CPU cores in steps of 2. CPU core numbers from 1-12 are real cores, and 13-24 are hyperthreads.

As figure 3.6 shows, in the case where we utilise two cores (the pink line,) increasing the number of threads above two if anything degrades performance. This is as expected. Moving up to the case with four cores (the orange line,) we see a similar story, going above four threads degrades performance. However, for six cores (the yellow line,) we see something different. Using six threads results in drastically degraded performance. In general, we can see using six threads or more always hurts performance. This might still be expected, but we see that performance doesn't vary much above six threads regardless of how many cores are provided. We also see that using two workers with as many cores as possible is the optimal solution. Both of these facts are inconsistent with what we expect to see.

These results were produced quickly, using no repeats, as an initial investigation. So at first, we reasoned that this could be the result of cold cache issues,

resulting in these strange runtimes. We reproduced the same experiment, using ten consecutive repeats. This produced the following graph in figure 3.7:

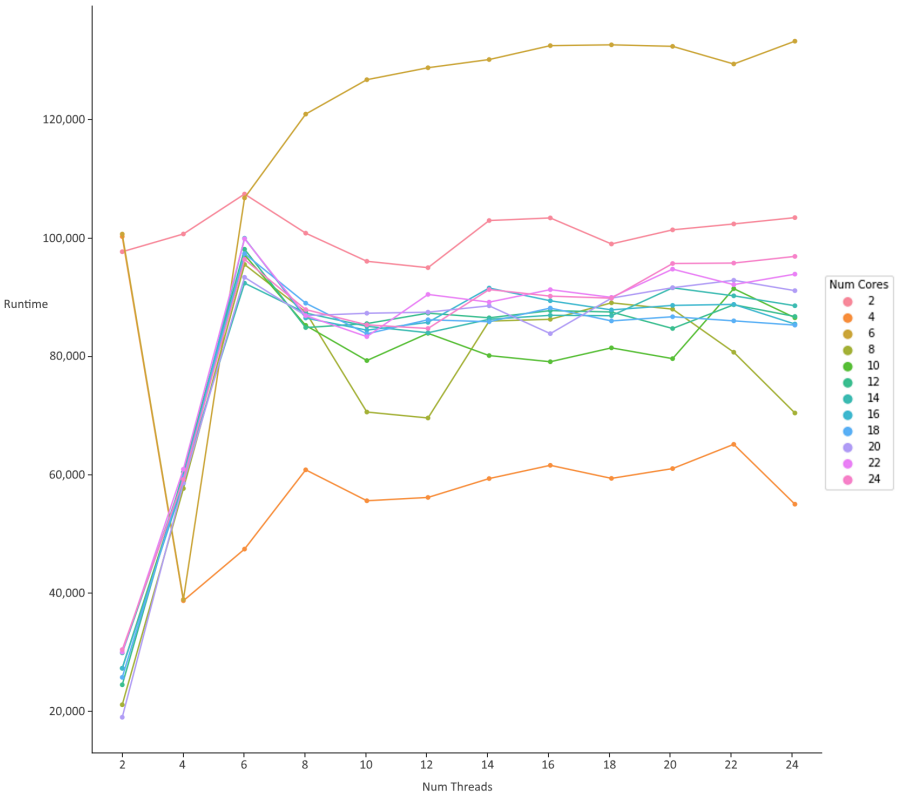

Figure 3.7: As the previous experiment, only averaging the results over 10 repeated runs.

Again, a similar story to the previous graph, with similar unexpected features. This rules out cache issues.

After further diagnosis, we discovered some other strange behaviour. Running the computation with one thread resulted in times of 500ms, and with two threads we saw 20,000ms. Not what we would expect at all! Even stranger, when using multiple threads pinned to the same CPU we again saw runtimes in the region of 500ms.

Narrowing it down further, we changed the workload function from sqrt(rand()) to sin(rand()). This produced similar behaviour, so either the sqrt() and the sin() functions have the same issue, or the problem is the rand() function. The most likely culprit seemed to be the rand() function, so another method of

preventing compiler optimisation was needed. This time, we tried using inline assembly code to provide the workload, so we can know exactly what is going on. An example of this is given in figure A.3 in appendix A. But this still required a function wrapper, and preallocated memory.

It was after this that we discovered the volatile keyword. Quoting from the C++ Standard (7.1.5.1) [33]:

"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."

Declaring the entire workload function as volatile successfully prevented the compiler from optimising it out, and solved our problem.

Upon further diagnosis, we found that the rand() function is not thread safe! This is consistent with the observed behaviour, since just one thread finished much faster, and many threads pinned to one core similarly fast. This was because, with all the threads running on the same core, they would never be running simultaneously, so calls to rand() would never interfere with each other. Using a non-thread safe function in this way is an easy to make, hard to diagnose error, and makes a good example of why parallel programming is hard.

Chapter 4

Experimental Methodology

In this chapter, we discuss our overarching experimental methodology, as well as the structure and details of each experiment. We start by providing the test machine details, then we describe the software that was created to evaluate our synthetic test program discussed in section 3.2. Next, we cover experiment specific details, along with any modifications made to our synthetic test program to facilitate them. Finally, we cover the software produced to analyse the data we collect.

4.1 Machine Details

We perform experiments on two machines, named spa and XXXII, in order to compare and explore performance characteristics across different architectures. Here we give the details of each machine. The main difference we are concerned with is the number of physical and virtual CPU cores each machine possesses, although CPU and RAM speeds will also affect our experiments.

spa:

- GNU/Linux kernel #1 SMP PREEMPT Fri Jul 15 12:46:41 UTC 2016 (84ae57e), release 4.1.27-27-default
- x_86_64 Instruction set
- 2 x Intel(R) Xeon(R) CPU E5-2620 @ 2.00 GHz, giving a total of 12 physical CPU cores, 24 virtual CPU cores
- 16GB RAM

- GCC version 4.8.5 (SUSE Linux), Thread model: posix

XXXII:

- GNU/Linux kernel #1 SMP Wed Sep 13 07:07:43 UTC 2017 (3e35b20), release 4.4.87-18.29-default
- x_86_64 Instruction set
- 4 x Intel(R) Xeon(R) CPU L7555 @ 1.87 GHz, giving a total of 32 physical CPU cores, 64 virtual CPU cores
- 64GB RAM
- GCC version 4.8.5 (SUSE Linux), Thread model: posix

Both of these machines utilise intel chips, which provide some form of hyper-threading. Hyperthreading is Intel's proprietary simultaneous multithreading (SMT) implementation, which is used to improve parallelisation of computations. For each physical CPU core, the operating system addresses two virtual CPU cores, sharing the workload when possible. This allows concurrent scheduling of two processes per core. In addition, two or more processes can use the same resources: if resources for one process are not available, then another process can continue if its resources are available.

The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel. Note that this does not directly add more computing power, but rather, facilitates the use of the existing computing power.

4.2 Experiment Infrastructure

In this section, we discuss the general parameters for our experiments, as well as the software produced in order to evaluate our synthetic test program with various parameters.

Our synthetic test program takes a configuration file which specifies the following parameters of a test run:

num_runs: The number of times to repeat the experiment. 101 was used for all experiments, 1 run to warm up the cache and 100 for data gathering.

num_stages: The number of stages in execution. This lets us plastically change parameters, as we can specify different parameters for each stage. Each field ending with _n can be specified per stage, with _0 for the first, _1 for the second, and so on.

grid_size: The size of the grid to use for computation. The data grid is a square grid comprised of doubles, and is initialised with the edges set to 1.0, and interior cells set to 0.0. Typical values range from 32-32768 in experiments.

kernels: Which computation kernels to use. Several can be specified, and they will be run one after another according to the durations/repeats specified. Options are: cpu, vm, hdd, io.

num_iterations_n: The number of iterations (passes over the data grid) to compute. Typically, this would vary from problem to problem, but for repeatability and comparison purposes we directly control this. Values range from 1-1000 in experiments.

set_pin_bool_n: Controls which virtual CPU cores each thread can utilise. Values are as follows:

- 0: Each thread can access any virtual CPU core.
- 1: Each thread is pinned to the virtual CPU core with the same ID (effectively one virtual CPU core per thread.)
- 2: Each thread can access the virtual CPU cores allocated to it by the pinnings_n field, allows precise control.

kernel_durations_n: Specifies the duration to execute each computation kernel, in milliseconds. This allows us to generate a workload which will run for a guaranteed amount of time. In practice, this was not used for experiments, for repeatability reasons. Instead, we used the kernel_repeats_n parameter.

kernel_repeats_n: Specifies how many times to repeat each computation kernel. For obvious reasons, we can only use this or kernel_durations_n, not both. Typical values range from 10-1000 to produce a substantial workload.

num_workers_n: The number of worker threads to use. Values range from 2-48 in experiments.

pinnings_n: Specifies the exact virtual CPU cores to which each thread is to be pinned (restricted.) Example value: "1..4,8 1..4,8 5..6 7" restricts thread 0 and 1 to virtual CPU cores 1 through 4 plus 8, thread 2 to 5 and 6, and thread 3 to 7.

This allows us to precisely control each aspect of a test run, and repeat it the desired number of times. However, in order to programmatically test different configurations, we must construct some additional software to execute our synthetic test program with the appropriate settings for our experiments.

To this effect, we created a bash script for each experiment to be carried out. Each script compiles the synthetic test program with the appropriate flags (turning on/off features such as the convergence test or different barrier implementations,) generates the range of config files required, and executes the synthetic test program according to the experiment requirements, logging each action as it goes. For some experiments, multiple instances of the synthetic test program are required to run simultaneously, requiring synchronisation, and separate config and logging files.

Other features of these bash scripts include a completion percentage display, machine specific settings, and a notification of when experiments are completed, and how long they took. The completion percentage is useful as these are shared systems, and writing the percentage complete to the MOTD of the machine allows other users to gauge when the machine will be free. The machine specific settings allows us to execute each experiment on multiple machines with the same scripts. An example of such a script is included in appendix B.

Before starting each experiment, we ensured that no other process was running which would not be part of the ordinary background of processes present in a system. We did this manually, using top to view the running processes. These experiments were performed on shared machines, making the MOTD update feature of our scripts useful, as it warns other users that experiments are going on, and should not be disturbed.

4.3 Experiment Specific Details

In this section, we discuss experimental methodology pertinent to specific experiments, as well as the modifications made to our synthetic test program in order to facilitate them.

4.3.1 Performance Characteristics Investigation

Barrier Synchronisation

In order to compare different barrier options, they must be implemented in the synthetic test program. These options are: no barrier, a custom reusable counter barrier, and a pthreads barrier. Each of these is activated by defining flags used in the preprocessor. This lets us choose which (if any) barrier implementation to use at compile time without so much as an extraneous if statement, which preservers realistic performance. This technique was used for other features for different experiments. These include process synchronisation, convergence test execution, and compute kernel execution.

Convergence Checks

In order to perform convergence checks, we must add the option of their use to our synthetic test program in a reliable and repeatable manner. In a typical stencil code application, a convergence check is run after each pass/iteration over the data grid, and the computation is halted once the convergence test is positive. This results in the program running for an unspecified number of iterations, which is dependent upon the exact problem instance being computed.

For our synthetic test application, we want our computation to run for a predictable number of iterations. This is for repeatability, and to facilitate fair comparisons between configurations, which otherwise might have a different number of passes over the grid. The way that this is achieved is by modifying our synthetic test program such that it runs for a given number of passes/iterations over the grid, still performing the convergence test, but ignoring the result.

Similar to the barrier synchronisation features, the convergence checks are controlled by defining flags used in the preprocessor.

4.3.2 Finding Interesting Instances

In order to find some interesting instances which stress different resources, we need some different elements of computation which stress different resources. To facilitate this, we added many examples of computation kernels to our synthetic test program, which stressed different resources (e.g. CPU, RAM,) using different methods (e.g. repeated use of sqrt() function, sin() function, or malloc() function.)

Many experiments were performed, with few repeats, in the initial search of the "performance space". Once we started to come close to the performance characteristics we desired, we took more care, and performed experiments with the usual 101 repeats, in order to properly dial in the best values we could (these results are presented later, in section 5.2.) This was done in the name of speed, as we have multiple parameters which affect the performance characteristics (namely grid size, num iterations, kernel, and kernel repeats.)

4.3.3 Optimal Threads Experiments

Recall, as described in section 2.2.4, the optimal threads experiments are designed to evaluate the performance characteristics of each interesting instance, running in isolation. Since we have our interesting instances, all that is required

is a more time consuming and thorough investigation, and as such, this does not require modifications to our synthetic test program.

For this thorough test, we aim to evaluate every possible number of threads for every possible number of virtual CPU cores. This would give us the optimal number of threads to use, for any given number of virtual CPU cores. However, after estimating the runtime of these experiments, to keep them manageable, we increment the number of threads and the number of cores in steps of two. This resulted in a total experimental programme runtime of approximately a week per machine.

4.3.4 Contention Experiments

As described in section 2.2.5, the purpose of the contention experiments is to evaluate the performance characteristics of pairs of instances, when run in contention. This requires executing our instances simultaneously. However, as described in section 3.2.1, our synthetic test program has a distinct start up phase, and a distinct processing phase, in which we gather our data. So in order to properly test our instances with contention, we must synchronise across them, such that they both enter the processing phase simultaneously. If this is not done, as our programs perform repeats, the processing phases would drastically drift out of synchronisation, as in many circumstances, one program finishes well before the other.

Adding this cross process synchronisation is the most significant modification to our synthetic test program. To implement it, we used named POSIX semaphores, to minimise extraneous library use, and preserve portability across machines. To prevent interference between distinct runs, we use the run number as the name of the semaphore.

Again, as in the optimal threads experiments, we must cut down the number of experiments to perform. For the contention experiments, the size of the parameter space of two programs running simultaneously is $O(N^4)$. This is because both programs have two parameters which can be varied, which are the number of cores to use, and the number of threads to use. Exhaustively testing this space is unrealistic due to time constraints, therefore, we incremented each parameter in steps of four. This kept the total experimental programme runtime to between one and two weeks on spa.

4.4 Additional Experiments

In this section, we give the experimental methodology for some additional experiments which should provide some further insight into the system. We performed some extended contention experiments, however due to time constrains, we could not perform any plastic experiments.

4.4.1 Extended Contention Experiments

We would like to take our contention experiments further, by investigating situations involving three programs. This would provide an interesting insight into more complex situations. Similar to our previous contention experiments, we would need to synchronise all three programs in the start up phase, and prevent interference between runs.

Another similarity to the previous contention experiments is that we will need to again cut down on the complexity of our experiments. With three programs, we have six variables, giving us a parameter space of $O(N^6)$. More than any other experiment, we must take care to ensure our experiments complete in reasonable time. It transpires that, to achieve this, we incremented each parameter in steps of four, with a limit of twelve. The runtime of these experiments ranged from 5 hours to 5 days on both spa and XXXII.

4.4.2 Plastic Experiments

The intention for these plastic experiments is to demonstrate our synthetic test program plastically changing from sub-optimal to optimal parameters. This involves utilising the plastic capabilities of our synthetic test program, in which we have multiple stages. We start the first stage with the sub-optimal parameters, and then at some point, we move into the second stage, where we use the optimal parameters. We designed and programmed this experiment, however, unfortunately we did not have the time to execute it.

4.5 Data Analysis

In this section we present the software created to analyse the data produced by our experiments.

For each experiment, we produced a corresponding Jupyter Notebook document [21], which loaded and processed the results obtained. Quoting from the Jupyter

Notebook documentation: "Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis" [18].

We chose to use Jupyter Notebooks with python as the programming language, as the interactive nature of the notebooks as well as the sophisticated scientific computing packages provided by python allow for complex data analysis.

The steps we take in each Jupyter Notebook document are as follows:

1. Select the experiment parameters, such as which machine the data is from. This affects how we label the data, and how we present the data.
2. Import and label the raw data from the target location.
3. Process the data, computing statistics such as the mean or the variance of each data point.
4. Present the data as a graph, and save a copy.

As discussed in section 3.2.2, we discard the first result due to cold cache issues. We then take the mean over one hundered points, and compute 95% confidence intervals for our graphs.

An example of a notebook we used is included in C.

Chapter 5

Results

In this chapter, we discuss the results obtained from the experiments performed using our synthetic test program.

Each section provides the results of the experiments described in the corresponding section in chapter 4, which describes each experiment.

5.1 Performance Characteristics Investigation

In this section we discuss the results obtained by the performance characteristics investigation. As such, two experiments were performed, each assessing different aspects of the simple synthetic Jacobi stencil code. The first deals with how barrier synchronisation, required for the program to be correct, affects performance, and the second deals with how convergence checks affect performance.

5.1.1 Barrier Synchronisation

In this experiment, we evaluated three options of barriers for the synthetic test program. One using a custom implementation, one using the pthreads implementation, and one using no barrier at all. Without a barrier the program is incorrect, however we include it to get a feel for the performance impact of the barrier, as we expect it to be significant. The results of this can be seen in figure 5.1.

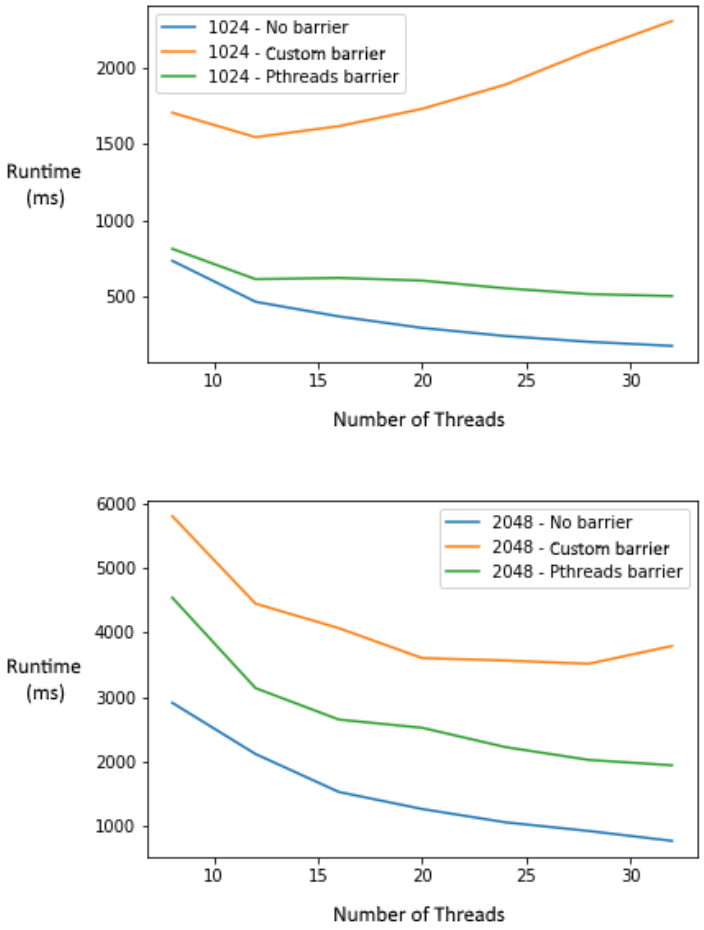

Figure 5.1: A set of graphs which show the runtimes of different barrier options in milliseconds against the number of threads used. The first graph is at grid size 1024, the second at 2048. Run on XXXII, each point is the mean of 100 repeats, with insignificant variance. Computation at each point is the basic jacobi stencil.

num_runs:	101
num_stages:	1
grid_size:	1024
num_iterations_0:	500
set_pin_bool_0:	0
pinnings_0:	-
kernel_repeats_0:	-
kernel_durations_0:	-
kernels:	-
num_workers_0:	8

Figure 5.2: The base config settings used for the barrier synchronisation experiment, the results of which are given in figure 5.1. Certain settings will change for different tests.

From figure 5.1, we can see that using any kind of barrier adds a consistent performance penalty. Also, that our custom barrier implementation performs significantly worse compared to the pthreads implementation. This provides another example of the difficulties of parallel programming, as the custom barrier is an implementation of a standard counter barrier algorithm, whereas the pthreads implementation has had the benefit of rigorous performance tuning over many years.

Another takeaway from these results is that at smaller grid sizes, and with only the basic jacobi work to do (no additional workload,) the barrier synchronisation has a more significant effect on the runtime. Indeed, looking at the case with a grid size of 1024 and 32 threads, the runtime can increase by a factor of 2-8x, depending on the barrier implementation.

After this experiment, we used the pthreads barrier implementation, as this is the version which would be selected by a performance conscience application programmer.

5.1.2 Convergence Checks

In this experiment, we evaluate how the stencil convergence checks affect the performance of the synthetic test program. To this effect, we performed tests at multiple large grid sizes, with and without the convergence checks. All experiments unless otherwise stated include simulated convergence checks, more details can be found in section 4.3.1. The results of these experiments can be seen in figure 5.3.

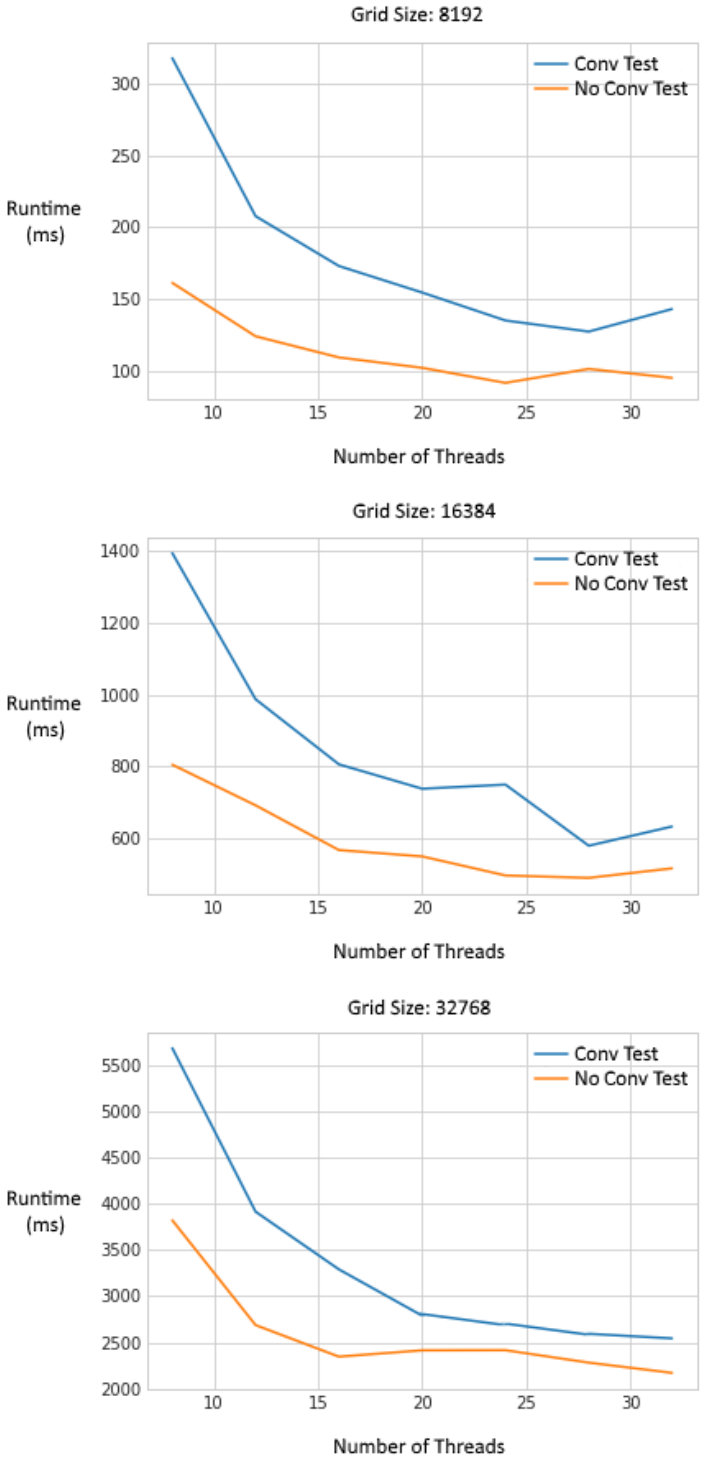

Figure 5.3: A set of graphs which demonstrate how the convergence test affects the performance of our synthetic test program. Each graph plots the runtime, in milliseconds, of our synthetic test program, with and without the convergence test, against the number of threads used. The three graphs show the case with a grid size of 8192, 16384, and 32768 respectively. Run on XXXII, each point is the mean of 100 repeats, with insignificant variance. Computation at each point is the basic jacobi stencil.

num_runs:	101
num_stages:	1
grid_size:	1024
num_iterations_0:	10
set_pin_bool_0:	0
pinnings_0:	-
kernel_repeats_0:	-
kernel_durations_0:	-
kernels:	-
num_workers_0:	8

Figure 5.4: The base config settings used for the convergence checks experiment, the results of which are given in figure 5.3. Certain settings will change for different tests.

The results presented in figure 5.3 are generally what we would expect. We see that at all grid sizes, the convergence test has a scaling cost, relative to the number of threads. This makes sense as the convergence test can be efficiently partitioned between threads.

We can also see that the convergence test has a significant cost, sometimes doubling the runtime of our synthetic test program. This is using the basic jacobi kernel of summing each point's neighbours however, and as we use more complex kernels, the relative cost will go down.

Additionally, these results show the effect doubling the grid length (with a square grid, so four times as much data.) The runtime seems to scale roughly linearly with regards to the amount of data in the grid, for equivalent numbers of threads, with and without the convergence test. We can see that, in the case with eight threads and no convergence test, at grid size 8192 we have a runtime of approximately 160ms. After quadrupling the amount of data (at gridsize 16384,) we have a runtime of approximately 800ms. After quadrupling again, we have a runtime of approximately 3800ms. These characteristics may change however when we use less predictable workloads.

5.2 Finding Interesting Instances

The purpose of these experiments is to find interesting instances, from a performance perspective, of the basic jacobi stencil. This involves adding a weightier computation kernel to be run at each point in the grid, in addition to the basic jacobi stencil. We would like to identify two or three different types of kernel, which stress different resources of the system.

For each of these, as discussed in section 2.2.3, we want to find one example where adding more threads harms the overall performance, and one example where the program can use as many threads as we can give it. To find such examples, we performed a measured exploration of the "performance space" of each kernel, the details of which are as follows:

5.2.1 A Range of Computation Kernels

Our first step was to come up with some different possible kernels. We want kernels which stress different resources, so we decided to create kernels which stress the CPU and the virtual memory. Other resources could also be targeted (e.g. I/O or HDD,) but to keep the number of experiments manageable, we focused on the typically most important resources.

From there, we have multiple options to create computation kernels, using different functions and methods. We compared the performance characteristics for a variety of kernels, and how they performed with different numbers of threads, and at different grid sizes. The results can be seen in figure 5.5.

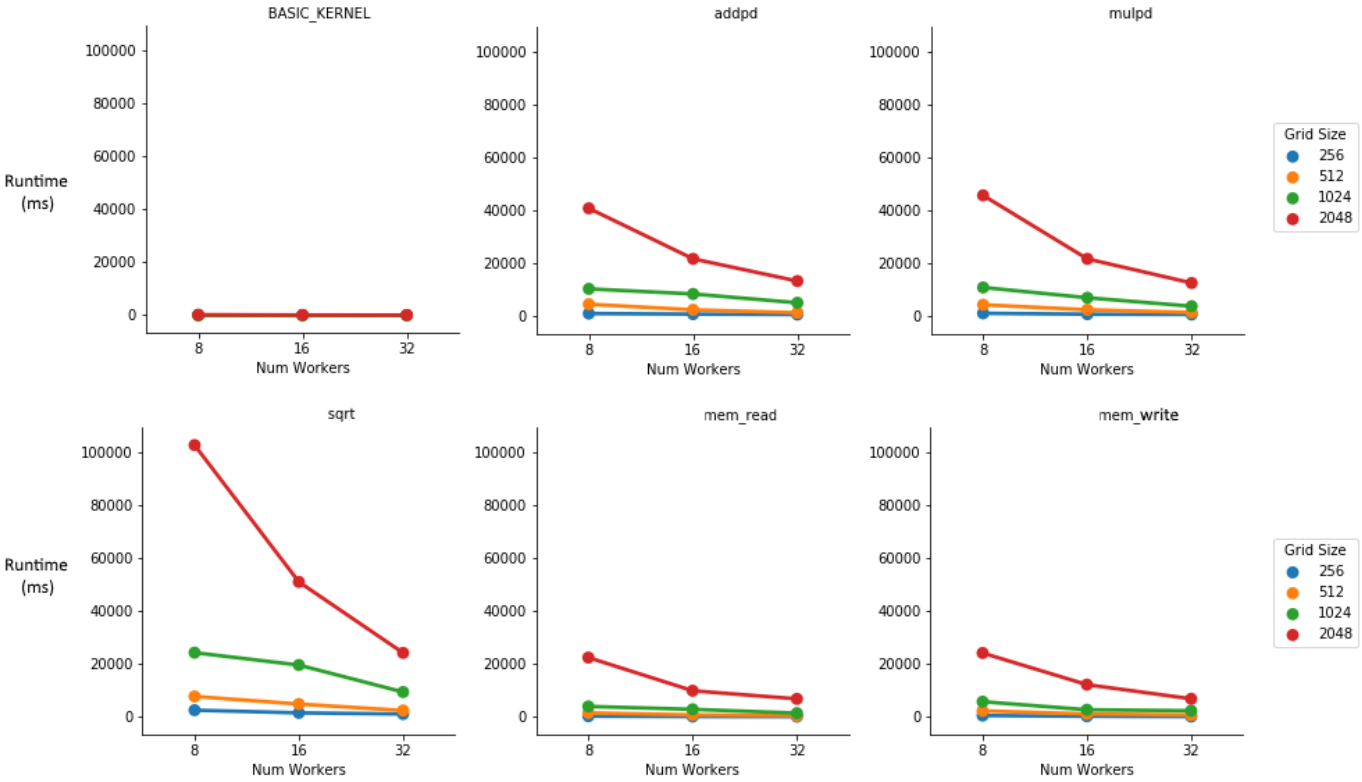

Figure 5.5: These graphs show the runtimes, in milliseconds, of various computation kernels with various grid sizes, against the number of threads used. Run on XXXII, each point is the mean of 10 repeats, with insignificant variance. We have fewer repeats with these graphs, as these graphs were part of a range whose only purpose was to let us get a feel for the performance characteristics of different kernels, such that we can select appropriate kernels for subsequent experiments. The kernels are as follows:

- **BASIC_KERNEL:** This kernel has no additional computation, it is simply the basic jacobi kernel. We include it here for comparison.
- **addpd:** This kernel is designed to stress the CPU using repeated addition operations. Specifically, with the x86 operation ADDPD: Add Packed Double-Precision Floating-Point Values.
- **mulpd:** This kernel is designed to stress the CPU using repeated multiplication operations. Specifically, with the x86 operation MULPD: Multiply Packed Double-Precision Floating-Point Values.
- **sqrt:** This kernel is designed to stress the CPU using repeated sqrt operations. Specifically, using the sqrt() function from the standard math library.
- **mem_read:** This kernel is designed to stress the VM using repeated memory operations. Specifically, we allocate a portion of memory, and then we step through it, touching (reading) certain values.
- **mem_write:** This kernel is designed to stress the VM using repeated memory operations. Specifically, we allocate a portion of memory, and then we step through it, writing certain values.

num_runs:	11
num_stages:	1
grid_size:	256
num_iterations_0:	10
set_pin_bool_0:	0
pinnings_0:	-
kernel_repeats_0:	1000
kernel_durations_0:	-
kernels:	addpd
num_workers_0:	8

Figure 5.6: The base config settings for the experiments used to compare computation kernels, the results of which are given in figure 5.5. Certain settings will change for different tests.

These results were good, and we see broadly what we expected. This was that at larger grid sizes, adding more threads should significantly reduce the runtime. Then, at smaller grid sizes, more threads should not help as much, or even increase the runtime. This can be seen best with the sqrt kernel, where at grid size 2048, we see significant gains by adding more threads, whereas at grid size 256, we see next to no gain.

This behaviour can also be seen with kernels which stress the virtual memory of the system, which is again, as intended.

The basic kernel was included to show the characteristics of the jacobi pattern with no additional computation, i.e. at each point we touch the neighbouring values and that is all. From this graph we see that a potential program would need a huge grid size to see any tangible benefit from adding threads.

Based on this analysis, we selected the following kernels for our subsequent experiments:

- **sqrt** because it involves common and reasonably expensive operations (loops, shifts, multiplication.)
- **mem_read** because it emulates a program which must read additional data before processing it.

Now that we have two kernels which stress different resources, we want to select two different instances of each: one where adding more threads at some point is detrimental, and one which can fully saturate our machine without reaching that point.

5.2.2 Selecting Interesting Gridsizes

With our selected kernels, we want to find instances of each where adding threads is always beneficial, or sometimes detrimental. We already have a case where adding threads is always beneficial for both (grid size 2048 from figure 5.5,) so to find a clear case where adding threads is detrimental, we experiment with smaller grid sizes, the results of which can be seen in figure 5.7.

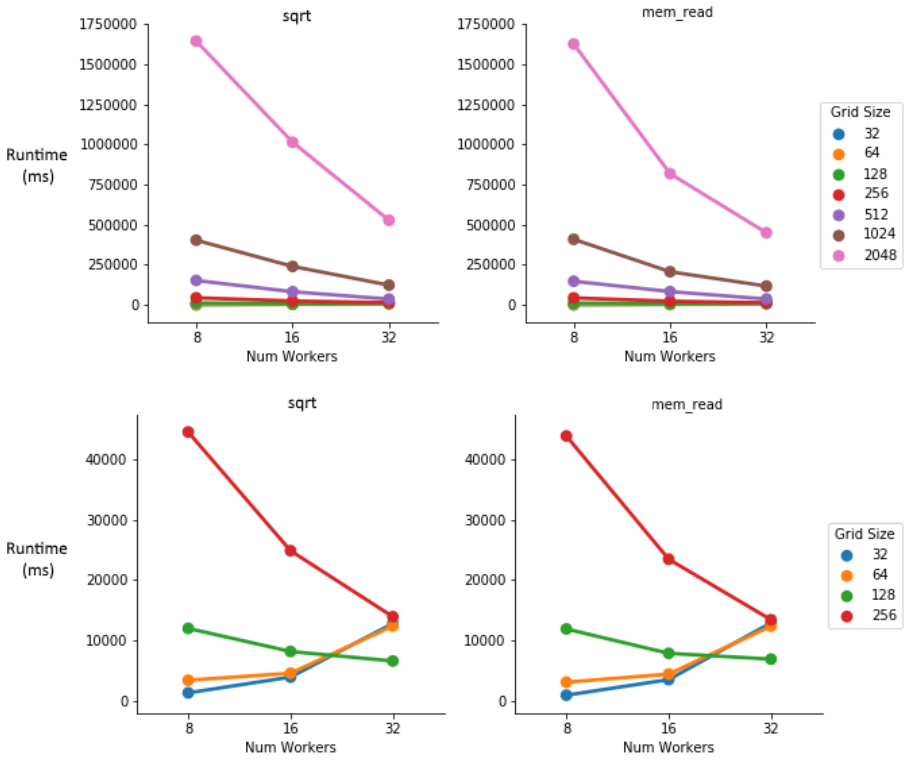

Figure 5.7: These graphs show the runtimes, in milliseconds, of various computation kernels with various grid sizes, against the number of threads used. Run on XXXII, each point is the mean of 10 repeats, with insignificant variance. We have fewer repeats with these graphs, as these graphs were part of a range whose only purpose was to let us get a feel for the performance characteristics of different kernels, such that we can select appropriate kernels for subsequent experiments.

num_runs:	11
num_stages:	1
grid_size:	32
num_iterations_0:	5000
set_pin_bool_0:	0
pinnings_0:	-
kernel_repeats_0:	500
kernel_durations_0:	-
kernels:	sqrt
num_workers_0:	8

Figure 5.8: The base config settings for the experiments used to select interesting grid sizes, the results of which are given in figure 5.7. Certain settings will change for different tests.

As we reduce the gridsize, for both types of kernel we see a turning point around grid size 128, where increasing the thread count results in a moderate gain in performance, but with the next grid size down we lose a significant amount of performance. These are good results, as they give clear examples of cases where adding threads becomes detrimental to performance.

However, we would also like to see if a similar effect could be had by changing the amount of computation done at each point in the grid, rather than the number of points in the grid. Both increase the total amount of work to be done, so we should see similar results. Also, this would allow more fine grained tuning of our instances.

5.2.3 Selecting Interesting Kernel Complexities

To make our kernels more complex, we simply repeat their computational core a given number of times. This gives us a linear control on the amount of work that is done at each point in the grid. Each kernel was tested with multiple grid sizes and different numbers of repeats. Varying the repeats should make our understanding of the differences between instances more fine grained, and thus, allow us to identify the point at which, as the grid size/number of repeats shrinks, increasing the number of threads would be sub-optimal. An example of the results for the sqrt (CPU stressing) kernel on XXXII is presented in figure 5.9.

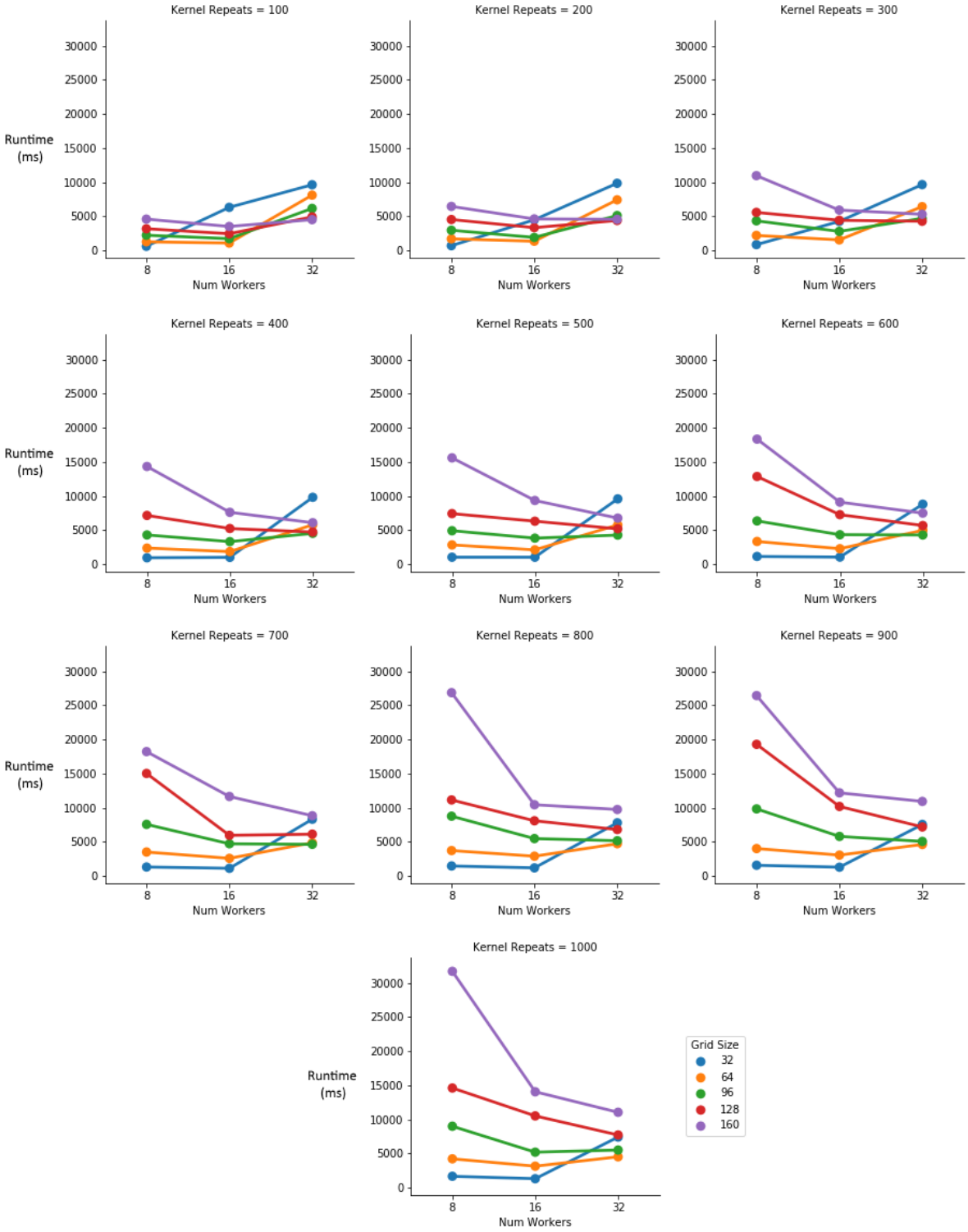

Figure 5.9: These graphs show the runtimes, in milliseconds, of the sqrt computation kernel with various grid sizes and various numbers of kernel repeats, against the number of threads used. Run on XXXII, each point is the mean of 10 repeats, with insignificant variance. We have fewer repeats with these graphs, as these graphs were part of a range whose only purpose was to let us get a feel for the performance characteristics of different kernels, such that we can select appropriate kernels for subsequent experiments.

num_runs:	11
num_stages:	1
grid_size:	32
num_iterations_0:	5000
set_pin_bool_0:	0
pinnings_0:	-
kernel_repeats_0:	100
kernel_durations_0:	-
kernels:	sqrt
num_workers_0:	8

Figure 5.10: The base config settings for the experiments used to select interesting kernel complexities, the results of which are given in figure 5.9. Certain settings will change for different tests.

From figure 5.9, looking at the case with 100 kernel repeats, we can see that the graph is "twisted." That is, whilst in all cases increasing threads beyond 16 degrades performance, the smaller grid sizes perform much worse with more threads, and the larger grid sizes are not affected as much. And again, as expected, we see that as the kernel complexity is increased (more kernel repeats) the graphs are slowly untwisted, since we are gradually increasing the total amount of work to be done, which lessens the ratio of overhead to useful work.

From these tests we have selected our interesting instances, ready for the next stage of experiments. These instances are:

CPU Small Uses the sqrt kernel (simply named CPU in our config) to stress the CPU. Small grid size and few kernel repeats, meaning we should encounter a point where adding threads degrades performance. Parameters are as follows:

num_runs: 101
num_stages: 1
grid_size: 32
kernels: CPU
num_iterations_n: 1000
set_pin_bool_n: 2
kernel_repeats_n: 75
num_workers_n: Varies
pinnings_n: Varies

CPU Large Uses the sqrt kernel (simply named CPU in our config) to stress the CPU. Large grid size and many kernel repeats, meaning adding threads should always benefit performance. Parameters are as follows:

num_runs: 101
num_stages: 1
grid_size: 256
kernels: CPU
num_iterations_n: 1
set_pin_bool_n: 2
kernel_repeats_n: 1000
num_workers_n: Varies
pinnings_n: Varies

VM Small Uses the mem_read kernel (simply named VM in our config) to stress the virtual memory (by allocating memory and touching certain bytes). Small grid size and few kernel repeats, meaning we should encounter a point where adding threads degrades performance. Parameters are as follows:

num_runs: 101
num_stages: 1
grid_size: 32
kernels: VM
num_iterations_n: 1000
set_pin_bool_n: 2
kernel_repeats_n: 10
num_workers_n: Varies
pinnings_n: Varies

VM Large Uses the mem_read kernel (simply named VM in our config) to stress the virtual memory (by allocating memory and touching certain bytes). Large grid size and many kernel repeats, meaning adding threads should always benefit performance. Parameters are as follows:

num_runs: 101
num_stages: 1
grid_size: 256
kernels: VM
num_iterations_n: 1
set_pin_bool_n: 2
kernel_repeats_n: 1000
num_workers_n: Varies
pinnings_n: Varies

5.3 Finding Optimal Thread Counts

With these experiments, the aim is to characterise the performance of each selected instance in fine detail, to give us a baseline for comparison. To this effect, we evaluate the runtime across a longer, more refined range of core counts and thread counts than in the previous section. Note - typically, programs are profiled individually, to find the optimal parameters. This experiment is akin to someone profiling a program to find the best possible performance.

As discussed in section 2.2.4, for these experiments, we are interested in a comparison across machines. Each experiment was performed on two machines (spa and XXXII,) adjusting the maximum core/thread counts where applicable. We have one machine with 2x6 Core CPUs, with hyper-threading, for a total of 24 threads (spa,) and another with 4x8 Core CPUs, with hyper-threading, for a total of 64 threads (XXXII). The exact details of these machines can be found in section 4.1. The config files used for these experiments are those from section 5.2.3.

5.3.1 Finding the Optimal Thread Counts for the CPU Small Workload

Looking at the case where we have two cores in figure 5.11, we can see that going above two threads only degrades performance. This is expected, since we are not adding any additional compute power, but we are adding more overhead with more threads. This continues up until we reach 6 cores. This seems to be the "sweet spot", where we have just the right amount of threads, and any more or any less degrades performance. This is a good result, as this is the performance characteristic we desired for the small workloads, as discussed in section 2.2.3.

Another observation is that wherever adding threads degrades performance, we see that it degrades at the same rate. This applies in both the non-hyperthreaded graph and the hyperthreaded graph. This is consistent with what we would expect, as adding a set number of threads should add a set amount of overhead in managing those threads, and inefficiency related to contention between threads.

Interestingly, once we get into the hyperthreads, and particularly with many hyperthreads, we see a larger "sweet spot", that is, in the case with 24 virtual CPU cores, assigning anywhere from 8-16 threads results in decent performance. This may be due to more spatial locality, and less context switching cost due to the hyperthreads being entirely virtual, and occupying the same die location as the corresponding physical CPU core.

In profiling a program from these results, one might conclude that using around

24 cores with 12 threads would be optimal. Later, we show that this is a potential pitfall if this program is to be run in conjunction with others, where contention over resources proves to be an issue.

From figure 5.12, we can see that for the CPU small workload, similar to the case on spa, we gain performance up to a point, and then it starts degrading as we lose threads. Interestingly, again, we have a "sweet spot" at around 6 threads. However, this occurs when we provide access to a large amount of cores, (26+,) and not when we restrict access to fewer virtual CPU cores (8-24.)

We can also see that once we start including hyperthreads, the performance characteristics of this workload are very regular, and perform best with 8 threads.

On the whole, these are good results, as they are in line with what we would expect.

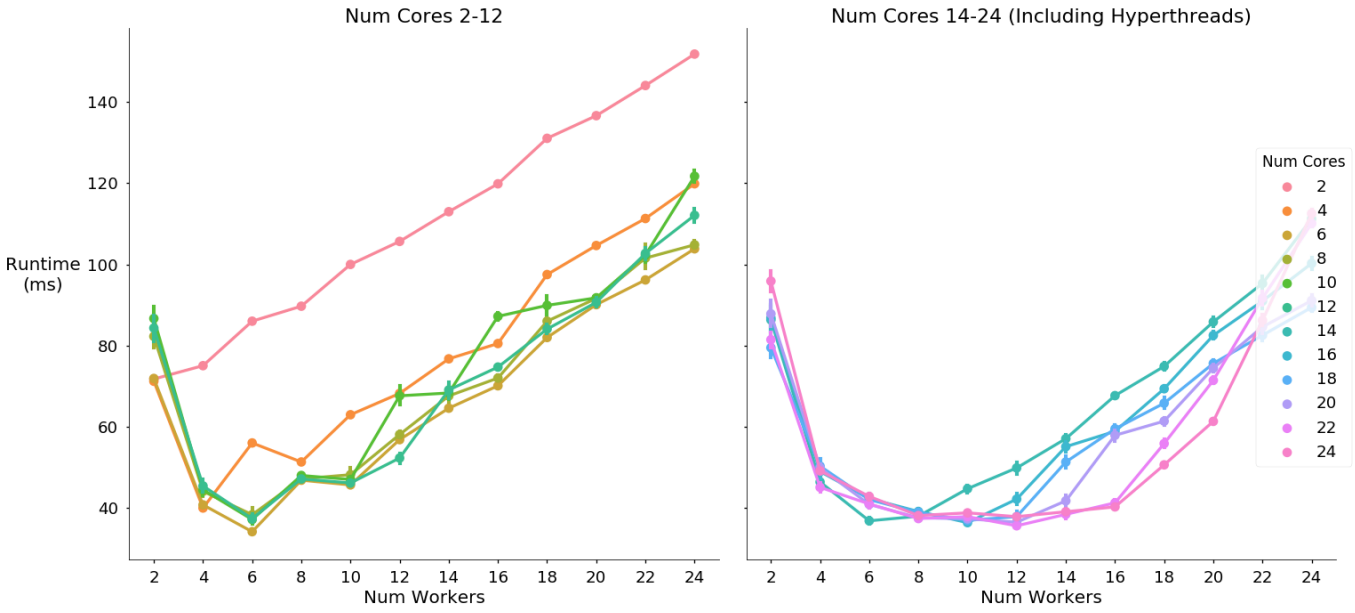

Figure 5.11: Runtimes in milliseconds of the CPU small workload on spa, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

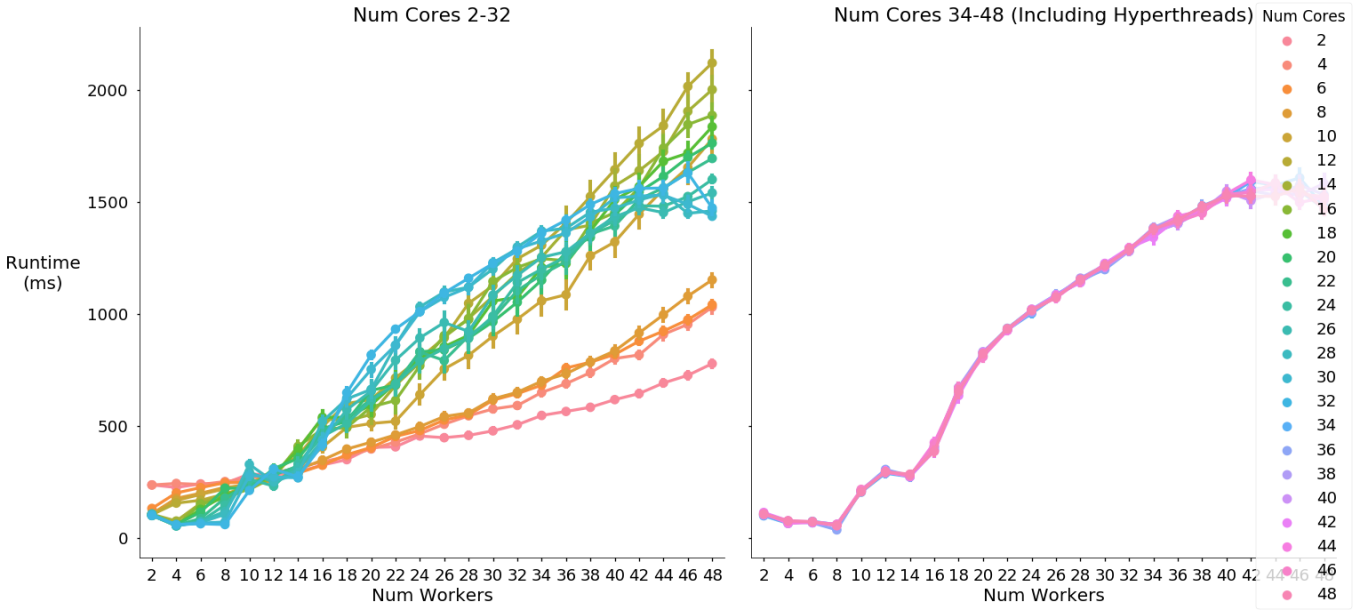

Figure 5.12: Runtimes in milliseconds of the CPU small workload on XXXII, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

5.3.2 Finding the Optimal Thread Counts for the CPU Large Workload

In contrast to the previous case, from the graphs in figure 5.13, we can see that there is little to no penalty to assigning too many threads, regardless of how many CPU cores we have access to. This is because we have more work to do per iteration, meaning less contention between threads. This is described in more detail in section 2.2.2. Again, this is a good result, as this is a desired characteristic of the large workloads, as discussed in section 2.2.3.

Another take away from these results is that hyperthreading does not hinder our performance, and may give some slight gain. Looking at the case with 24 virtual CPU cores, there is a significant jump in performance from the case with 22 virtual CPU cores. We cannot say why this is, as answering this question would require further investigation, and we felt our time was better spent elsewhere. If we discard this case, we do not see any benefit from utilising hyperthreading.

The results presented in figure 5.14 for XXXII resemble the results given in figure 5.13 for spa. In general, we see that for this workload, we do not reach the point where adding more threads would degrade performance.

However, in this case, adding more threads than cores can sometimes be beneficial, e.g. with 4 cores, using 16 threads performs substantially better than 4 threads.

Another takeaway from these graphs is that utilising hyperthreads gives us approximately the same performance as the case without. This is similar to the results from spa, if we discard the case with 24 cores.

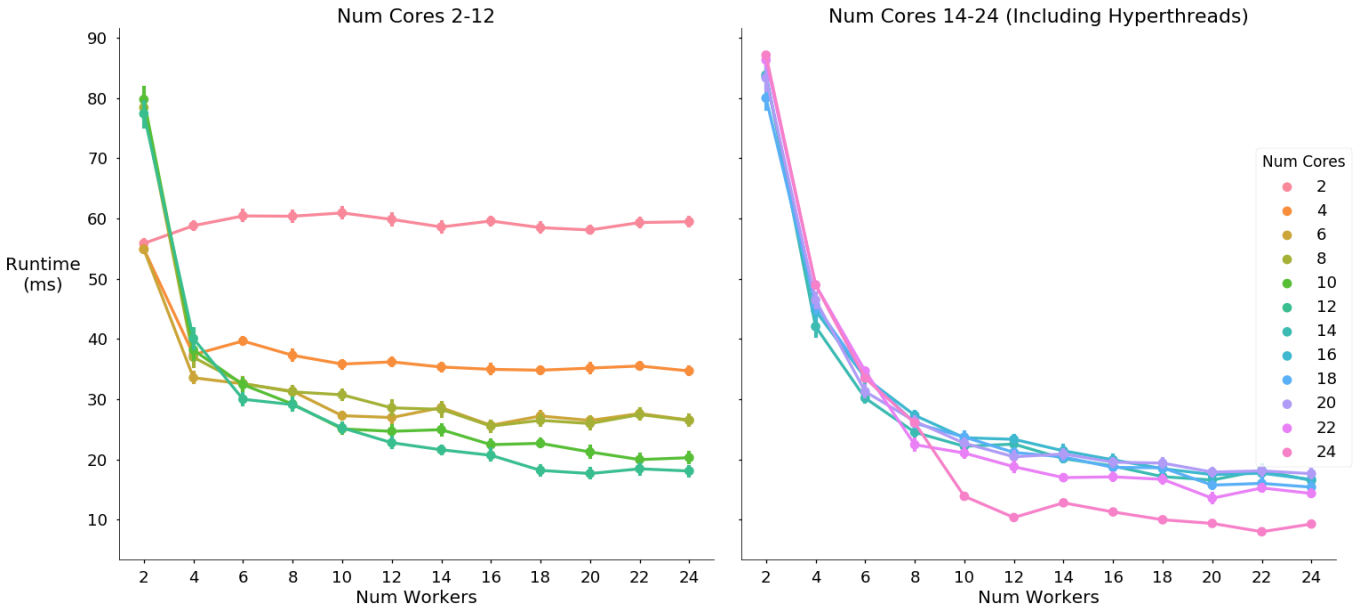

Figure 5.13: Runtimes in milliseconds of the CPU large workload on spa, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

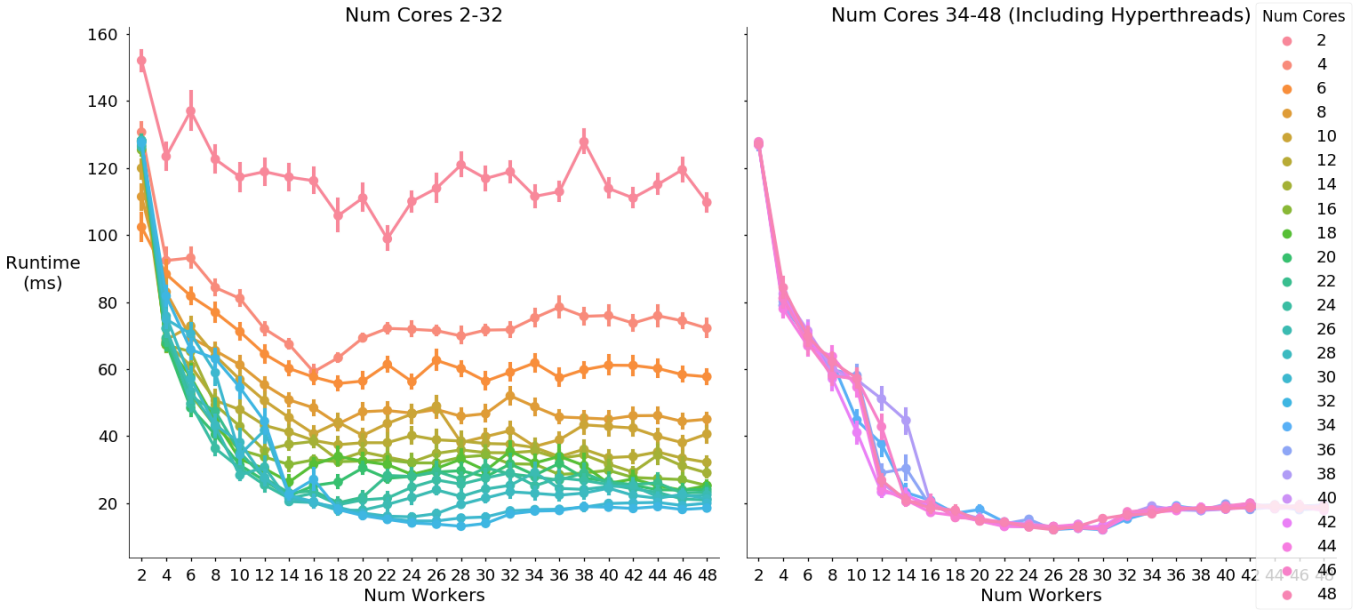

Figure 5.14: Runtimes in milliseconds of the CPU large workload on XXXII, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

5.3.3 Finding the Optimal Thread Counts for the VM Small Workload

Comparing the graphs in figure 5.15 to the previous cases, we can see similarities to both. Similar to the first case, we can see that, at least for the case without hyperthreads, there is a point at which adding more threads degrades performance. Indeed, again it seems to be at 6 cores/threads. Another similarity is that, again for the case without hyperthreads, adding more threads than we have cores generally degrades performance. However, the degradation is not as steady, or even in the case where we have access to 12 cores, we do not see much degradation at all.

We can see reasonable scaling of performance with the number of workers, up until we reach 6 cores/threads. Here, our performance plateaus, suggesting that the critical resource, in this case the virtual memory of the system, is saturated.

In the case where we have access to hyperthreads, interestingly we see that there is clear, tangible, benefit, in that we obtain our fastest runtimes. Perhaps hyperthreading, which allows threads in the same die location on the CPU to share data, facilitates efficient data use. In general, a strategy of assigning as many cores and threads as we can would be reasonable, as we only just start to see some performance degradation as we approach 24 threads in the case where we have access to 24 cores. However, we expect this to change when the program is run in contention with others.

The results presented in 5.16 tell a similar story to those in 5.15, in that they seem to follow a regular pattern, that with providing access to more virtual CPU cores, we do not gain more performance, but rather a bigger "sweet spot", where we can allocate more threads without losing performance. Again, this large "sweet spot" suggests that the critical resource (the virtual memory) is saturated.

However these results do present some oddities. For example, in the case where we have access to two virtual CPU cores, the performance varies wildly with different numbers of workers, with a large amount of variance. This variance may be due to the cost of memory operations changing, as we have a large amount of threads running on just two cores, meaning that there are many context switches between threads. We see a similar pattern when we have access to four virtual CPU cores, however, once we are past four, we settle into the regular pattern, and when we start utilising hyperthreading, we neither lose nor gain performance. This is in contrast to the results from spa, as in that case, we could both lose or gain performance.

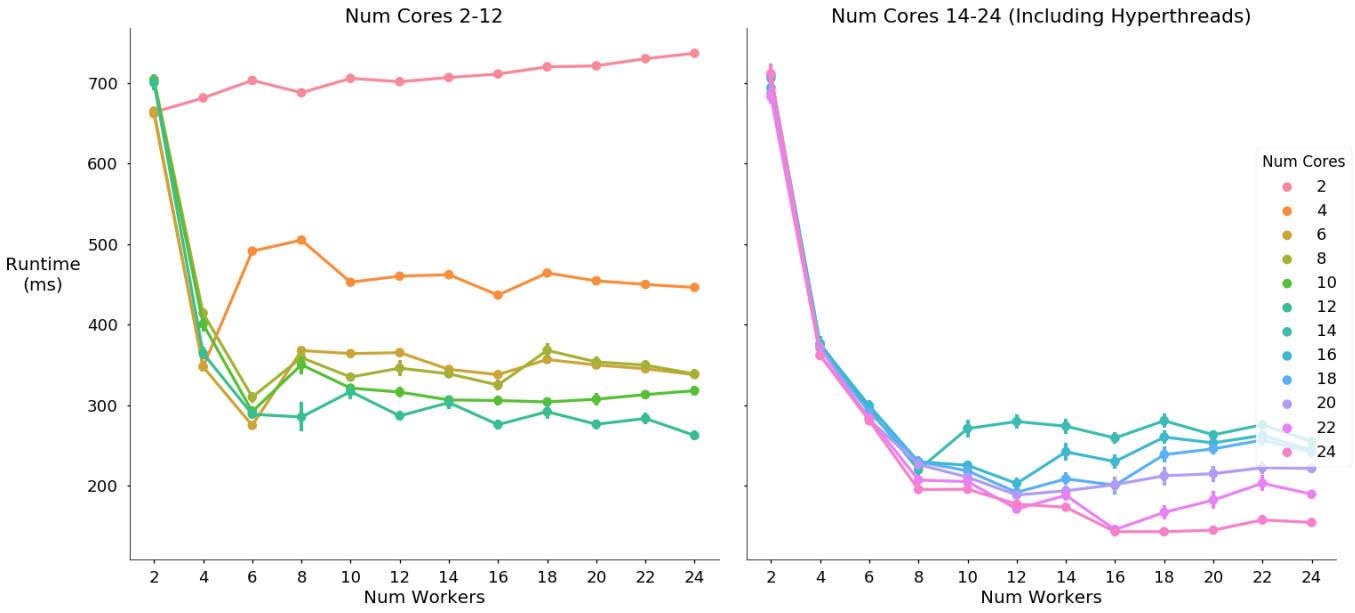

Figure 5.15: Runtimes in milliseconds of the VM small workload on spa, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

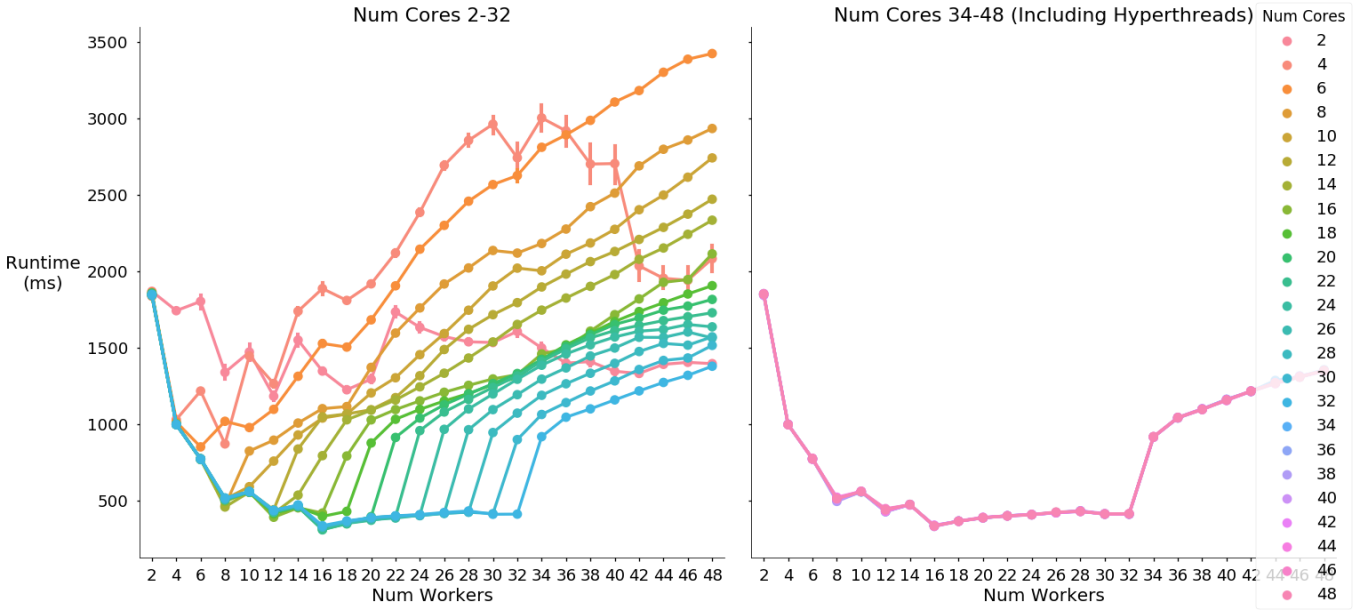

Figure 5.16: Runtimes in milliseconds of the VM small workload on XXXII, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

5.3.4 Finding the Optimal Thread Counts for the VM Large Workload

In the graphs given in figure 5.17, we see the most consistent performance scaling yet. With only minor variance, we see that all cases start by following the same performance curve. When we assign more threads than virtual CPU cores, we see each curve level off, indicating that assigning more threads than cores has little to no penalty.

Again, we see reasonable scaling of performance up to a point, and then it starts to slow. Compare the runtimes for 6 threads (approximately 1500ms) and for 12 (approximately 900ms,) The performance is not quite scaling as expected, suggesting that the critical resource, in this case the virtual memory of the system, is beginning to be saturated.

Another aspect of these graphs is that hyperthreading, again, gives us a tangible benefit, as was the case with our previous VM workload. However, it is not nearly as substantial.

These results in figure 5.18 closely resemble those from spa, 5.17. In fact, almost all of the same conclusions can be drawn from these. However, we see much clear evidence of the critical resource becoming saturated, as we do not significantly improve our performance going above 20 threads.

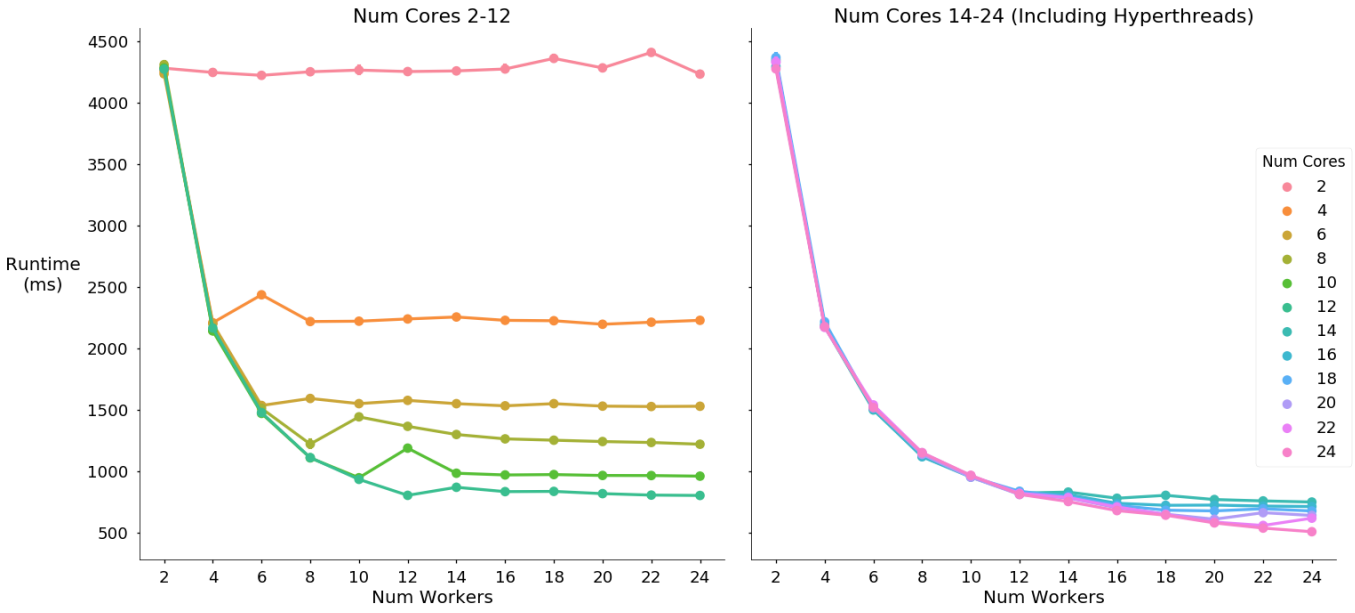

Figure 5.17: Runtimes in milliseconds of the VM large workload on spa, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

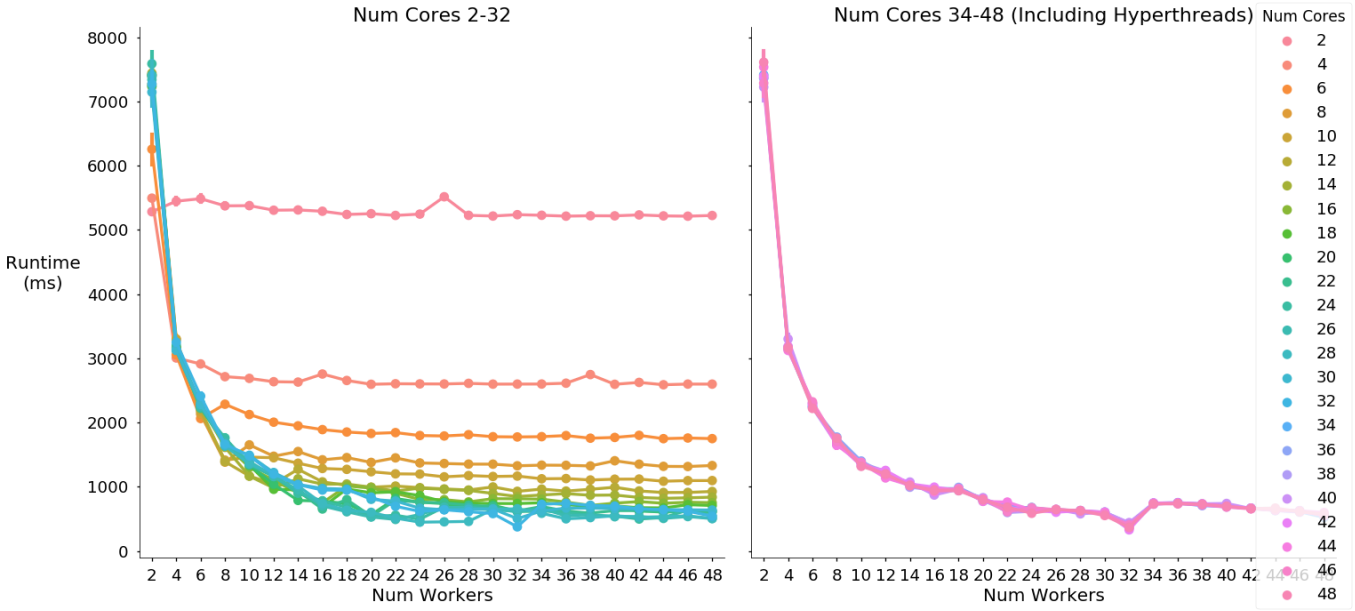

Figure 5.18: Runtimes in milliseconds of the VM large workload on XXXII, for varying numbers of workers (threads) and cores (virtual CPU cores.) Points are the mean of 100 values, error bars show 95% confidence intervals. Results have been split across two graphs for clarity.

5.3.5 Overall Conclusion

From these results, we can see that when a program is run in isolation, there is no disadvantage to providing the program with more cores than necessary. This may be because, whilst modern schedulers are not explicitly contention aware, they do aim to minimise unnecessary context switches, meaning that even with access to more CPU cores, we may not see increased overhead.

Both of our VM workloads have shown signs of the critical resource (the memory of the system) becoming saturated, it will be interesting to see how running these programs in contention affects their performance characteristics.

The intention of these experiments is to ascertain the performance characteristics of these programs when run in isolation, such that we can perform a direct comparison to their performance when run in contention with others. These results achieve that, we can use them as if we were profiling each program, to select the best parameters. For our contention experiments, we will see how choosing the optimal parameters for each program when they were run in isolation compares to the overall optimal parameters, when we take into account contention.

5.4 Contention Experiments

In this section, we present the results of the contention experiments, as described in section 2.2.5. These experiments are designed to assess the performance of two programs running in contention, with different assignments of threads and cores. To do this, we start both programs simultaneously, measuring overall performance in terms of throughput, by summing the runtimes of each program. This is done for each combination of values for the number of workers and the number of cores for each program.

Recall that the purpose of these experiments is to investigate if and how much we could improve performance by taking program contention into account in scheduling. To do this, we will find the performance of the case where we take each programs best settings from data corresponding to them running in isolation, (i.e. find the best number of cores / threads to use from the previous graphs,) and we want to compare this to the case which gave us the best performance when they are run simultaneously (i.e., in contention.) The difference between these two cases is how much we could improve by taking contention into account.

We intended to present results from multiple machines (spa and XXXII,) however due to time constraints, we could not complete the experiments on XXXII. This is because on spa, incrementing the number of workers and the number

of cores in steps of four, we have six values to evaluate (4 - 24.) With two programs, we have four variables to alter, so this results in $6^4 = 1296$ experiments. On XXXII, we have twelve values to vary (4 - 48,) so this results in $12^4 = 20,736$. Since the experiments on spa took around a week to complete, we felt our time was better spent on other aspects of the project.

Presenting these results is complex, since we have five variables in our experiments, the number of cores for each program, the number of workers for each program, and the total runtime. To present this, for the graph of each experiment, we fix the number of cores assigned to each program to whatever gave us the best overall runtime (note - from the previous experiments, assigning more cores than threads did not do very much, meaning that the number of cores assigned is of lesser importance.) Then we produce a 3D plot of the data, varying the number of workers assigned to each program. This surface will give us some insight into the "performance space" of our applications. Finally, we plot two points, one showing the runtime of the individually profiled case (in purple,) and the other showing the runtime of the optimal case (in red.)

The config files used for these experiments are combinations of those from section 5.2.3.

5.4.1 Evaluating CPU Large and VM Large in Contention

The results presented in figure 5.19 should show little contention, as the workloads used (CPU large and VM large) stress different resources, meaning that they are not competing, so we should be able to use the data from each program running in isolation to make a good prediction of the optimal parameters. We can see that this is indeed the case, as we only see a speedup of 1.03 between the the individually profiled case (the purple dot) and the optimal configuration (the red dot.)

We can see in this graph that we have two performance curves which show that adding threads improves performance. The first, for CPU large, is very clear, and is the biggest factor in minimising the runtime. The second, for VM large, can be seen along the top edge, where the number of workers for CPU large is four.

These results tell us that, for workloads which do not compete for the same resource, our contention aware plastic parallel programming library cannot provide a significant improvement. However, we also would not lose any performance, meaning other programs may still benefit, even if this pair will not.

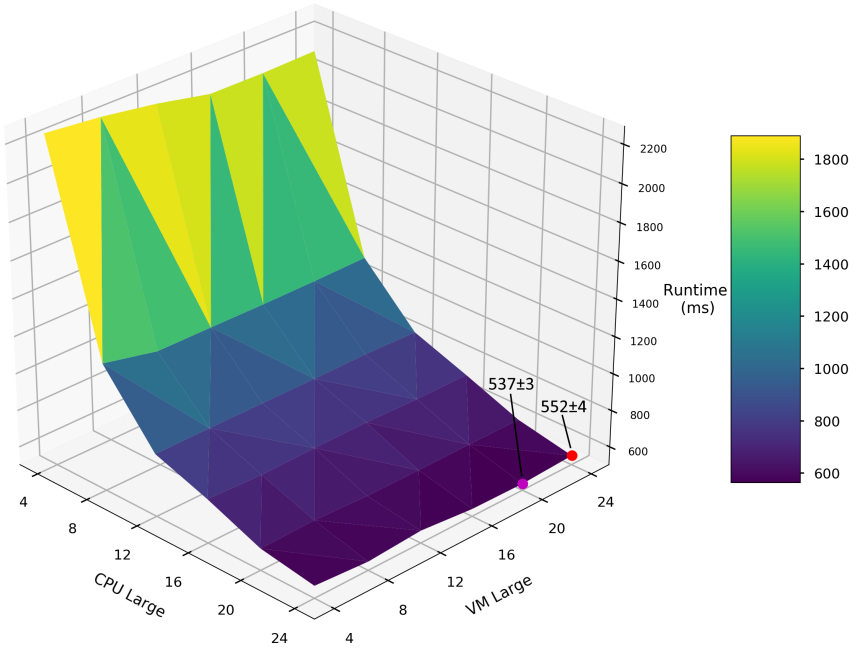

Figure 5.19: A 3D graph showing the total (summed) runtimes in milliseconds of the CPU large and VM large workloads running in contention on spa, and how they are affected by the number of workers assigned to each program. The red dot shows the result we would get if we used settings which were optimal when these programs were run in isolation. The purple dot shows the overall optimal case, which comparatively has a speedup of 1.03. This is what we could gain with a contention aware plastic parallel programming library.
These results represent the case where the number of cores assigned to each program is optimal, (in this case, num cores for CPU large = 24 and num cores for VM large = 24.) Each point is the mean of 100 runs, with insignificant variance.

5.4.2 Evaluating CPU Small and CPU Large in Contention

The results presented in figure 5.20 show the characteristics of the CPU small and CPU large workloads when run in contention. These workloads directly compete for resources, and as such, here we expect to see the possibility for improvement. Indeed, this is the case, with a huge speedup of 2.44, and very different parameters (independent case: CPU small num_cores: 20, CPU small num_workers: 12, CPU large num_cores: 24, CPU large num_workers: 24, optimal case: CPU small num_cores: 24, CPU small num_workers: 20, CPU large

num_cores: 20, CPU large num_workers: 12.)

This large change in parameters may be surprising at first, however, from looking at the topology of the graph, the downwards slope from 24 threads for CPU large makes sense. Since CPU large can generally occupy all the cores it can get, we must reduce them in order for CPU small to get its fair share, allowing us to run both programs simultaneously and saturate the entire machine, improving our performance. Also, recall that on spa, we have 12 physical CPU cores and 24 virtual ones. So, when programs are run in isolation, we may gain a tangible benefit from utilising hyperthreading. That is, by assigning more threads than physical CPU cores. Indeed, looking at our previous results for CPU large on spa (section 5.13,) we may be able to gain a slight edge. However, when run in contention where CPU cores are at a premium, this is unlikely to be an optimal strategy. All this helps explain the change in threads for CPU large from the independent case (red dot) and the optimal case (purple dot.)

Looking at the topology of the graph again, we can see that there is a trough in which we can change the number of threads for CPU small, without impacting the runtime too significantly. This explains the change in threads for CPU small from the independent case (red dot) and the optimal case (purple dot.) A reason for this trough may be that the runtime of these programs running in conjunction is dominated by CPU large, and whilst there are slopes in this trough, they are simply not as significant.

These results show us that for contentious workloads, there is a large potential for improvement, and that if one workload dominates the other in the competition for resources, (in this case, CPU large dominates,) the parameters we should set for it are more critical to our overall performance, as they must enable the sharing of resources.

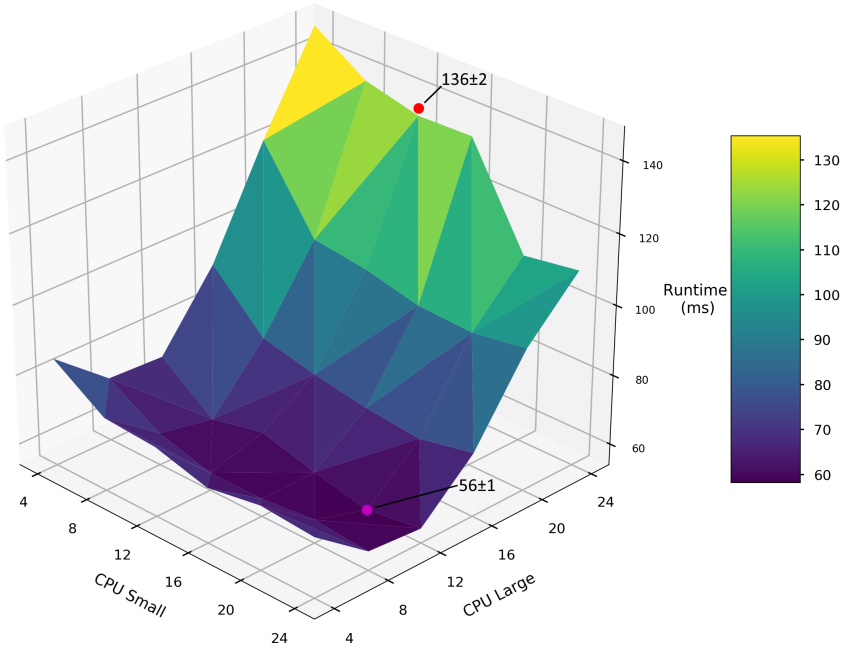

Figure 5.20: A 3D graph showing the total (summed) runtimes in milliseconds of the CPU small and CPU large workloads running in contention on spa, and how they are affected by the number of workers assigned to each program. The red dot shows the result we would get if we used settings which were optimal when these programs were run in isolation. The purple dot shows the overall optimal case, which comparatively has a speedup of 2.44. This is what we could gain with a contention aware plastic parallel programming library.
These results represent the case where the number of cores assigned to each program is optimal, (in this case, num cores for CPU small = 24 and num cores for CPU large = 20.) Each point is the mean of 100 runs, with insignificant variance.

5.4.3 Evaluating CPU Small and VM Small in Contention

We would expect that, since our two workloads do not directly compete for the same system resources, we would not see much improvement or difference between the individually profiled case and the optimal case. However, looking at the results in figure 5.21, we can see that this is not the case. Indeed, we see a significant speedup of 1.60.

It seems that the performance characteristics of our small workloads significantly change when they are run in conjunction with other programs. From the graph,

we can see that the "sweet spot" for the CPU small workload is not around 16 workers, as opposed to 6 (from the individually profiled case.) This may be because, unlike our large workloads, the small ones are "balanced" at their sweet spot, and this relates to other factors, notably CPU speed. If another program is running, the small workload will get less CPU time, analogous to the CPU speed decreasing. It is plausible then that the sweet spot would shift, as the balance has changed.

These results tell us that for "small" workloads, (that is, workloads where adding threads is not an optimal strategy,) their performance characteristics when run in contention can change, and getting the parameters right is important. This shows promise for our contention aware plastic parallel programming library.

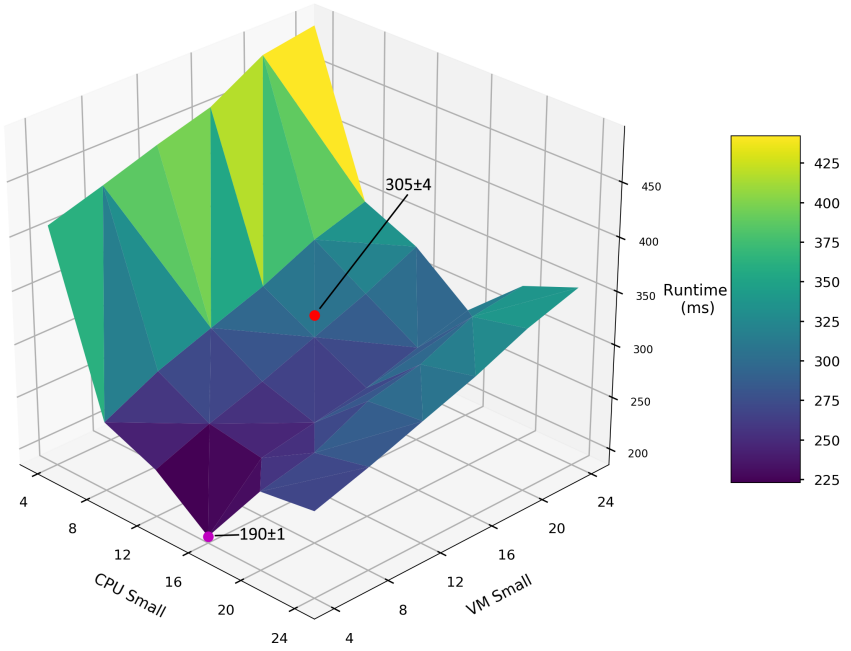

Figure 5.21: A 3D graph showing the total (summed) runtimes in milliseconds of the CPU small and VM small workloads running in contention on spa, and how they are affected by the number of workers assigned to each program. The red dot shows the result we would get if we used settings which were optimal when these programs were run in isolation. The purple dot shows the overall optimal case, which comparatively has a speedup of 1.60. This is what we could gain with a contention aware plastic parallel programming library.
These results represent the case where the number of cores assigned to each program is optimal, (in this case, num cores for CPU small = 24 and num cores for VM small = 4.) Each point is the mean of 100 runs, with insignificant variance.

5.4.4 Evaluating VM Small and CPU Large in Contention

The VM small and CPU large workloads, again, do not directly compete for system resources. Indeed, from figure 5.22, we can see that the VM small workload has little effect on the total runtime, and that it is mostly affected by the CPU large workload. We do, however, see a substantial speedup of 1.22. This is not as drastic an improvement as seen in our previous cases, and we can also see that the values for the parameters of each program are reasonably close, but this small change still marks a valuable improvement.

These results again show us that the performance characteristics of our small workloads significantly change when run in conjunction with other programs, and that there is potential for a compelling gain in performance.

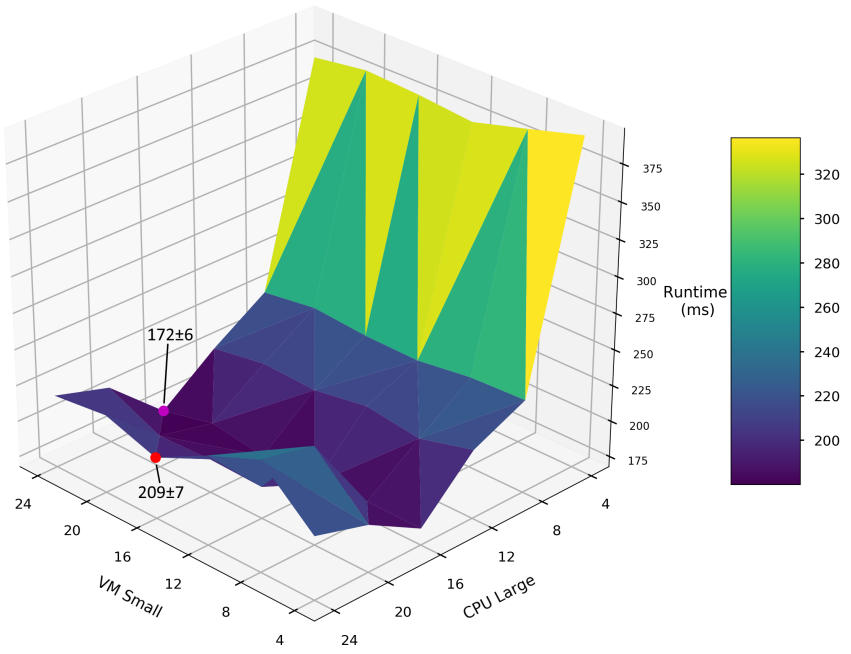

Figure 5.22: A 3D graph showing the total (summed) runtimes in milliseconds of the VM small and CPU large workloads running in contention on spa, and how they are affected by the number of workers assigned to each program. The red dot shows the result we would get if we used settings which were optimal when these programs were run in isolation. The purple dot shows the overall optimal case, which comparatively has a speedup of 1.22. This is what we could gain with a contention aware plastic parallel programming library.
These results represent the case where the number of cores assigned to each program is optimal, (in this case, num cores for VM small = 24 and num cores for CPU large = 24.) Each point is the mean of 100 runs, with insignificant variance.

5.4.5 Evaluating VM Small and VM Large in Contention

The results presented in figure 5.23 are contrary to what we might expect. The VM small and large workloads directly compete for the use of system resources, and as such, we would expect them to interfere with each other. However, these results do not reflect this. In fact, we see a speedup of merely 1.06. Looking

back at our previous results, (in sections 5.3.3 and 5.3.4,) this is actually not wholly unexpected, as in all cases we saw evidence of the critical resource (in this case the virtual memory) becoming saturated. This would explain why we see this plateau in our graph, and in particular why significant changes to the assignments of threads does not significantly affect our performance, since the number of threads does not directly relate to the performance of a VM workload.

For our contention aware plastic parallel programming library, the takeaway here is that for applications where the virtual memory is a performance critical resource, we may not be able to attain performance gains where we would expect, or at all. As such, an optimal strategy in this situation would be to minimise the overall number of threads used, in order to minimise overhead and free up resources for other programs running on the system.

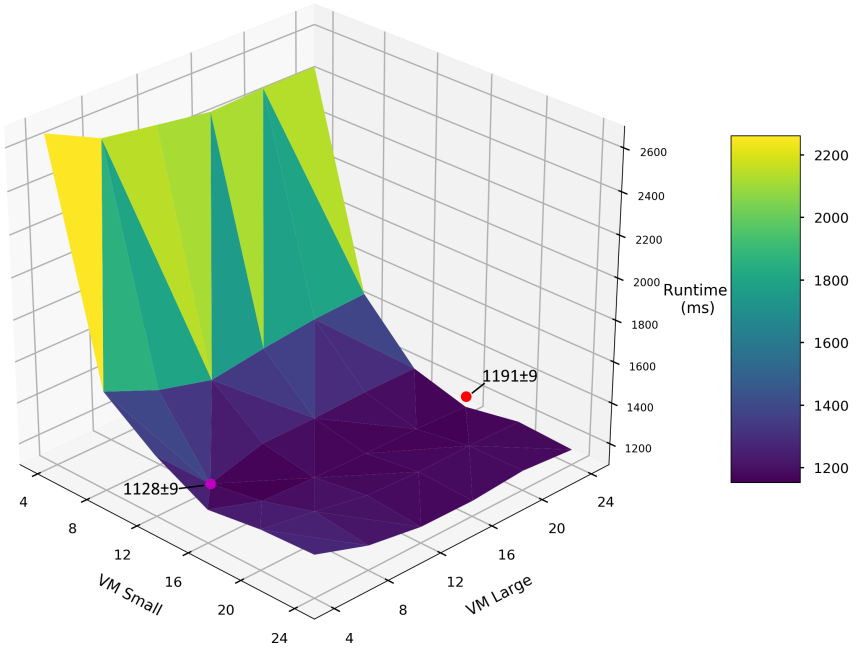

Figure 5.23: A 3D graph showing the total (summed) runtimes in milliseconds of the VM small and VM large workloads running in contention on spa, and how they are affected by the number of workers assigned to each program. The red dot shows the result we would get if we used settings which were optimal when these programs were run in isolation. The purple dot shows the overall optimal case, which comparatively has a speedup of 1.06. This is what we could gain with a contention aware plastic parallel programming library.
These results represent the case where the number of cores assigned to each program is optimal, (in this case, num cores for VM small = 16 and num cores for VM large = 24.) Each point is the mean of 100 runs, with insignificant variance.

5.4.6 Overall Conclusion

Again, the performance characteristics of our VM workloads show that their critical resource (the virtual memory) is saturated, as we saw when these programs were run in isolation (in section 5.3.) This is best illustrated in figure 5.23.

These results have shown that there is clear potential for substantial speedups from the use of a contention aware plastic parallel programming library. We have seen that the performance characteristics of programs can change signifi-

cantly when run in contention with others (particularly our "small" programs,) and that depending upon the situation, using parameters calculated from each program running in isolation may or may not be a good strategy.

5.5 Additional Experiments

5.5.1 Extended Contention Experiments

The intention behind these extended contention experiments is to investigate a more complex scenario, involving three programs run in contention. To keep the total runtime of these experiments reasonable, we further restricted the range of each parameter in our tests compared to our previous contention experiments. We test 3 values per parameter, 4, 8, and 12, resulting in 729 tests per experiment. Testing 6 values per parameter would result in 46,656 tests!

This relatively low granularity means that there may exist an even better configuration of parameters that we simply did not evaluate. As such, this means that the overall speedup results that we see can only be improved upon. This goes for all of our experiments, as in each case, we had to sacrifice some detail due to time constraints, however this is especially applicable here, as we had to significantly restrict the number of combinations we tested.

The result presented in figure 5.24 shows the speedup of one combination of programs, run on spa. The results presented in figure 5.25 show the speedups of different combinations of programs, run on XXXII. Comparing the case with three instances of CPU small, we can see some interesting features. Notably, that the speedup we see on spa is much more drastic compared to the speedup on XXXII. Also, we see that whilst the independent runtime on spa is worse than either case on XXXII, the contention aware runtime is significantly better than either case on XXXII. Throughout our other tests, taking a look at our raw data, we observe that spa proved to be the faster machine, with anywhere up to a 1.8x speedup compared to the same workload on XXXII. We would therefore expect the contention aware runtime for spa to be an improvement over XXXII, and this suggests that the independent case for spa may simply a particularly bad configuration of parameters.

Looking at the other results from XXXII, presented in figure 5.25, we can see another interesting feature. The situations involving purely CPU workloads experienced speedups from 1.13 to 1.18, whereas the situations involving VM workloads saw more significant speedups of 1.39 and 2.12, as well as much longer overall runtimes. This is intuitive, as memory operations are typically much more expensive compared to purely computational operations, and as such, using sub-optimal parameters for a memory dependent application may

result in larger slowdowns compared to a CPU dependent application.

Kernel 1	Size 1	Kernel 2	Size 2	Kernel 3	Size 3	Independent Runtime	Contention Aware Runtime	Speedup
CPU	Small	CPU	Small	CPU	Small	385 ± 2 ms	159 ± 6 ms	2.43

Figure 5.24: A table showing the total runtimes of different combinations of three programs in milliseconds run on spa, first using settings as if they were individually profiled (Independent Runtime,) then using contention aware settings (Contention Aware Runtime,) and the speedup between them. Due to the high dimensionality of each case, we present this as a table, rather than a sequence of graphs. Each data point is the mean of 100 runs, shown with 95% confidence intervals.

Kernel 1	Size 1	Kernel 2	Size 2	Kernel 3	Size 3	Independent Runtime	Contention Aware Runtime	Speedup
CPU	Small	CPU	Small	CPU	Small	298 ± 7 ms	251 ± 12 ms	1.18
CPU	Small	CPU	Small	CPU	Large	271 ± 5 ms	236 ± 13 ms	1.14
CPU	Small	CPU	Small	VM	Small	1144 ± 28 ms	826 ± 14 ms	1.39
CPU	Small	CPU	Small	VM	Large	3390 ± 57 ms	1602 ± 54 ms	2.12
CPU	Small	CPU	Large	CPU	Large	236 ± 5 ms	209 ± 3 ms	1.13

Figure 5.25: A table showing the total runtimes of different combinations of three programs in milliseconds run on XXXII, first using settings as if they were individually profiled (Independent Runtime,) then using contention aware settings (Contention Aware Runtime,) and the speedup between them. Due to the high dimensionality of each case, we present this as a table, rather than a sequence of graphs. Each data point is the mean of 100 runs, shown with 95% confidence intervals.

Chapter 6

Future Work and Conclusion

6.1 Summary

Previous work has demonstrated that contention between programs has performance implications, and that by considering their contention when scheduling, we can improve system throughput [8]. In this project, we have built upon this, by investigating the question: "Can we gain more performance by exploiting both contention aware scheduling and plastic programming?" As such, we have laid the groundwork for a contention aware plastic parallel programming library. We have demonstrated how such a library may be implemented, that the overhead incurred is acceptable, and that with two programs run in contention, it is possible to attain substantial gains in performance, with potential speedups from 1.03 to 2.44, compared to a non contention-aware system. We have also demonstrated that, with three programs run in contention, there are potential speedups of 1.13 to 2.43 compared to a non contention-aware system.

Our work towards a contention aware plastic parallel programming library shows promise, and provides exciting potential for further research, which are discussed in section 6.2.

6.2 Future Work

In this section, we discuss possible extensions to this project, which could be done by a future student.

6.2.1 More Detailed And Numerous Experiments

For many of our experiments, in particular the contention experiments, we had to cut down on their detail, and the overall number, due to time constraints. The existing framework easily allows us to run additional, increasingly detailed experiments, and a significant step would be to evaluate the potential of this system with real applications, as opposed to our purely synthetic workloads, so this is left as work for the future.

6.2.2 More Plasticity

Currently, the plasticity provided in the synthetic test program is comprised of the ability to change the number of threads used, as well as the virtual CPU cores they are pinned (restricted) to. One extension could be to add additional plastic features, and to evaluate the gain, if any, that they provide. As long as the program is still correct, (in that it produces the correct output,) any plastic modification could be made. For example, we could vary the data structures used, or the pattern in which data is accessed (currently, just sequential.)

6.2.3 A Complete Contention Aware Plastic Programming Library

In the first year of this project [20], we demonstrated how a contention aware plastic programming library could be implemented, as well as showing that the overhead incurred was not significant. However, the pattern we implemented (map-array) did not show much room for improvement. In this second year, we have evaluated a more complex pattern, that of stencil codes, and specifically the jacobi pattern, without integrating it into such a library. One natural extension would be to add this pattern to such a library, and possibly other common parallel programming patterns, such as the pipeline pattern, or a recursive pattern. Another natural extension would be to evaluate more workloads, such as workloads which stress the HDD, or perform many OS calls. A synthetic test program with multiple patterns and workloads could give substantial importent data.

6.2.4 A Rudimentary Controller Application

In the contention aware plastic parallel programming library we presented in the first year of this project [20], we employed a controller application to subsidise the communication and co-ordination between instances of our library. Because

of the complexities of implementing the prototype, the controller application acted according to hand written rules for testing purposes, whereas, in a complete system, it would act according to input that it gathers about the current state of the system. Implementing such a controller application is a possible extension of this project.

6.2.5 A More Advanced Controller Application

In the previous extension, we would still be using hand written (albeit complex) rules to implement the controller application. It may be of some benefit to use some sort of artificial intelligence to figure out the optimal configuration. To generate the input data, we could use established methods of predicting performance, such as software profiling programs before hand, or hardware monitoring programs during execution. This is a well researched field, and here we present a short overview of these two methods:

- Software profiling data involves collecting statistics such as the local IPC and L2 cache misses in a simulator. Data such as this can be used to predict the programs performance, and has been used to tune the compilation of a program for greater performance [11].
- Hardware monitoring requires keeping track of various statistics during program execution. In previous work involving hardware based profiling techniques, several statistics have been evaluated, such as IPC, floating point IPC, resource utilisation, and instruction dependencies [3, 5, 14, 22, 26, 29, 30]. All of these statistics could be used to inform the controller application, in particular, IPC has been shown to be a particularly useful statistic to monitor. The controller application would gather such statistics at regular intervals, and use it to inform its decisions. The advantage of this approach is that it gives data at runtime, allowing for the controller to dynamically adjust the overall configuration on the fly.

6.2.6 Multiple Back-Ends And GPU Support

Currently, our project is based in C++ and utilises pthreads as the parallel programming model. Additional support could be added for other approaches to parallelism (some of which are described in section 1.2.1,) and even languages which target GPUs, such as OpenCL and CUDA. Providing this would allow the controller application to switch between them, as effectively another parameter that we can plastically change. The main decision however, would be between the CPU focused and the GPU focused approaches, as this would provide the most dramatic difference, and is dependent upon the characteristics of the workload in question.

GPUs have lots of compute cores, but with a much simpler architecture compared to a standard CPU core. This makes them powerful for floating point calculations in scientific computing. Utilising GPU oriented parallel programming approaches would introduce new challenges, such as explicitly managing data (copying back and forth from main memory to GPU memory,) and restrictions imposed by the language. For example, in CUDA, device code (to be run on the GPU) must be C only, and cannot be called recursively. Another limitation is the lack of direct control of scheduling. GPUs have separate, efficient, thread managers which schedule threads, typically assigning many more threads than cores.

6.2.7 Other Scheduler Metrics

In this project, we investigated the utility of a contention aware plastic programming library with a focus on throughput as the metric to optimise. An extension to this project could investigate other metrics, such as:

- **Resource Utilisation:** The percentage of time that a resource is busy
- **Turnaround Time:** The amount of time between a process arriving ready to compute to the time it completes
- **Power Efficiency:** The cost of compute power in terms of the energy used

6.2.8 Other Computing Domains

As another extension, other computing domains could be considered.

Mobile Systems

The mobile computing domain introduces new challenges. Power efficiency is of key importance, and as such, this means that we can encounter diverse hardware. Many CPU cores is commonplace, as this leads to greater power efficiency. This makes parallel programming of key importance, in order to utilise the hardware efficiently. Another technique is to combine many low power CPU cores with few high power CPU cores, and even low power GPUs, to enable both efficiency and performance when required. Such features, and shifting our focus to power efficiency, would drastically change how we schedule applications.

Server Systems

A significant application of this technology is server systems with multiple parallel workloads. Taking this into account brings into question many possible extensions. Again, power efficiency is of key importance in many server systems, which typically prefer high core count systems.

A more interesting challenge however would be supporting the use of distributed applications. This involves multiple machines coordinating on the same problem, and brings many questions. Should the controller application be distributed? Should controller applications even know about other nodes working on the same problem? Indeed, we may have a situation where one instance of a program on one node may use a different implementation to another instance on an entirely separate node, even though they are working on the same problem.

6.3 Conclusion

In conclusion, during this project, we have demonstrated that by exploiting contention aware scheduling and plasticity, we can attain substantial gains in performance in situations where programs are run in contention over and above the simple gains attributed to the introduction of parallelism. This lays the foundations for much potential future work.

Chapter 7

Bibliography

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

[2] Hartwig Anzt, Jack Dongarra, and Enrique S. Quintana-Ort. Adaptive precision solvers for sparse linear systems. *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing - E2SC '15*, 2015.

[3] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. *ACM SIGARCH Computer Architecture News*, 29(2):218–229, 2001.

[4] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley, 2008.

[5] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter Cook, and David Albonesi. An adaptive issue queue for reduced power at high performance. *Power-Aware Computer Systems*, pages 25–39, 2001.

[6] Bryan Catanzaro and Kurt Keutzer. Parallel computing with patterns and frameworks. *XRDS: Crossroads, The ACM Magazine for Students*, 17(1):22, 2010.

[7] Rohit Chandra. *Parallel programming in openMP*. Morgan Kaufmann, 2009.

[8] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. Lira. *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '15*, 2015.

[9] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. *2008 IEEE International Symposium on Workload Characterization*, 2008.

[10] Yu Deng, Li Wang, Xiaobo Yan, and Xuejun Yang. A double-buffering strategy for the srf management in the imagine stream processor. *2008 The 9th International Conference for Young Computer Scientists*, 2008.

[11] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction. *ACM SIGPLAN Notices*, 35(11):202–211, 2000.

[12] Johan Enmyren and Christoph W. Kessler. Skepu. *Proceedings of the fourth international workshop on High-level parallel programming and applications - HLPP '10*, 2010.

[13] August Ernstsson, Lu Li, and Christoph Kessler. Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, 2017.

[14] D. Folegnani and A. Gonzalez. Energy-effective issue logic. *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001.

[15] Richard Friedman. Openmp. `http://www.openmp.org/`, 1997. [Online; accessed November-2017].

[16] Matt Godbolt. Compiler explorer tool. `https://godbolt.org/`, 2017. [Online; accessed November-2017].

[17] Daniel Hackenberg, Thomas Ilsche, Robert Schone, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[18] Antonino Ingargiola. Jupyter/ipython notebook quick start guide. `https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html`, 2015. [Online; accessed November-2017].

[19] Intel. Threading building blocks. `https://www.threadingbuildingblocks.org/`, 2006. [Online; accessed November-2017].

[20] Mark Jenkins. Design and analysis of a plastic parallel programming system, 2017.

[21] Project Jupyter. Project jupyter. `https://jupyter.org/`, 2014. [Online; accessed November-2017].

[22] Roberto Maro, Yu Bai, and R. Iris Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. *Power-Aware Computer Systems*, pages 97–111, 2001.

[23] Paul McKenney, Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole. Is parallel programming hard, and if so, why? `http://www.cs.pdx.edu/pdfs/tr0902.pdf`, 2009. [Online; accessed November-2017].

[24] Microsoft. Parallel patterns library (ppl). `https://msdn.microsoft.com/en-us/library/dd492418.aspx`, 2015. [Online; accessed November-2017].

[25] University of Illinois. Parallel patterns. `http://snir.cs.illinois.edu/PPP.html`, 2004. [Online; accessed November-2017].

[26] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2002.

[27] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. *Data Compression Conference (DCC'06)*, 2006.

[28] Caitlin Sadowski and Andrew Shewmaker. The last mile. *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, 2010.

[29] Ruchira Sasanka, Christopher J. Hughes, and Sarita V. Adve. Joint local and global hardware adaptations for energy. *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X) - ASPLOS '02*, 2002.

[30] J.S. Seng, E.S. Tune, and D.M. Tullsen. Reducing power with dynamic critical path information. *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2002.

[31] R.B. Sheeparamatti, B.G. Sheeparamatti, Manjula Bharamagoudar, and Nayan Ambali. Simulink model for double buffering. *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, 2006.

[32] Adam Smyk and Marek Tudruj. Application of mixed mpi/openmp programming in a multi smp cluster computer. *Parallel Processing and Applied Mathematics*, 2002.

[33] Bjarne Stroustrup. Standard for programming language c++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf`, 2005. [Online; accessed November-2017].

[34] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. Mpi and communication—high-performance and scalable mpi over infiniband with reduced memory usage. *Proceedings of the 2006 ACM IEEE conference on Supercomputing - SC '06*, 2006.

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

[36] Amos Waterland. stress - tool to impose load on and stress test systems. `https://people.seas.harvard.edu/~apw/stress/`, 2014. [Online; accessed 29-March-2018].

[37] Xiyang I.A. Yang and Rajat Mittal. Acceleration of the jacobi iterative method by factors exceeding 100 using scheduled relaxation. *Journal of Computational Physics*, 274:695–708, 2014.

Appendices

Appendix A

Code Snippets

```
#ifdef MY_BARRIER
#define MB( x ) x
#pragma message "MY_BARRIER ACTIVE"
#else
#define MB( x )
#endif

#ifdef PTHREAD_BARRIER
#define PTB( x ) x
#pragma message "PTHREAD_BARRIER ACTIVE"
#else
#define PTB( x )
#endif

#ifdef BASIC_KERNEL_SMALL
#define BKS( x ) x
#pragma message "BASIC_KERNEL_SMALL ACTIVE"
#else
#define BKS( x )
#endif

#ifdef BASIC_KERNEL_LARGE
#define BKL( x ) x
#pragma message "BASIC_KERNEL_LARGE ACTIVE"
#else
#define BKL( x )
#endif

#ifdef EXECUTE_KERNELS
#define EXK( x ) x
#pragma message "EXECUTE_KERNELS ACTIVE"
#else
#define EXK( x )
#endif

#ifdef CONVERGENCE_TEST
#define CVG( x ) x
#pragma message "CONVERGE_TEST ACTIVE"
#else
#define CVG( x )
#endif
```

Figure A.1: Some of the C++ preprocessor commands which are used to switch certain features on and off in our synthetic test program. Taken from the file /jacobi/src/jacobi.cpp, which can be found in the project directory.

```
// Each Worker computes values in one strip of the grids. The main
    worker loop does two computations to avoid copying from one grid to
    the other
void worker(uint32_t my_id, uint32_t stage) {

   // Set our affinity
   force_affinity_set(pinnings.at(stage).at(my_id));

   // Determine first and last rows of my strip of the grids
   uint32_t first = row_allocations.at(stage).at(my_id);
   uint32_t last = row_allocations.at(stage).at(my_id + 1);

   // Create grid pointers
   std::vector<std::vector<double>>* src_grid = &grid1;
   std::vector<std::vector<double>>* tgt_grid = &grid2;

   for (uint32_t iter = 0; iter < num_iterations.at(stage); iter++) {

      // Update my points
      for (uint32_t i = first; i < last; i++) {
         for (uint32_t j = border_size; j < grid_size + border_size;
              j++) {

            BKS(basic_kernel_small(*(src_grid), *(tgt_grid), i, j);)

            BKL(basic_kernel_large(*(src_grid), *(tgt_grid), i, j);)

            EXK(execute_kernels(stage, i, j);)
         }
      }

      // Barriers
      MB(my_barrier(stage);)
      PTB(pthread_barrier_wait(&pthread_barriers.at(stage));)

      // Simulate convergence test
      CVG(convergence_test(first, last, stage, my_id);)

      // Flip grid pointers
      std::vector<std::vector<double>>* temp = src_grid;
      src_grid = tgt_grid;
      tgt_grid = temp;
   }
}
```

Figure A.2: The worker function used in our synthetic test program. This shows how the preprocessor commands are used to switch features on and off, how the data grid is divided between threads, and how we use double buffering (with the src_grid and tgt_grid pointers.) Taken from the file /jacobi/src/jacobi.cpp, which can be found in the project directory.

```
uint64_t addpd_kernel(double *buffer, uint64_t repeat) {
...
uint64_t ret = 0;

passes = repeat / 32; // 32 128-Bit accesses in inner loop

addr = (unsigned long long) buffer;

if (!passes) return ret;

/*
 * Input: RAX: addr (pointer to the buffer)
 *        RBX: passes (number of iterations)
 *        RCX: length (total number of accesses)
 */

__asm__ __volatile__(
    "mov %%rax,%%r9;" // addr
    "mov %%rbx,%%r10;" // passes

    // Initialize registers
    "movaps 0(%%r9), %%xmm0;"
    ...
    "movapd 112(%%r9), %%xmm15;"

    ".align 64;"
    "_work_loop_add_pd:"
    "addpd %%xmm8, %%xmm0;"
    ...
    "addpd %%xmm15, %%xmm7;"

    "sub $1,%%r10;"
    "jnz _work_loop_add_pd;"

    : "=a" (a), "=b" (b), "=c" (c), "=d" (d)
    : "a"(addr), "b" (passes)
    : "%r9", "%r10", "xmm0", "xmm1", "xmm2", "xmm3", "xmm4", "xmm5",
        "xmm6", "xmm7", "xmm8", "xmm9", "xmm10", "xmm11", "xmm12",
        "xmm13", "xmm14", "xmm15"
);

ret = passes;

return ret;
}
```

Figure A.3: An example of one of kernels programmed in inline assembly, mulpd. Six such kernels were developed, each utilising different operations for the main portion of work. Taken from the file /jacobi/kernels/src/depreciated/asm_kernels.cpp, which can be found in the project directory.

Appendix B

Experiment Script

```
#!/bin/bash
...
STEP=$( bc -l <<< "100 / ($NUM_RUNS1 * $NUM_RUNS2)" )
TOTAL=$STEP

# Write new configs
scripts/config_generation/cpu_small.sh $FILENAME1
scripts/config_generation/vm_large.sh $FILENAME2
...
make flags="-DSYNC_PROCS=2 -DPTHREAD_BARRIER -DBASIC_KERNEL_SMALL
    -DVARY_KERNEL_LOAD -DEXECUTE_KERNELS -DCONVERGENCE_TEST" main | tee
    $LOG_FILENAME1 $LOG_FILENAME2 > /dev/null
...
printf "0.000%%"

sudo scripts/update-motd.sh "Experiments running! 0.000% complete" | tee
    $LOG_FILENAME1 $LOG_FILENAME2 > /dev/null
...
for ((k=$NUM_CORES_MIN; k<=$NUM_CORES_MAX; k+=$NUM_CORES_STEP))
do
   for ((l=$NUM_WORKERS_MIN; l<=$NUM_WORKERS_MAX; l+=$NUM_WORKERS_STEP))
   do
       # Setup parameters for program 2
       STRING="0..$(($k-1)) "
       ...
       head -n -2 $FILENAME2 > temp.ini

       echo "num_workers_0: \"$l\"" >> temp.ini
       echo "pinnings_0: \"$FULL_STRING\"" >> temp.ini

       mv temp.ini $FILENAME2

       # Run programs
       bin/jacobi $FILENAME1 $COUNT "$(basename $BASH_SOURCE .sh)_1" >>
           $LOG_FILENAME1 &
       bin/jacobi $FILENAME2 $COUNT "$(basename $BASH_SOURCE .sh)_2" >>
           $LOG_FILENAME2
       ...
       printf "\r%.3f%%" "$TOTAL"
       ...
       sudo scripts/update-motd.sh "$(printf "Experiments running!
           %.3f%% complete" "$TOTAL")" | tee $LOG_FILENAME1
           $LOG_FILENAME2 > /dev/null
```

Figure B.1: An example of an experiment script used to call our synthetic test program with various parameters for a contention experiment. Taken from the file /jacobi/scripts/contention/cpu_small_and_vm_large.sh, which can be found in the project directory.

Appendix C

Data Analysis Program

```
...
fetch_data(folder_path1, file_names, data, ["Runtime"])

raw_dataset = []

num_workers_min = 2
num_workers_step = 2
num_workers_values = range(num_workers_min, num_workers_max +
    num_workers_step, num_workers_step)
nwv_len = len(num_workers_values)
num_workers = num_workers_values[0]
...
for i in range(len(data)):
    num_workers = num_workers_values[i % nwv_len]
    num_cores = num_cores_values[(i // nwv_len) % ncv_len]

    if num_cores > greater_than_threshold:
        greater_than = True

    for j in range(1, len(data[i])):
        raw_dataset.append([32, num_cores, num_workers,
            data[i].values[j][0], greater_than])

raw_dataset = pd.DataFrame(raw_dataset)
raw_dataset.columns = ["Grid Size", "Num Cores", "Num Workers", "Time",
    "Num Cores Greater Than Physical Cores"]

sns.set_context("talk", font_scale=1.4)
ax = sns.factorplot(x="Num Workers", y="Time", hue="Num Cores", col="Num
    Cores Greater Than Physical Cores", size=10, scale=0.8, ci=95,
    data=raw_dataset);

axes = ax.axes.flatten()

axes[0].yaxis.labelpad = 50
axes[0].set_ylabel('Runtime\n(ms)', rotation=0)

if machine == "spa":
    axes[0].set_title("Num Cores 2-12")
    axes[1].set_title("Num Cores 14-24 (Including Hyperthreads)")
...
```

Figure C.1: An example of a data analysis notebook, for the optimal threads experiments. The notebooks for the contention experiments are too large to include. Taken from the file /data_analysis/year 2/optimal_threads/optimal_threads_merged.ipynb, which can be found in the project directory.