Design And Analysis Of A Plastic Parallel Programming System

Mark Jenkins

School Of Informatics
University Of Edinburgh
2017

Abstract

This is the first part of an MInf project which will span two years. In this report, we combine the ideas of contention aware scheduling, plastic programming, and skeleton programming, to create a library where incarnations of its skeletons will explicitly work together to share and optimize the use of system resources.

We perform multiple experiments assessing the performance of our library, with the outcome that we can obtain performance improvements over traditional parallel programming. We also discuss how it could be taken further, with both ideas for the next year of this MInf project, and for work to be done after this project is complete.

Contents

Chapter 1

Introduction

Throughout the history of computing, computer programmers and hardware engineers have exploited parallelism, with software and architectural innovation complementing technological improvements to provide increased performance. Architects have exploited bit and instruction parallelism, and now programmers are increasingly exploiting task/data parallelism in their applications [14] for various reasons [1].

It is recognised that writing correct and efficient parallel programs is hard [11], as the introduction of parallelism introduces a whole host of new problems [12], (e.g. unreproducible bugs due to race conditions, and difficulties debugging as high level instructions need to be decomposed into atomic assembly code to understand what is going on). The main purpose of multi-threading is improved performance, however parallel programs can often be slower then their serial counterparts, be it due to dependencies or hardware contention. Overhead must be balanced such that we don't introduce so much more work organizing threads and computations that our performance gains disappear.

Utilizing hardware efficiently is a major research challenge, especially considering that, today, a single program or library needs to deal with multiple different incarnations of the task it is trying to run. These arise from different circumstances, ranging from the hardware it is running on to the task size. Creating a "one size fits all" solution has become increasingly difficult, especially for high performance parallel applications. As such, there are solutions to help mitigate this problem [3], which utilize the idea of plastic programming, that is, changing the specifics of an implementation depending on the circumstances. However, such solutions are only "plastic" at compile time, if circumstances change during runtime they cannot adapt. In particular, the most common situation a program may encounter would be the sharing of the computer's resources with other programs. This is again exacerbated for high performance parallel pro-

grams, as they typically attempt to use all the resources they can get their hands on.

Even when we have an ideal parallel program, often resources are shared between multiple programs and users, leading to resource contention. This is the case for most use cases, from the serious computing resources in data centres, to the low powered hardware of mobile phones. Even in our own computing labs, we have messages to the effect of "Do not leave applications running on this machine" or "please nice your programs", which are often unseen or disregarded.

It is known that in such a situation with two programs, with careful selection of program parameters, we can obtain a better average runtime for both programs [7].

The aim of this project is to simplify the challenges of parallel programming and to provide improved performance by utilizing three key ideas:

- Co-Scheduling
- Plastic Programming
- Skeleton Programming

and to investigate the performance ramifications. Combining these ideas results in these problems becoming particularly tricky, with many different challenges involved in combining them.

The layout of the report is as follows; In chapter 2 we will provide an exploration of the ideas behind the project, and give a quick overview of how they are combined. In chapter 3, we will detail the thought process behind the design of our system, and then in chapter 4 we will then go into the specifics of the implementation of the system. Chapter 5 presents a carefully selected series of experiments to assess this system, and in chapter 6 we discuss the results of these experiments and their ramifications, and provide an overall conclusion to this year of the project. Chapter 7 provides the roadmap of future work for the second part of this two year MInf project and beyond.

Chapter 2

Background

In this chapter, we will detail the current approaches to parallel programming. We will then explore the three key ideas requisite to this project, such that we can discuss how they are combined and the implications.

The main new idea in this project is that of dynamic contention aware scheduling and optimization. It has been shown to be an important factor in multiprogramming systems with performance implications [7]. Plasticity is a technique to respond to this challenge, and take it further. This results in complex code, making it hard to ensure correctness. So we use skeletons to abstract this complexity away from the programmer. It also has the nice side effect of dividing the challenge into a pattern-by-pattern basis.

In this project we will produce such skeletons, and investigate the performance implications of these ideas, in order to evaluate whether they will have a significant effect.

2.1 Current Solutions

Current solutions for parallel programming include:

- Pthreads (POSIX Threads [4])
- MPI (Message Passing Interface [13])
- OpenMP (Open Multi-Processing [9] [6])

These are the more conventional methods of parallel programming.

Pthreads provides the highest level of fine grain control, leaving most of the work to the programmer. As such, some parallel processing techniques may only be possible with Pthreads. It is implemented in the POSIX library, and is well integrated with the standard GNU compiler collection (**gcc**). Pthreads only makes sense with shared memory architectures.

MPI is a communication library, and as such mostly details how processes can communicate. It is the dominant model used in the high performance computing industry today [13]. It can not only be used for a parallel program running on a single machine, but can also be used to implement a parallel program running on a distributed memory system.

OpenMP is comprised of compiler directives, runtime library routines, and environment variables. It is mostly used for loop parallelization, and is much higher level. It is also not limited to C. Like Pthreads, OpenMP is intended for shared memory architecture as it is thread bound.

Each of these methods have their own way of dealing with the complications introduced with parallel programming, which range 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 each of these particular parallel programming methods.

2.2 What Is Contention Aware Scheduling?

It is known that in multiprogramming systems, with many programs running simultaneously, the choice of program to socket mapping significantly affects the performance of the system [7]. Just considering two programs running on the same socket, we can see from the graph in figure 2.1 that certain programs perform differently with others, with some strange cases where the programs actually display better performance when running in contention with another. This problem is called co-scheduling.

So with this evidence, we can see that if we take into account such factors in our scheduler, we may obtain better overall performance. The outcome of the LIRA paper concludes that throughput gains of 3-7% can be seen. Socket/resource aware scheduling in this manner is called a contention aware scheduler. Adding in the plastic programming idea could make this particularly powerful, because we know and control the specifics of the implementations, and not only can we control what program runs where, we can also adjust the implementation the program is using.

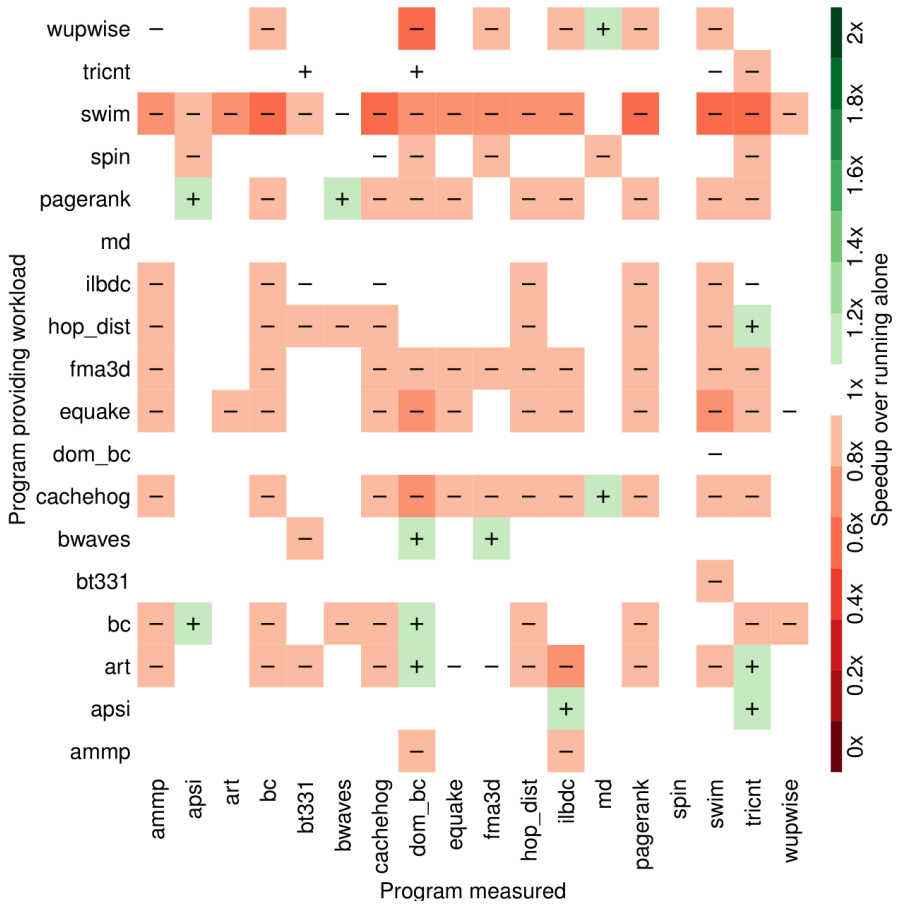

Figure 2.1: Taken from the LIRA paper [7]: Pair-wise speedup of programs, comparing sharing a socket to using separate sockets. Boxes annotated with a − indicate cases where performance decreased, and + where performance increased. In LIRA, this information was used to select which programs, from a given workset, should be scheduled concurrently, in order to maximize performance.

2.3 What Is Plastic Programming?

When programming an algorithm, there are often many choices about the specific implementation which can greatly affect performance, and the best choice depends on the circumstances of the problem. We tend to have more choices with parallel programs, but this is the case even for sequential programs. As an example, for a sorting problem with a large input size, radix sort would perform best, whereas for a small input size, insertion sort would be better. So naturally, in the interests of performance, we can conceive of a better overall implementation by combining the two approaches, so if the task size is large we would use radix sort, and then once it is reduced (for example, in a recursive algorithm,) we would use insertion sort.

An example of such a hypotheticaL situation can be seen in figure 2.2, which shows 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 the optimal decision points where a plastic programming system should switch algorithms.

Such compositions are commonplace, such as the sorting example discussed in the PetaBricks paper introduction [3]. Compared to PetaBricks, our implementation of plastic programming will be a little different. With the PetaBricks system, the programmer must specify multiple implementations for the compiler to switch between. In our system, we provide the various implementations in our library, meaning no extra effort is required from the programmer.

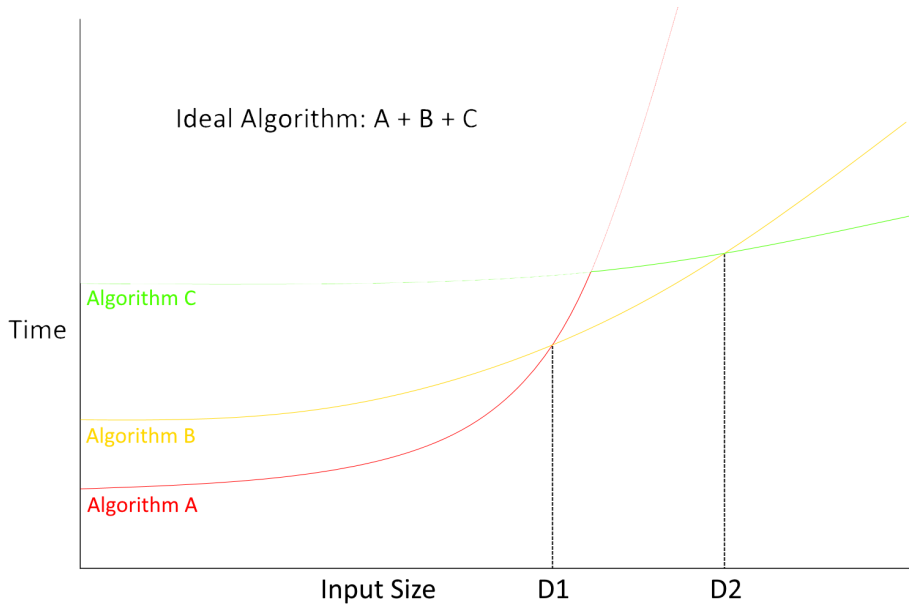

Figure 2.2: A graph showing three algorithms with different runtime curves, which depend upon the array size. Combining these algorithms would provide an improved algorithm, with D1 and D2 showing the optimal decision points where a plastic programming system should switch algorithms

2.4 What Is Skeleton Programming?

Skeleton programming is a high-level programming model. Skeletons allow us to abstract away all the complexity involved in parallel programming, plastic programming, and co-scheduling. The essence of skeleton programming is that the skeleton provides the core structure of an algorithm, the user provides some code (In our case, a function), which then produces a correct program for the task at hand [5]. The skeleton handles the hard-work of providing and optimizing the code (In our case, dealing with parallelism, plasticity, and co-scheduling).

The consequences of this are twofold:

- Errors are reduced substantially, as parallel programming is not easy, even without plasticity and contention aware scheduling.
- We can assess the program's complexity, since we know the algorithmic details of the skeleton.

Typically, multiple skeletons are combined to produce a more complex program, for example, a common combination is map and reduce (specific skeletons will be discussed in section 3.1.1.) The ability to combine skeletons makes them a powerful tool, allowing programmers to easily create clean complex programs.

Previous work to make parallel programming simple utilizing skeletons includes:

- SkePU [8]
- The Münster Skeleton Library Muesli [2]

These methods simplify the problem, with both utilizing skeleton programming to do so. The closest thing to our proposed system is SkePU, since each SkePU skeleton has multiple implementations, and SkePU automatically selects the expected fastest implementation variant. In contrast to our system, this selection is not dynamic, i.e. it does not change during runtime.

2.5 Summary

In this chapter, we described the three fundamental concepts behind this project, contention aware scheduling, plastic programming, and skeleton programming. With just contention aware scheduling, we can achieve greater performance [7]. In this project we will take this idea further, adding plastic programming, so we

can adapt the implementation of programs at runtime. This lets us change the implementations to ones which we know work well together. Combining these two ideas results in considerable complexity, so to make it simple for a programmer to use and easier for us to program, we use skeletons to abstract this complexity. The details of this complexity and how these ideas are combined are described in chapter 4.

The intent of this project is to explore the extent to which this approach can outperform a contention aware scheduled system which doesn't utilize plasticity. This is the plan for the first year of this MInf project, with the structure of the second year detailed in chapter 7.

Chapter 3

Design

In this chapter, we will discuss the design of the system created to investigate this problem, and detail how such a system could be extended for the future.

3.1 System description

The ideas described in the background section are combined to produce a skeleton programming library with plasticity and contention aware scheduling. To keep it simple, we build the system incrementally, starting with a single parallel pattern, later adding plasticity and contention aware scheduling.

3.1.1 Skeleton Foundation

As discussed in the background section, one of the key ideas behind the project is that of skeleton programming, using predefined patterns to aid the programmer. Possibly the most common skeleton is map, which is a skeleton that takes a function, an array of data, and applies the function to each member of the array. The map operation itself is inherently parallelizable, since each 'task' (processing one element of the input array) is independent, so we can simply assign different chunks of the input array to different threads. This is called an "embarrassingly parallel" problem.

Other such problems include map-array, reduce, and scan. These are all implemented in SkePU with a corresponding skeleton (as described in section 2.4.) In our case, we will focus on the map-array pattern, with further patterns left for possible future work. Map-array is similar to the map pattern in that it applies

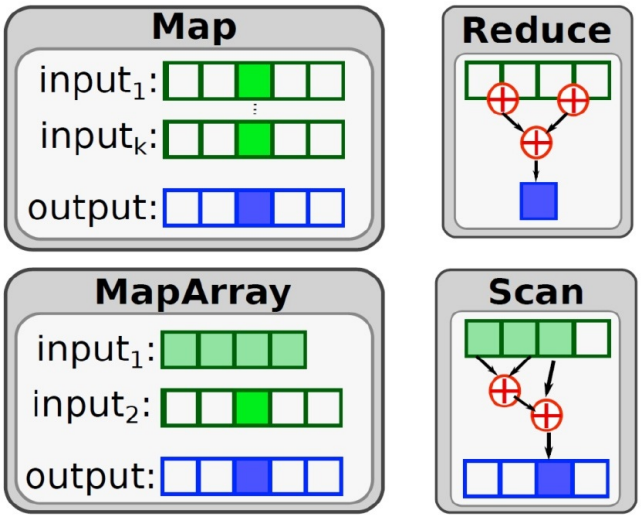

Figure 3.1: Taken from SkePU documentation: Examples of skeletons

a given user function to each element in a list, however map-array also allows the function to access a user provided array. A diagram to better explain this is given in figure 3.1.

Map-array was chosen as the map pattern is likely the most well known pattern and certainly one of the most useful, and map-array provides further functionality on top of this. It also provides a good basis for developing further patterns, and allows for complex testing, which will be covered in the evaluation section of this report.

3.1.2 Adding Plasticity

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

- 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

The thread count and pinnings are self explanatory. The schedule however requires some explanation.

The most basic method to divide the tasks is to give an equal amount to each thread. This is fine if the complexity of the tasks is uniform, but if it is skewed, the amount of computation to be done by each thread is imbalanced. This is illustrated in figure 3.2. This is because we have idle cores during computation, which is a wasted resource in a multi-threaded execution. However, if we allocate the tasks differently, we can obtain better performance, as illustrated in figure 3.3, where the improved schedule runs twice as fast as the simple one.

So load balancing a workload is critical to performance in such a multi-threaded application. However, optimizing the task distribution in this manner is non trivial, and it depends upon the computation to be done as well as the number of threads and other resources available at runtime.

A solution to this problem is to provide many different task distributions, and let the user pick or the machine select which distribution to use. OpenMP documentation calls these schedules, and some examples of these are:

- Static - An equal number of tasks allocated to all threads
- Dynamic individual - Each thread retrieves one task at a time, and once completed, it goes back for more
- Dynamic chunks - Each thread retrieves N tasks at a time, and once completed, it goes back for more
- Tapered - Each thread starts by retrieving N tasks at a time, and as the computation continues, it retrieves fewer and fewer

Thread count and schedule were chosen as they seem the most critical to performance, and thread pinnings was added as this is was investigated in the LIRA paper [7] as a factor contention aware scheduling could exploit.

Once we have added plasticity, we can experiment with the specifics of an implementation, and see how they affect the performance of the system. This would be the use case of utilizing our library with no other program running, (So no contention aware scheduling), and we can explore how we can adapt the program using plasticity at runtime in this case. We may be able to improve performance even under these conditions, depending upon the configuration of the machine (e.g., are there more CPU cores available) and the problem (e.g. do we have many small tasks or few large tasks.)

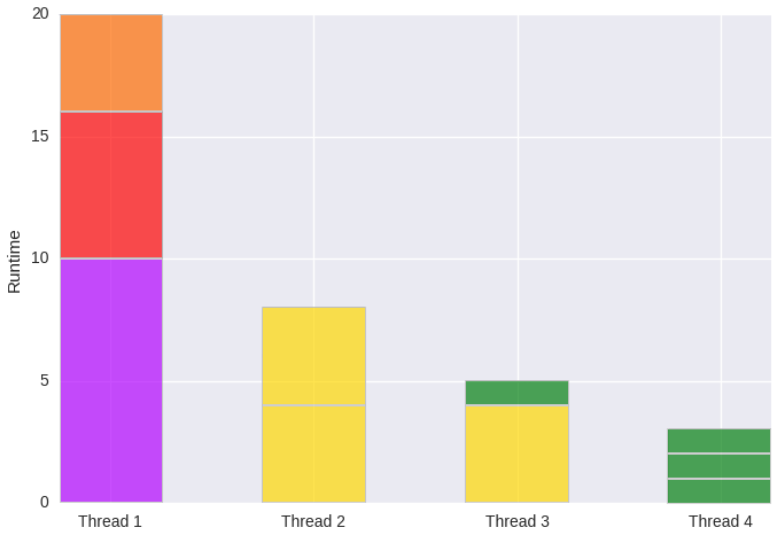

Figure 3.2: A worst case scenario of a static schedule assigning each thread an equal number of tasks. Suppose we have a set of twelve independent tasks with the following set of execution times (These would be unknown to the scheduler): {10, 6, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1}. With four threads, a simple division of tasks would be three tasks each distributed in order. This figure illustrates this distribution of work. Note that the total execution time is 20 time units.

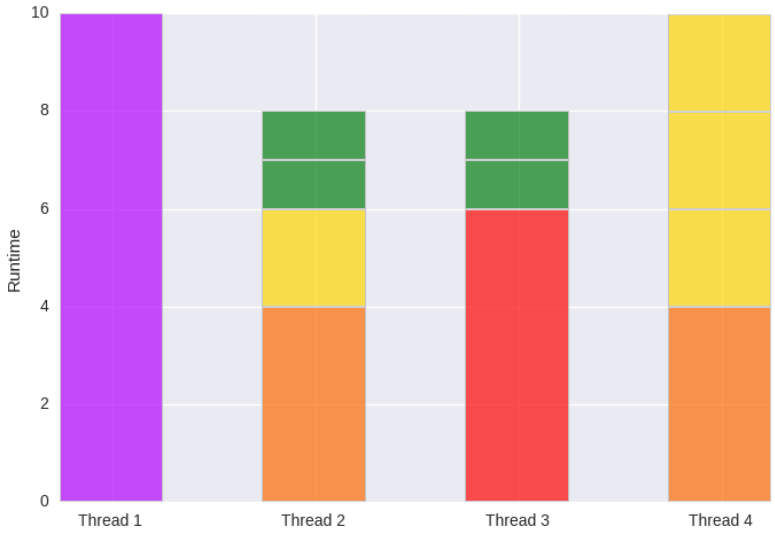

Figure 3.3: An optimized version of figure 3.2. Here we have the same set of tasks, but we have a better distribution of work, so the total execution time is 10 time units.

3.1.3 Contention Aware Scheduling

To add contention aware scheduling, we need multiple applications using our library to be able to collaborate, and adapt their behaviour accordingly. To do this, we use a separate controller application, with which all instances of our program can communicate. This provides a single known point of contact, and a designated thread for computing program parameters with respect to all aspects of the system.

Once our programs can communicate, and we can control each aspect of them, we can implement contention aware scheduling. In this phase of the project, we simply program a set of predefined actions for the controller to take, in order to manually control what each implementation does, as we are only investigating if this approach seems promising. We leave implementing some algorithm for automatic parameter tuning for future work.

A diagram showing the high level communication model is given in figure 3.4. This is expanded upon in section 4.3, showing specifics about the implementation.

Now that we have contention aware scheduling, we can experiment with multiple programs running on a system at once. An example of how contention aware scheduling can be enhanced with plastic programming is given in figure 3.5.

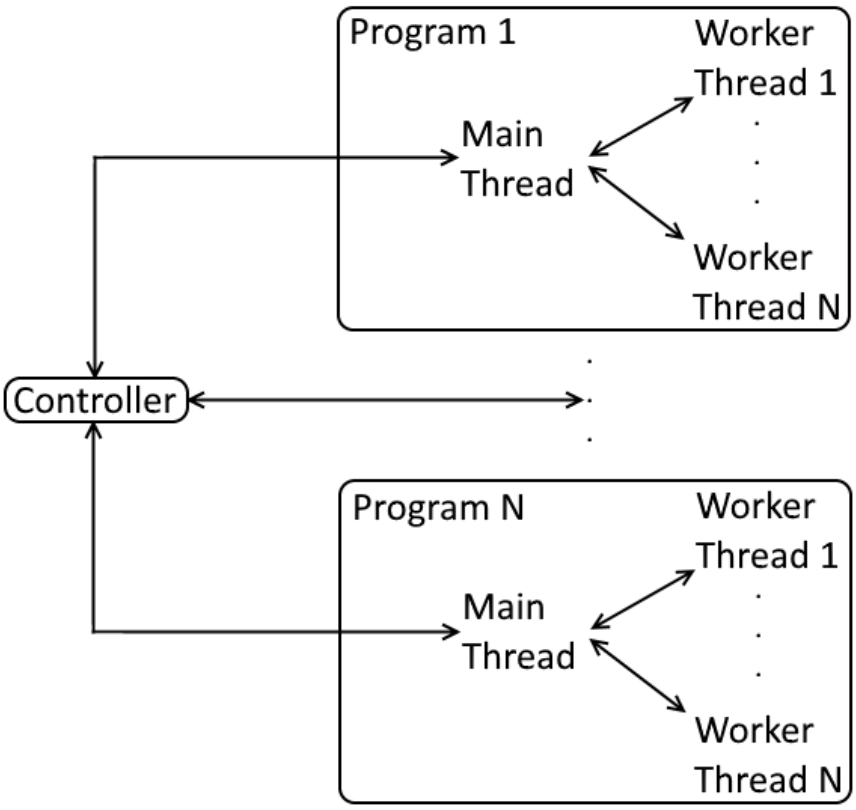

Figure 3.4: High level communication model of the system, with an arbitrary number of programs, with an arbitrary number of threads. Two way communication occurs between the controller and each main thread, and then between each main thread and it's worker threads.

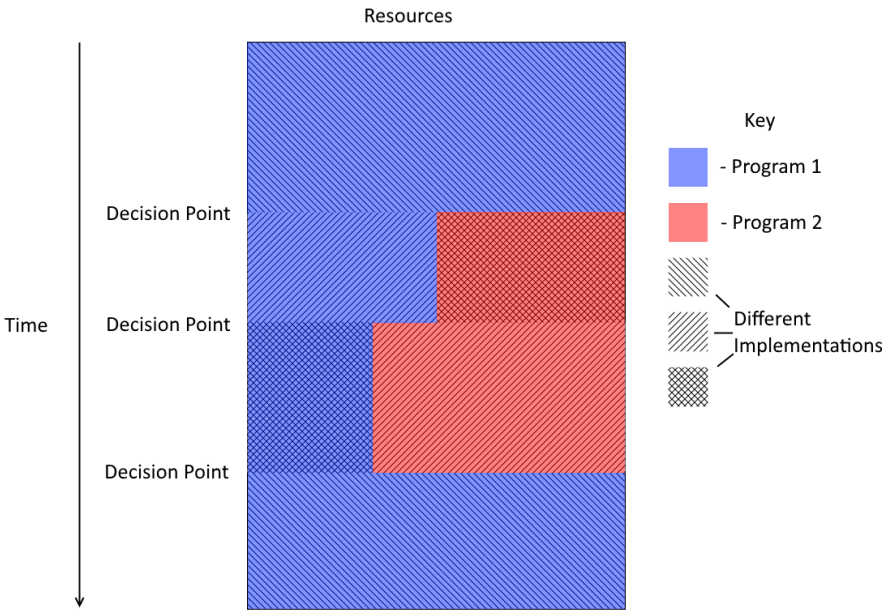

Figure 3.5: An example of how contention aware scheduling can be enhanced with plastic programming. We have two programs, represented by different colours. As time progresses, we see that when program 2 launches, we have a decision point. Here, the system decides how many resources to give each program and what implementation they should use, according to what is optimal (In this project, we do this manually.)

Moving further in time, we have another decision point. This one is triggered by some change in the machine, which means the optimal configuration has changed. So the system updates the configuration of each program, and continues.

Program 2 then terminates, triggering another decision point, and the system again updates the configuration.

3.1.4 Evaluation

To properly evaluate the outcome of this project, we need some way of testing the libraries performance. To this end, we implemented a synthetic program which calls the skeleton with an artificial workload, collecting and recording metrics detailing the libraries performance with different parameters.

We also need points of comparison in terms of performance. So, in addition to our synthetic program, we also implement an equivalent sequential and an OpenMP version, in which we can vary similar parameters and produce comparable statistics.

The detailed experimental program will be discussed in 5.

Chapter 4

Implementation

In this chapter, we present specific implementation details of the project, providing relevant code highlights,and cover the problems that occurred and how they were solved.

We start with the implementation of a basic skeleton, then we cover adding plasticity, and contention aware scheduling. Finally, we finish by covering the applications developed to evaluate the project.

C++ was chosen as a basis, since it's fast, and provides language constructs such as templates and overloading. Another, personal, reason is that I wished to learn something new, as I had not used C++ before.

The parallel backend that the system is based upon is Pthreads due to its wide availability, and the level of fine control. It allows us to tune all parameters of the program and implement functions which are not possible with other solutions, e.g. detailed metric analysis.

4.1 Skeleton Foundation

The first skeleton we will implement is the map-array skeleton, as described in section 3.1.1. It is reasonably straight forward to create a sequential skeleton, using c++ templates to create a templated function which takes a function amongst other things as it's arguments. This will be our skeleton. The interface of our skeleton is given in figure 4.1, and a usage example in figure 4.2.

```
1
2   template <typename in1, typename in2, typename out>
3   void map_array(deque<in1>& input1,
4                  deque<in2>& input2,
5                  out (*user_function) (in1, deque<in2>),
6                  deque<out>& output,
7                  string output_filename = "",
8                  parameters params = parameters())
```

Figure 4.1: Interface of our 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.

```
1   int user_function(int in1, deque<int> in2) {
2      return in1 + in2[in1];
3   }
4
5   int main() {
6      // Inputs.
7      deque<int> input1(ARRAY_SIZE);
8      deque<int> input2(ARRAY_SIZE_2);
9
10     // Put data in inputs.
11
12     // Output.
13     deque<int> output(ARRAY_SIZE);
14
15     // Start mapArray.
16     map_array(input1, input2, user_function, output);
17
18     // Record output.
19  }
```

Figure 4.2: 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.

Parallelizing this presents two problems, namely:

- How to divide tasks amongst threads (In map array, one task is one application of the user function to an element of the input array)
- How many threads should be used

In a basic skeleton, both of these parameters must be specified beforehand and will not change at runtime. The ability to vary them would provide the different implementations necessary for the plasticity portion of this project.

To implement multiple schedules, we use a bag of tasks object. A bag of tasks is a collection of independent, usually similar tasks which are to be executed. It is a model which is usually combined with some form of parallelism.

Each thread will be given the location of the same bag, and will retrieve a specified number of tasks at a time from the bag. Since each schedule essentially consists of retrieving a different number tasks, we can use this to implement the basic schedules (Static, Dynamic chunks, Dynamic Individual). More complex schedules can easily be added in the future, as they can reuse the get tasks method, an simply adjust how many tasks each thread retrieves at one time.

The bag also provides the main source of inter-thread communication using shared memory, and contains various semaphores for controlling the threads. This will be expanded upon in sections 4.2 and 4.3.

This bag of tasks object also gives us the basis for further extensions, such as using multiple bags, adding task stealing, or adding tasks to the bag during computation.

Providing a variable number of worker threads is simple now that we have the bag of tasks, all we need to do is to adjust our initial calculations when calculating how many tasks each thread should receive according to the schedule. These worker threads will be spawned by the original thread, the "main" thread. It will block until all computation is complete, it will then join with each of the worker threads.

Figure 4.3 shows how we create our worker thread using Pthreads, and figure 4.4 hows how we can later join with them.

```
1   template <typename in1, typename in2, typename out>
2   void map_array(. . .) {
3     . . .
4     // Calculate info for data partitioning.
5     deque<thread_data<in1, in2, out>> thread_data_deque = ↩
          calc_thread_data(bot.numTasksRemaining(), bot, params);
6
7     // Variables for creating and managing threads.
8     deque<pthread_t> threads(params.thread_pinnings.size());
9
10    // Create all our needed threads.
11    for (uint32_t i = 0; i < params.thread_pinnings.size(); i++)↩
          {
12      int rc = pthread_create(&threads.at(i), NULL, ↩
            mapArrayThread<in1, in2, out>, (void *) &↩
            thread_data_deque.at(i));
13
14      // Create thread name.
15      char thread_name[16];
16      sprintf(thread_name, "MA Thread %u", i);
17
18      // Set thread name.
19      pthread_setname_np(threads.at(i), thread_name);
20
21      if (rc) {
22        // If we couldn t create a new thread, throw an error ↩
              and exit.
23        print("[Main] ERROR; return code from pthread_create() ↩
              is ", rc, "\n");
24        exit(-1);
25      }
26    }
27    . . .
28  }
```

Figure 4.3: Parallel portion of our map_array skeleton. This figure shows how we use Pthreads to create our worker threads, providing each with access to it's own thread data object.

```
1  // Joins with given number of threads.
2  void join_with_threads(std::deque<pthread_t> threads, uint32_t↩
       num_threads_to_join) {
3    int inital_threads_max_index = threads.size() - 1;
4
5    for (uint32_t i = 0; i < num_threads_to_join; i++) {
6      // Join with thread
7      int rc = pthread_join(threads.back(), NULL);
8
9      // Update thread tracking variables.
10     threads.pop_back();
11
12     if (rc) {
13       // If we couldn t join with the thread, throw an error ↩
             and exit.
14       print("[Main] ERROR; return code from pthread_join() is ↩
             ", rc, "\n");
15       exit(-1);
16     }
17   }
18 }
```

Figure 4.4: Function which joins with the specified number of threads. Starts with the last thread to be created, so that if we wish to terminate threads, the threads which stay executing preserve their logical ordering.

4.2 Adding Plasticity

We already have a sort of compile time plasticity in our system, in that we can choose some parameters of the skeleton before compilation. We can choose the number of worker threads and the schedule used. The other main aspect we would like to control is what CPU core each thread executes on. This is called processor affinity or thread pinning. This was added to the skeleton by adding control variables to the bag of tasks, which control the CPU affinity of each thread. Each thread then simply sets it's affinity to the intended CPU, using the function in figure 4.7.

To add runtime plasticity, we need to be able to change the implementation of the skeleton on the fly. The most straightforward method of achieving this is to stop all computation, terminate worker threads, update the parameters, and resume computation. In a basic skeleton, (as described at the end of section 4.1,) the main thread would block until the computation is complete, and then would join with each of the worker threads. We change this so that, instead of blocking, it monitors the computation and can instruct the threads to terminate at any point (using the afore mentioned bag of tasks described in section 4.1 for communication. This functionality is shown in figure 4.6). An overview of the code used to implement the process can be seen in figure 4.5.

When this switching is done and what the new parameters should be is another matter. A future system would calculate these things dependent upon the current state of the system, but for our preliminary investigation, we manually produce them in a synthetic environment, and communicate them to the main thread using the controller application, as described in section 4.3.

An obvious optimization of this system would be to modify the implementation without the need to terminate threads. This graceful switching of the implementation is left for future work, as this optimization is very complex, and would likely result in a marginal speedup proportional to the number of times we wish to change the implementation. We don't plan on switching implementations excessively, and any delay added could be overcome in testing by increasing the input size, and this project is only investigating if this approach to parallel programming is promising.

Another optimization would involve modifying the static schedule. In our implementation, worker threads only check if they should terminate when they contact the bag for more tasks. Since the static schedule means they only contact the bag at the start of the computation, once they have started, they cannot be interrupted. To optimize this, and make a "plastic" static schedule, we could change the communication method so that worker threads can be interrupted. This hasn't been necessary for our current work, and as such has been left for future work.

```
1   template <typename in1, typename in2, typename out>
2   void map_array(. . .) {
3     . . .
4     while (bot.empty == false) {
5
6       // Get message from controller.
7
8       // Terminating threads.
9       bot.thread_control.assign(params.thread_pinnings.size(), ↩
            Terminate);
10
11      // Joining with threads.
12      join_with_threads(threads, params.thread_pinnings.size())↩
            ;
13
14      // Update parameters.
15
16      // Restart map_array.
17
18    }
19    . . .
20  }
```

Figure 4.5: An example of how the main thread handles a switch in the implementation.

```
1  // Thread control enum. Threads either run (execute), change ↩
       strategies (update), or stop (terminate).
2  enum Thread_Control {Execute, Update, Terminate};
3
4  template <class in1, class in2, class out>
5  class BagOfTasks {
6    public:
7      // Variables to control if threads terminate.
8      deque<Thread_Control> thread_control;
9
10     // Check for if bag is empty.
11     bool empty = false;
12     . . .
13
14     // Returns specified number of tasks or less.
15     tasks<in1, in2, out> getTasks(uint32_t num) {
16       // Get mutex.
17       lock_guard<mutex> lock(m);
18      . . .
19
20      // Calculate number of tasks to return.
21      . . .
22
23      if (num_tasks == 0) {
24        empty = true;
25       }
26      . . .
27    }
28 };
```

Figure 4.6: The inter-thread communication sections of the bag of tasks object. thread_control provides a control variable for each thread, which it reads and reacts to. empty boolean is updated when the getTasks method is run, and is read periodically by the main thread to check if we have completed the computation.

```
1  // Sticks current thread to given CPU core.
2  int stick_this_thread_to_cpu(uint32_t core_id) {
3     uint32_t num_cores = sysconf(_SC_NPROCESSORS_ONLN);
4
5     // Check for valid core_id.
6     if (core_id >= num_cores) {
7        return EINVAL;
8      }
9
10    cpu_set_t cpuset;
11    CPU_ZERO(&cpuset);
12    CPU_SET(core_id, &cpuset);
13
14    pthread_t current_thread = pthread_self();
15
16    return pthread_setaffinity_np(current_thread, sizeof(↩
          cpu_set_t), &cpuset);
17 }
```

Figure 4.7: Function for sticking a thread to a particular CPU core. Enables thread pinning plasticity for each thread.

4.3 Contention Aware Scheduling

As detailed in section 3.1.3 of the design chapter, we will use a separate controller application to co-ordinate and control all programs using our library. This simplifies the implementation significantly, as it gives us a single point of contact, and a single place to calculate an optimal configuration for each program. To achieve this, we need our separate controller application, and two main additions to our system:

- Inter-process communication
- Separate communication/control thread for each program

We use the ZeroMQ library to provide inter-process communication for it's speed, although it could be replaced with another method. It uses tcp sockets, and serves to simplify inter-process communication. Currently I have made simple use of ZeroMQ for now, and may optimize it in the future. For inter-thread communication (between a main thread and worker threads), we use shared memory, again for speed and simplicity.

In order to communicate with the controller while a program is using our library, we utilize our main thread, which will manage the worker threads, switch implementations when instructed, and will clean up after we have finished our computation. The main thread communicates with the controller, registering with it when we start and de-registering once complete. During computation, it listens to messages from the controller using non-blocking communication. This allows us to also check if we are finished processing. For the initial registration and de-registration, we use blocking communication, as these messages are vital to the system. This system is illustrated at a high level in figure 4.8, which expands upon figure 3.4, showing implementation details. Figure 4.9 provides an example of the code used to perform this, and shows the use of the ZeroMQ library.

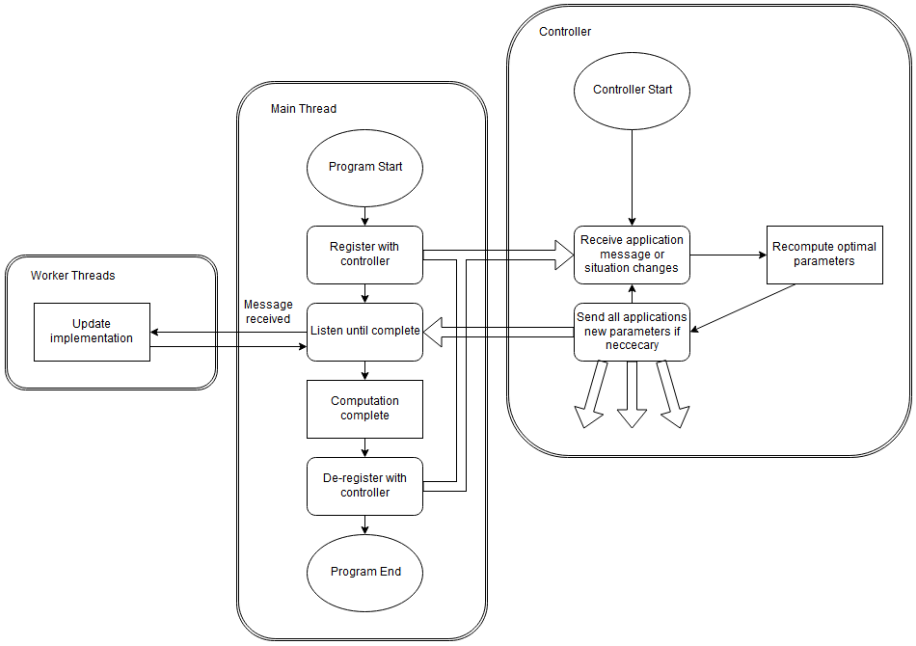

Figure 4.8: Communication model for communications between applications and the controller. Thin lines represent program flow, thick lines represent inter-process messages.

```
template <typename in1, typename in2, typename out>
void map_array(. . .) {
  . . .
  // Get our PID to send to the controller.
  uint32_t pid = pthread_self();

  // Prepare our context and socket
  context_t context(1);
  socket_t  socket(context, ZMQ_PAIR);
  socket.connect("tcp://localhost:5555");

  print("\n[Main] Registering with controller...\n\n");

  // Create registration message.
  struct message rgstr;
  rgstr.header                    = APP_REG;
  rgstr.pid                       = pid;
  rgstr.settings.schedule         = params.schedule;

  fill_n(rgstr.settings.thread_pinnings, MAX_NUM_THREADS, -1);
  copy(params.thread_pinnings.begin(), params.thread_pinnings.↩
      end(), rgstr.settings.thread_pinnings);

  // Send registration message.
  m_send(socket, rgstr);

  while (bot.empty == false) {
    usleep(1);

    // Get message from controller. (Use a non-blocking ↩
        receive so we can still check if bot.empty == true)
    struct message msg = m_no_block_recv(socket);

    if (msg.header == APP_UPDATE) {
      print("\n[Main] Received new parameters from controller↩
          !\n\n");

      // Update parameters.
    }
  }

  // Create termination message.
  struct message term;
  term.header = APP_TERM;
  term.pid = pid;

  // Send termination message.
  m_send(socket, term);

  socket.close();
```

Figure 4.9: An example of how the main thread implements a change to the parameters received from the controller. Demonstrates the usage of the ZeroMQ library to provide inter-process communication with the controller.

4.4 Testing Programs

To evaluate our library, we need programs to run and test the library under differing conditions, and we need programs to represent competing approaches to parallel programming.

The main map-array test application is a tool which can read experiment parameters from a configuration file, and then run each experiment in sequence, recording various metrics for later analysis. This provides a convenient framework for carrying out experiments, and easily enables us to queue up a set of experiments. This is important because such experiments can take hours.

For meaningful analysis, we need to synthesize a workload. One which we can scale, so we can test different sized tasks and varying task distributions. To do this we use the Collatz function to generate a CPU intensive workload. A constant starting number is used, and the sequence is repeated multiple times, to scale the workload.

The comparison programs consist of a purely sequential implementation, and an OpenMP implementation. These were chosen because the former represents an implementation a traditional sequential programmer would use, and OpenMP because it is a popular method of parallelizing code, with a focus on performance and a simple interface. They are set up such that they can use the same synthetic workload we use for testing our library.

The final tool we constructed for evaluation is a python program which was used to create graphs from the generated data.

4.5 Supporting Code

This project required complex code to achieve it's goals. In this section, we present other extracts of code, demonstrating it's complexity.

Basic functions must be reworked when we introduce parallelism to a program, let alone plasticity and contention aware scheduling. Figure 4.10 demonstrates a thread safe method for printing, by enforcing a mutex to be obtained before printing.

Figure 4.11 shows how multiple programs were launched for testing. This was used in the testing framework, in experiment 5 for launching the additional programs at regular intervals.

```cpp
// .hpp file:

// Print function entry point. Defaults to the cout output stream.
template <class ...Args>
std::ostream& print(const Args& ...args) {
    std::lock_guard<std::mutex> _(get_cout_mutex());
    return print(std::cout, args...);
}

// Retrieve mutex. This must be done using this non-templated function, so that we have one mutex instance across all instantiations of the templated functions.
std::mutex& get_cout_mutex();

// Case when given an output stream.
template <class ...Args>
std::ostream& print(std::ostream& outS, const Args& ...args) {
    return print_rec(outS, args...);
}

// Recursively prints arguments.
template <class A0, class ...Args>
std::ostream& print_rec(std::ostream& outS, const A0& a0, const Args& ...args) {
    outS << a0;
    return print_rec(outS, args...);
}

// With nothing to add, just return the stream.
std::ostream& print_rec(std::ostream& outS);

// .cpp file:

// With nothing to add to the output stream, just return the stream.
std::ostream& print_rec(std::ostream& outS) {
    return outS;
}

// Retrieve mutex.
std::mutex& get_cout_mutex() {
   // Static, so that we have one mutex instance across all instantiations of the templated functions.
    static std::mutex m;
    return m;
}
```

Figure 4.10: A thread safe print function. Prints a variable number of and a variable type of arguments to an output stream. Implementation is split accross .hpp and .cpp files, as the need for templates requires that certain functions must be defined in the header, so the compiler can see them during linking.

```c
// Example usage:

const char *args[64] = {"../bin/program_2_map_array_test" , "../build/configs/config7.ini" , NULL};
 int out = exec_prog(args);

static int exec_prog(const char **args) {
    pid_t pid;
    int   status, timeout;

    if ((pid = fork()) == 0) {
        if (execve(args[0], (char **)args , NULL) == -1) {
            perror("fork() failed [%m]");

            return -1;
        }
    }


    timeout = 1000;

    while (waitpid(pid , &status , WNOHANG) == 0) {
            if (--timeout < 0) {
                    perror("timeout");

                    return -1;
            }

            sleep(1);
    }

    printf("%s WEXITSTATUS %d WIFEXITED %d [status %d]\n\n", args[0], WEXITSTATUS(status), WIFEXITED(status), status);

    if (WIFEXITED(status) != 1 || WEXITSTATUS(status) != 0) {
            perror("%s failed, halt system");

            return -1;
    }

    return 0;
}
```

Figure 4.11: A function for executing an external program, used in the setup for experiment 5.

Chapter 5

Experimental Methodology And Program

The primary goal of this project is to investigate if we can achieve better performance using our approach to parallel programming compared to other methods. Accordingly, we will focus our experiments on evaluating the performance of the system. To this end, we have created a carefully designed set of experiments to asses the system under different conditions, conditions which may be found in future applications. In this chapter we describe our evaluation methodology and the particular experiments we will run. The outcome of each experiment will be discussed in the results chapter in the same order they are presented here.

5.1 Evaluation Methodology

All experiments were run on an Ubuntu virtual machine, with four cores at 4GHz, 4096MB of memory, and the rest of the system at idle. All programs were compiled at optimization level 3, and every result averaged over 50 repeats.

Multiple different statistics can be collected for each thread:

1. Total runtime
2. Time spent doing work
3. Time spent in overhead
4. Time blocked by main thread

5. Number of tasks completed

In these experiments however, we will focus on total runtime, as we are mostly concerned with the overall performance. The other statistics were mostly used during development.

5.2 Programme of Experiments

In this section, we detail each experiment, with the thought process behind it, how it will be executed, and the expected results if appropriate. The actual results will be presented in chapter 6, where we will compare them with our expectations and discuss their implications.

In each experiment we will vary a different set of parameters. These parameters can be partitioned into three separate classes; Application Input Parameters, Resources Granted, and Skeleton Parameters. More parameters can be added, (e.g. different resources granted,) but this is left for future work, as currently we only want a preliminary investigation into these ideas. The parameters we can change are:

Application Input Parameters:

- Number of tasks - The number of tasks to be carried out
- Task grain - The complexity of the tasks, possible values are small, medium, and large. Large is 10x the complexity of medium, and medium 10x the complexity of small.
- Task grain distribution - If we have range of task grains, this describes how they distributed throughout the input array. Possible values are uniform (evenly distributed throughout,) and biased (The more complex tasks are gathered together at the start of the array. This is the worst case scenario for the static schedule.)

Resources Granted:

- Number of CPU cores - The amount of CPU cores provided in the machine.

Skeleton Parameters:

- Number of threads used - The number of threads our program uses for multi-threading.

- Thread pinning - The configuration of threads pinned to CPU cores.
- Schedule - How the tasks are distributed amongst threads. See section 3.1.2 for the possible schedules.

Our first experiment will test how much overhead our metrics collection adds. Experiment 2 then establishes the baseline performance of our library, comparing it to see if it is on a par with an OpenMP solution. Next in experiment 3, we investigate the overhead that has been introduced by contention aware scheduling and plasticity. Experiment 4 then shows the importance of choosing the correct schedule, and how our system could deal with a suboptimal schedule using plasticity. This is again compared to an OpenMP solution.

Thus far, experiments 1-4 have all been conducted with a single application utilizing our library. In experiment 5, we investigate a situation with multiple applications using our library. We investigate the performance of the system with many applications competing for resources, with multiple strategies for how to share them.

5.2.1 Experiment 1 - Metrics Collection Overhead

The aim of our first experiment is to investigate if our method for collecting metrics about the system adds any significant overhead that we must take into account in assessing further experiments. To this end, we compare the runtimes of our system both with and without metrics collection, to see if there is any significant difference. The system is run with no controller application or runtime plasticity, since we only wish to measure the overhead added by our metrics collection.

The experiment parameters are designed such that it is a worse case scenario, and as a result would produce the most significant difference in runtimes. This worst case scenario would be many small tasks, with multiple threads, since metrics functions are called before and after tasks are completed, and are managed on a per-thread basis.

Since our metrics collection consists of some function calls and updating tallys, we don't expect the performance impact to be significant.

Number of tasks	100,000
Task Grain	100,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Dynamic Chunks (Chunk Size = 1000)

Table 5.1: Experiment 1 Parameters

5.2.2 Experiment 2 - Absolute Performance

The purpose of this experiment is to gauge the absolute performance of our system compared to other approaches. This should test if our approach has significantly more overhead than other approaches, and also if the underlying implementation is correct such that performance is on a par with competing approaches. To provide these points of comparison, we will test against a purely sequential approach, and another using OpenMP. We will also vary the schedule, again to verify that our implementation is on a par with a current parallel programming method (OpenMP) with different schedules.

Note here that the Static schedule and the Dynamic Chunks schedule with a chunk size of 1 represent opposing ends of an overhead spectrum. With a static schedule, we have the least amount of overhead possible, (good for when the task variance is uniform), and with the Dynamic Chunks schedule and a chunk size of 1, we have the most amount of overhead possible, but since we fetch tasks one at a time, we will have the best task complexity balance across our threads (good for skewed/high variance.)

As for the expected outcome, we would predict that we would be close to a purely sequential implementation in single threaded circumstances, if we use a static schedule, as the only additional overhead would be in managing the one thread. For other schedules, we may see a significant performance drop, since we would be using a sub-optimal schedule. But, OpenMP would likely see a similar performance drop with a sub-optimal schedule.

The main question we want to answer is how we compare to a current parallel programming approach, OpenMP. Do we scale with thread/CPU core count in a similar fashion? I would expect that we may be behind OpenMP in terms of performance, as OpenMP is a huge, well established project, and as such may be much more optimized than our implementation. Whilst the outcome of this experiment will no doubt be useful, we are also concerned with whether we can improve upon our own performance by adding plasticity and contention aware scheduling, so we need only compare against ourselves in experiment 5.

Number of tasks	10,000
Task Grain	1,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	1, 2, 3, 4
Thread pinning	Uniform
Schedule	Static

Table 5.2: Experiment 2 Parameters

5.2.3 Experiment 3 - Plasticity And Contention Aware Scheduling Framework Overhead

This experiment is designed to investigate whether the two main additions to a simple skeleton incur any significant overhead. That is, plasticity and contention aware scheduling. To do this, we will compute the overhead of communicating with the controller (contention aware scheduling) and switching strategies (plasticity), without actually changing any details of the implementation, so that any performance difference will be attributable to communication and switching overhead. We switch once each time, after the controller receives a registration message, it replies with a message triggering the switch.

We will then run the same experiment with no controller or plasticity, as in experiment 1. The runtimes of both cases will be compared to see the total amount of delay introduced.

Whilst we expect that we will incur some overhead, we don't expect it to be significant, as it will be of constant time and not scale with regards to input size.

Number of tasks	1,000 10,000 100,000
Task Grain	1.000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Dynamic Chunks

Table 5.3: Experiment 3 Parameters

5.2.4 Experiment 4 - Schedule Choice Importance

This experiment is designed to highlight the importance of the choice of schedule, and the benefits plasticity can bring even with just a single application. To do this, we compare the runtimes of different schedules with a biased task grain distribution. With a biased task distribution, we should see the static schedule perform the worst, and the dynamic individual or dynamic chunks schedule perform the best.

For the plasticity portion of this experiment, we will also add cases where we start with a sub-optimal schedule, and then, after 5 seconds, change to a better schedule. This will be done with a uniform schedule, starting with the dynamic individual schedule, and switching to the static schedule.

Number of tasks	100,000
Task Grain	10,000,000 and 100,000 Repeats
Task Grain Distribution	Biased
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Static, Dynamic Chunks (Chunk Size = 1,000), Dynamic Chunks (Chunk Size = 1) Static, then dynamic chunks after 5 seconds

Table 5.4: Experiment 4 Parameters

5.2.5 Experiment 5 - Absolute Multiprogramming Performance

The purpose of this experiment is to investigate the absolute performance of our library in a situation with multiple applications. We will start with the controller and one application running, we then introduce a second application, causing the first to modify it's parameters. The second application will then complete it's computation, again causing the first to modify it's parameters. Finally, the first application will finish, completing the experiment.

We compare two situations, one simulating a typical multiprogramming system, where we run the programs as they would be without any plasticity or contention aware scheduling. The other situation will be including plasticity and contention aware scheduling.

To get an idea of the performance, we will compare the runtimes of the two programs, with the runtime of the first application representing the total runtime of the entire experiment. We would expect the case with plasticity and contention aware scheduling to fare better, since the overhead isn't too large, and we would enforce that the total number of threads is equal to the number of CPU cores. This means that there is less unnecessary thread switching, as the OS switches between all threads.

Program 1	
Number of tasks	30,000
Task Grain	10,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4 4, then 2, then 4
Thread pinning	Loose, Uniform
Schedule	Dynamic_individual
Program 2	
Number of tasks	15,000
Task Grain	10,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4 2
Thread pinning	Loose, Uniform
Schedule	Dynamic_individual

Table 5.5: Experiment 5 Parameters

Chapter 6

Results And Conclusion

In this chapter of the report, we discuss the results of the experiments and their ramifications. The structure of this section mirrors that of the experiments section, so that the first results discussed will be the first experiment detailed in chapter 5. The error bars on the graphs show the standard deviation, and all time measurements are in milliseconds.

6.1 Experiment Results

6.1.1 Experiment 1 - Metrics Collection Overhead

In experiment 1, we investigated our method for collecting metrics about the system adds any significant overhead that we must take into account in assessing further experiments. Looking at the following results, particularly the percentage of total runtime, the performance impact of metrics collection isn't too severe. Still, detailed metrics were disabled for future experiments.

Of course, in a production system, we wouldn't need any metrics collection.

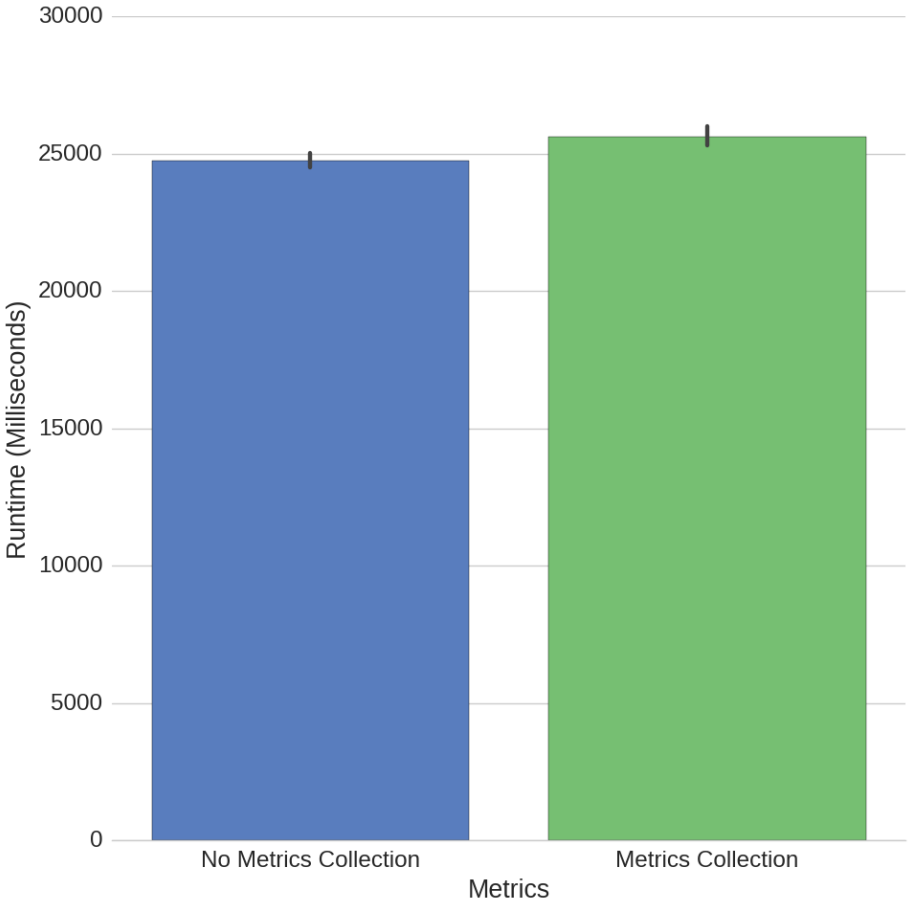

Figure 6.1: Experiment 1 results

Average delay added	877ms
Percentage of total runtime	3.5%

Table 6.1: Experiment 1 Results Analysis

Number of tasks	100,000
Task Grain	100,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Dynamic Chunks (Chunk Size = 1000)

Table 6.2: Experiment 1 Parameters

6.1.2 Experiment 2 - Absolute Performance

In this experiment, the total run times of three implementations are measured. One sequential implementation, a standard modern parallel implementation utilizing OpenMP, and our plastic implementation of map_array. Our plastic implementation is running with no plasticity for the moment, and with no messaging functionality at all. This is so it is comparable to a standard parallel implementation, since the purpose of this experiment is to test our baseline performance.

Both OpenMP and our implementation are using a dynamic chunks schedule, with a chunk size of 1000.

Figure 6.2 shows us that with a single thread, our performance is similar to a sequential implementation, and as we increase the thread count, our performance scales accordingly. Overall, this shows that our implementation performs on a par with current parallel implementations, providing a good baseline performance.

Number of tasks	10,000
Task Grain	1,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	1, 2, 3, 4
Thread pinning	Uniform
Schedule	Static

Table 6.3: Experiment 2 Parameters

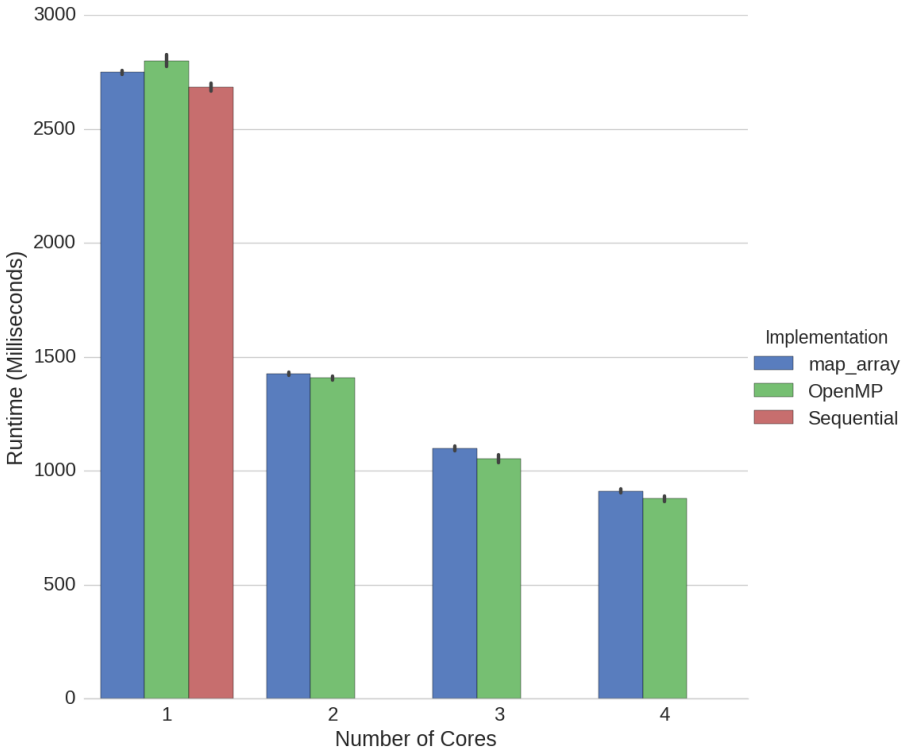

Figure 6.2: Experiment 2 results

6.1.3 Experiment 3 - Plasticity And Contention Aware Scheduling Framework Overhead

This experiment was designed to assess the amount of overhead added by plasticity and the contention aware scheduling framework. The results are strange, because we would expect the runtime to only increase with the added overhead, but we have two cases where the runtime decreases.

With our smallest array size of 1000, we see an increase of 2ms, which is 2.7% of the total runtime. These results are very close, and within the margin of error.

Our other results both show a decrease in runtime, again both within the margin of error. Repeat experiments to reduce the variance would help, and running them on a VM may introduce more variance than in a non-virtualized system.

The overall takeaway from this experiment, is that the added overhead is negligible when measuring the total runtime, with a wide selection of array sizes.

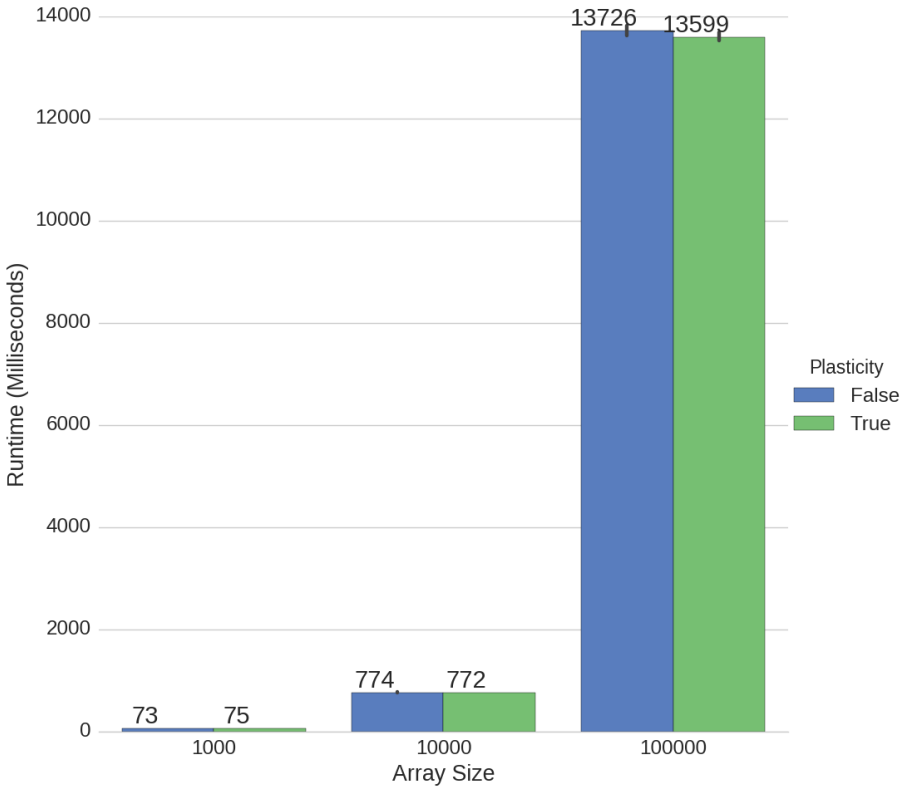

Figure 6.3: Experiment 3 results

Average delay added (Few tasks)	2ms
Percentage of total runtime (Few tasks)	2.7%
Average delay added (Med. tasks)	-2ms
Percentage of total runtime (Med tasks)	N/A
Average delay added (Many tasks)	-127ms
Percentage of total runtime (Many tasks)	N/A

Table 6.4: Experiment 3 Results Analysis

Number of tasks	1,000 10,000 100,000
Task Grain	1.000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Dynamic Chunks

Table 6.5: Experiment 3 Parameters

6.1.4 Experiment 4 - Schedule Choice Importance

This experiment was designed to highlight the importance of choosing the correct schedule, and how we can benefit from runtime plasticity. We have a biased set of tasks, (first quarter of which are large tasks, the rest small,) meaning the static schedule will perform poorly, and the dynamic schedules will fare well.

This is reflected in the results, where we see the dynamic schedules perform almost three times as well as the static schedule. Also included is a case where we start with the static schedule, and then switch to dynamic chunks after five seconds. We see it perform on a par with our dynamic schedules, again greatly improving upon the static schedule.

There is a small difference between our dynamic chunks and dynamic individual schedules. With a chunk size of 1,000, our dynamic individual schedule accesses the bag of tasks 1,000 times as often. Considering that it is implemented in shared memory, it makes sense that this penalty isn't large. If retrieving tasks was more costly, it would affect the dynamic individual schedule proportionately more than the dynamic chunks schedule. We also have a variety of task sizes, meaning the time between bag accesses varies. In turn, this means threads are less likely to contend with each other when attempting to obtain the mutex to access the bag.

In our plastic schedule, we change from static to dynamic_chunks after 5 seconds. In these first 5 seconds, all threads will still have tasks to process, meaning we still have the maximum throughput. Since we change to a dynamic schedule before we have any down time, we match the performance of a constant dynamic schedule.

Number of tasks	100,000
Task Grain	10,000,000 and 100,000 Repeats
Task Grain Distribution	Biased
Number of CPU cores	4
Number of threads used	4
Thread pinning	Uniform
Schedule	Static, Dynamic Chunks (Chunk Size = 1,000), Dynamic Chunks (Chunk Size = 1) Static, then dynamic chunks after 5 seconds

Table 6.6: Experiment 4 Parameters

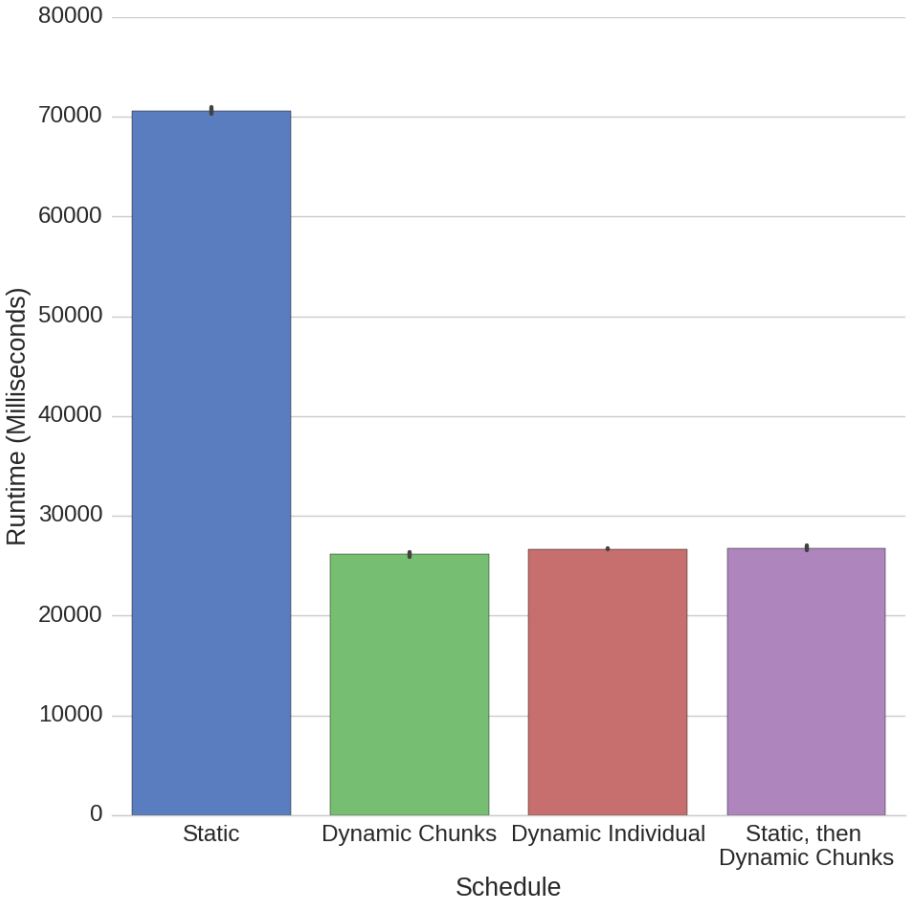

Figure 6.4: Experiment 4 results

6.1.5 Experiment 5 - Absolute Multiprogramming Performance

The purpose of this experiment is to investigate the absolute performance of our library in a situation with multiple applications. We compare two situations, the first, named anarchy, is where both programs are run with four threads with no plasticity or contention aware scheduling. This is a typical situation for a multiprogramming system. The second situation, is using plasticity and contention aware scheduling, where we start with four threads in program 1, once program 2 starts we reduce this to two, and assign two threads to program 2. Once program 2 is finished, we switch back to using four threads in program 1.

For a further investigation into the performance of our library, on a hunch, we also ran the experiment with different message checking intervals. This means that each program's main thread sleeps for Nms in between checks for whether the computation is complete or if it has any messages.

Looking at the results, our library is consistently faster than anarchy, which is a promising sign. We also significantly reduce the variance using our library, possibly due to the reduced thread switching. Comparing the different sleep times, we see that significant improvements can be made, if we tune this variable correctly. Since we are reducing some overhead by making this interval longer, (with the compromise of a slower reaction speed to messages), we could reason that if the library was more sophisticated, and if we thoroughly optimized our code, we may see substantial performance gains.

There is a strange performance gain going from a 20ms checking interval to a 100ms interval. While unusual, it held for the 50 repeats, so it is not an isolated case. Due to time constraints, this could not be investigated further, but this could be a possibility for the future work in the next year of this MInf project.

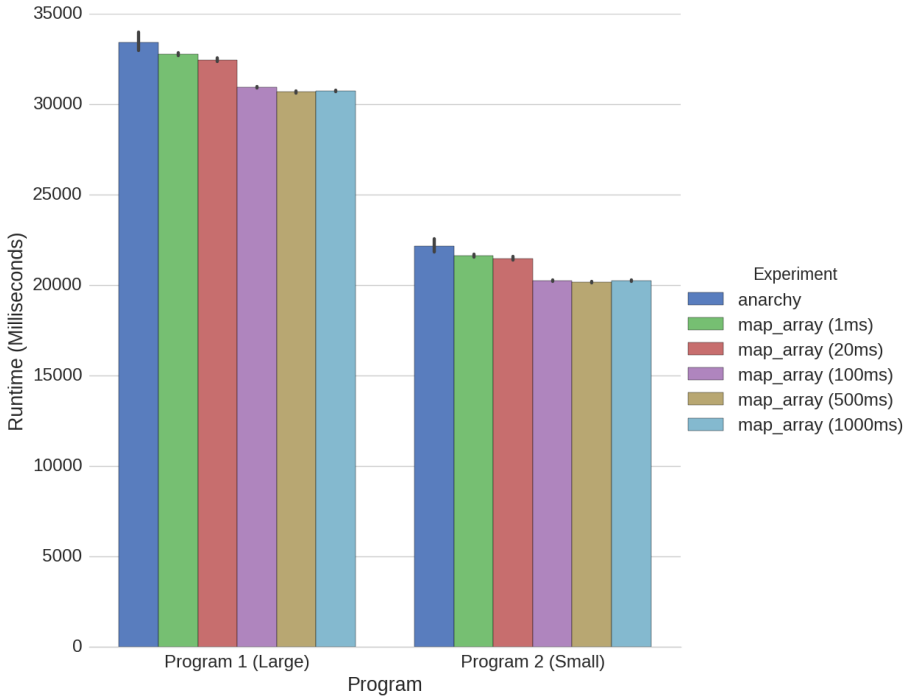

Figure 6.5: Experiment 5 results

Program 1	
Number of tasks	30,000
Task Grain	10,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4 4, then 2, then 4
Thread pinning	Loose, Uniform
Schedule	Dynamic_individual
Program 2	
Number of tasks	15,000
Task Grain	10,000,000 Repeats
Task Grain Distribution	Uniform
Number of CPU cores	4
Number of threads used	4 2
Thread pinning	Loose, Uniform
Schedule	Dynamic_individual

Table 6.7: Experiment 5 Parameters

6.2 Conclusion

6.2.1 Personal Reflection

In this section we provide a personal reflection on the first year of this project, and thoughts for the future.

Working on this project, I have learned much about parallel programming, C++ programming, and managing a large software engineering project. I had not done any of these things prior to this project. Overall I'm pleased with how this project turned out. I've completed all the goals I set at the start of the year, and then some.

Looking back, I wish I had developed more of the report alongside the development of the project, and that I had provided more attention to developing more sophisticated testing software. Whilst I was able to adapt the current system to my needs, it could be easier to perform experiments. As such, this is definitely something I will improve in the next year of this MInf project.

The outcome of this report has provided exciting results for next year, and I'm looking forward to implementing new features, and particularly to further

experiments investigating the system. I would also like to again learn new things, particularly programming for GPUs, and more advanced inter-process communication. Hopefully I can improve this project further, and leave an easy to extend code base for future students.

6.2.2 Overall Conclusion

In this section of the report, we discuss the implications of each experiment, and what they mean for the future of the project.

Experiment 1 investigates the overhead added by detailed metrics collection. These metrics were mostly used during development, and as such were disabled for future experiments to minimize their performance impact.

Experiment 2 establishes a baseline of performance, confirming that we are on a par with current parallel programming approaches. This is a good result, and confirms that our approach does not concede significant overhead.

Experiment 3 investigates the overhead involved in contention aware scheduling and plasticity. The results build upon the conclusion of experiment 2, in that we do not add any significant overhead, this time by measuring the overhead of plasticity and contention aware scheduling.

Experiment 4 is a demonstration of the value of choosing the correct schedule, and shows the benefits of plasticity.

Experiment 5 confirms that we can improve upon our standard implementation with plasticity and contention aware scheduling. We also see reduced variability, both of which are promising results for the early stages of this project. It provides a solid foundation for the next phase of this project in the following year. We also show that we can improve our performance further by optimizing and tuning the code, again providing promising work for the future.

A future system would be useful in any performance orientated application, even when the machine will only be running a single instance, as we can still optimize the implementation to the environment on that machine (as demonstrated in experiment 4.) It would, however, come into it's own when we have multiple instances running simultaneously on a machine, a common situation with modern multiprogramming machines.

Overall, the first year of this project has provided a solid foundation for the work to be done next year, and future work for others following the completion of the project.

Chapter 7

Future Work

This chapter discusses further work that could be done to advance this project. This project has a large scope for development, and as such, we will select a subset of tasks to focus on in the second year. We also discuss possibilities for future applications, and the library's utility beyond simply investigating the nature of the problem.

7.1 Future Development

Here we provide an overview for possible work to be done in the future. We split the tasks into work for the second phase of this MInf project, and interesting work for successors:

Next year:

- Improved experiment framework, with support for experiments involving multiple programs
- Experiment with different array access patterns, and investigate how they affect performance
- Experiment with different types of task, such as different bottlenecks (CPU bottleneck/Memory bottleneck)
- Test the library on a variety of hardware and systems
- Make work interruptable, currently, once we have assigned tasks, a worker thread cannot be interrupted. This means that the static schedule, which

assigns all tasks at once, cannot be interrupted/switched without manually forcing it to do so

- Currently, we only have the map array pattern. More patterns could be implemented, as a real-world version of our library would provide multiple common patterns of parallel programming, and may even utilize multiple backends allowing for different features (e.g. Standard Pthreads, OpenCL/CUDA for multi-GPU computation).
- Support for different parallel programming backends, possibly utilizing more exotic hardware such as GPUs
- Implementing some algorithm for automatic parameter tuning. It's feasible that the system could assess both the tasks presented and the environment (e.g. the particulars of the machine, currently running applications), and automatically allocate the resources of the machine so we perform in the most efficient manner.

Follow on work:

- Different methods of task distribution could be investigated, such as a distributed bag of tasks or adding task stealing
- Project could be adapted for use in a distributed system, where we may have a variety of hardware, machines, and tasks to contend with. In this scenario, we would have a huge amount of input parameters to our scheduler, resulting in a complex problem. We may have programs spread across multiple machines, we need to consider what machine is best to run a particular program on, and what other programs are running on that machine. The system could feasibly be controlled by a single controller application, serving requests from each machine, or it could be distributed across the machines, or we may use a hierarchy, with a controller per machine and a master controller.
- We currently optimize our library for performance, however the system may also be useful in non-performance orientated applications, for example, we may want to optimize energy usage for mobile applications. We could also optimize for a mix of performance and energy, which, with studies reporting that *the total cost of a server will be primarily a function of the power it consumes* [10], would be useful in server farms or data centres.
- In deciding what resources to allocate to which program, we could implement a priority system. Both an internal priority system and a user controlled priority system would be useful. In an internal priority system, we would asses attributes of each program, and possibly assign more resources to one than the other (e.g. if one program scales better than another, we may decide to grant it extra CPU cores until it is complete.)

A user controlled priority system would deal with cases where some programs are more or less important compared to other programs to the user. We could allow the user to manually control priorities, or possibly even the distribution of resources.

Chapter 8

Bibliography

[1] The free lunch is over: A fundamental turn toward concurrency in software, 2017.

[2] Muesli: Multiple utility, evaluation and scripting language interface, 2017.

[3] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. Petabricks. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09* (2009).

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

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

[6] Chandra, R. *Parallel programming in openMP*. Morgan Kaufmann, 2009.

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

[8] Enmyren, J., and Kessler, C. W. Skepu. *Proceedings of the fourth international workshop on High-level parallel programming and applications - HLPP '10* (2010).

[9] Friedman, R. Home - openmp, 2017.

[10] Holzle, U., and Barroso, L. A. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-scale Machines (Synthesis Lectures on Computer Architecture)*, 1 ed. Morgan Claypool Publishers, 2009.

[11] Meade, A., Buckley, J., and Collins, J. J. Challenges of evolving sequential to parallel code. *Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11* (2011).

[12] Meade, A., Buckley, J., and Collins, J. J. Challenges of evolving sequential to parallel code. *Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11* (2011).

[13] Sur, S., Koop, M. J., and Panda, D. K. 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).

[14] Sutter, H., and Larus, J. Software and the concurrency revolution. *Queue 3*, 7 (2005), 54.