An Investigation Into The Performance 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 paper, 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. We then implement multiple experiments to investigate if we can obtain any performance improvements over traditional parallel programming, finding

Insert results summary here. We found X, Y, Z..

Assess it's usefulness,

We have shown X, Y, Z

how it could be taken further, and possibly used to implement a useful tool for programmers.

Planned for part 2 of Minf project, and other possible future work.

Contents

1 Introduction **3**

2 Background **6**

2.1 Current Solutions 6

2.2 What Is Contention Aware Scheduling? 7

2.3 What Is Plastic Programming? 8

2.4 What Is Skeleton Programming? 9

3 Design **10**

3.1 System description 10

3.1.1 Skeleton Foundation 10

3.1.2 Adding Plasticity 11

3.1.3 Contention Aware Scheduling 13

3.1.4 Evaluation 16

3.2 Designing For Future Applications 16

4 Implementation **17**

4.1 17

4.1.1 Skeleton Foundation 17

4.1.2 Adding Plasticity 17
4.1.3 Contention Aware Scheduling 17
4.1.4 Evaluation 17

5 Evaluation And Methodology **18**
5.1 Evaluation Methodology 18

6 Results **20**
6.1 20

7 Discussion **22**
7.1 22

8 Future Work **23**
8.1 23

9 Conclusion **24**
9.1 24

Chapter 1

Introduction

Outline what we will cover in each chapter, Add overview of precisely what was done

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.

It is recognised that writing correct and efficient parallel programs is hard, (cite `http://www.futurechips.org/tips-for-power-coders/parallel-programming.html` or `https://parallel.illinois.edu/blog/three-challenges-parallel-programming` or something, `https://dl.acm.org/citation.cfm?id=2093943&dl=ACM&coll=DL&CFID=720336161&CFTOKEN=82786616`) as the introduction of parallelism introduces a whole host of new problems, (e.g. unreproducible bugs due to race conditions, and difficult debugging as high level instructions need ot be decomposed into atomic assembly code to understand what is going on). The sole purpose of multi-threading is improved performance, however parallel programs can 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 (cite petabricks), which utilize this 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, and during runtime if circumstances change 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 programs, as they typically attempt to use all the resources they can get their virtual 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 (cite lira paper).

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 incorporating them.

The layout of the report is as follows:

Should this be a list of bullet points?

- First we will provide an exploration of the ideas behind the report, and provide an overview of how they are combined in the background chapter.
- Then we will detail the thought process behind the design of the solution in the design chapter.
- We will then go into the specifics of the implementation of the design in the implementation chapter.

- We then plan a carefully selected series of experiments to assess this implementation in the evaluation and methodology chapter.
- Then we discuss the results of these experiments and their ramifications in the results chapter.
- Next, we discuss what all this means (***and how it compares to current methods?***) in the discussion chapter. ***Is this chapter neccecary if we discuss our results in the results chapter, and have a conclusion chapter?***
- Then we layout the roadmap of future work for the second part of this two year MInf project, and also work that could be done after that in the future work chapter.
- We then end with providing our final thoughts on the topic in the conclusion chapter.

Chapter 2

Background

In this chapter, we will detail the current solutions for 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. It has been shown to be an important factor in multiprogramming systems with performance implications (cite LIRA). 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, as it is not known whether they will have a significant effect.

2.1 Current Solutions

Current solutions for parallel programming include:

- Pthreads (POSIX Threads)
- MPI (Message Passing Interface)
- OpenMP (Open Multi-Processing)

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 available with the standard GNU compiler collection (**gcc**). (***Is this accurate?***)

MPI is a communication protocol, and as such mostly details how threads can communicate. It is the dominant model used in the high performance computing industry today (cite https://dl.acm.org/citation.cfm?id=1188565). It can not only be used to implement a parallel program running on a single machine, but can also be used to model 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.

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 is entire books dedicated to the use of each of these particular parallel programming methods.

There does exist attempts to make parallel programming simple and easy for a transitioning programmer. These include:

- SkePU
- The Mnster Skeleton Library Muesli

These methods simplify the problem, with both utilizing skeleton programming to do so. Since each SkePU skeleton has multiple implementations, and SkePU automatically selects the expected fastest implementation variant, it is the closest to the system we wish to implement.

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. (cite LIRA Section 2: Motivating Example) In the cite'd(?) case, just considering two programs running on the same socket, we can see from the graph in figure 2 (Cite) 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 these 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.

2.3 What Is Plastic Programming?

When programming an algorithm, there is 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 while the task size is large we would use radix sort, and then once it is reduced we would use insertion sort. An example of this can be seen in figure 2.1

Such compositions are commonplace, such as the sorting example discussed in the PetaBricks paper (cite petabricks: Introduction, paragraph 2).

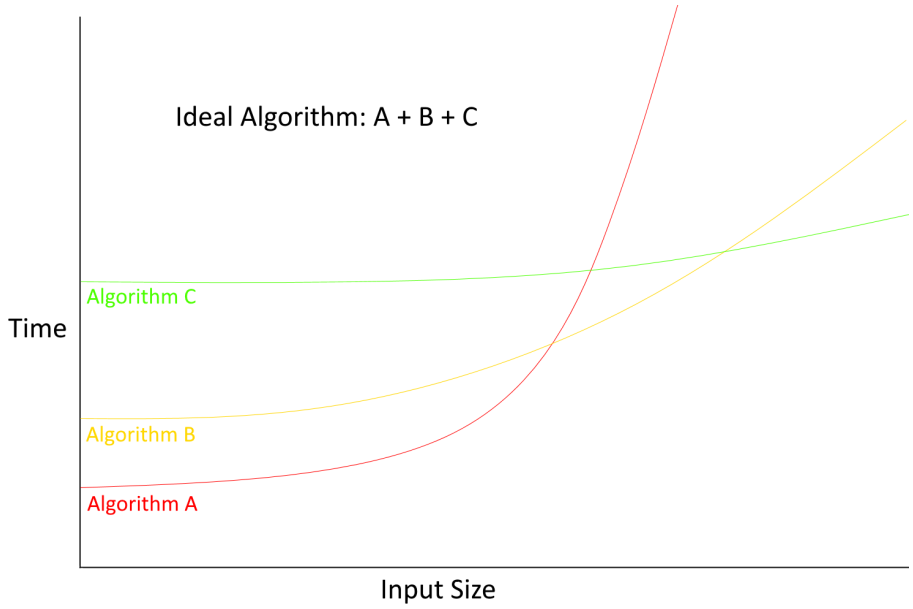

Figure 2.1: An example of three algorithms with different runtime curves, which combined would provide an improved algorithm

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. 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. The ability to combine skeletons makes them a powerful tool, allowing programmers to easily create clean complex programs.

Chapter 3

Design

Explain pattern implemented, (similar to OpenMP (Loop scheduling) and skepu, add differences to blackbourn's work) details of library (Controller etc), and how it may be used in a real system

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. The first pattern implemented is the map array pattern, with further patterns left for possible future work. Map-array is similar to the map pattern, in that it applies a given user function to each element in a list, however map-array also allows the function to access a user provided array. This 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 it also allows 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 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 3.1. 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 3.2.

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. These are called schedules, and some common ones are:

- Static - An equal allocation 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 less and less

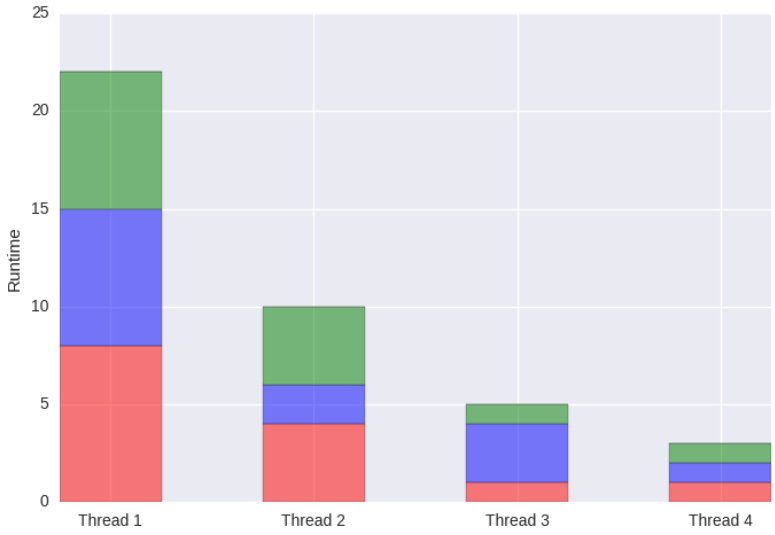

Figure 3.1: A worst case scenario of a static schedule assigning each thread an equal number of tasks

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 (cite LIRA) 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 the machine (e.g., is there more CPU cores available) and the problem (e.g. do we have many small tasks or few large tasks.)

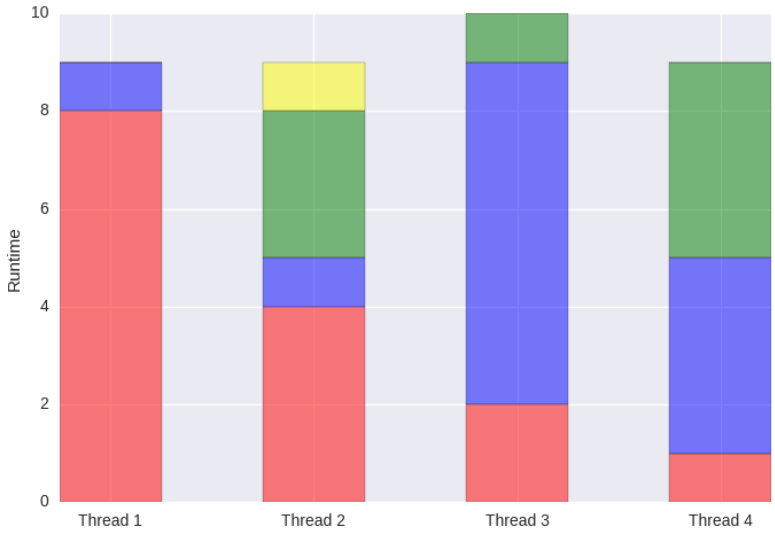

Figure 3.2: An optimized version of figure 3.1, notice that thread 2 has four tasks, but the total runtime is substantially less

3.1.3 Contention Aware Scheduling

To add contention aware scheduling, we need multiple instances of 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. An example of how our programs and the controller program communicate is shown in figure

Once our programs can communicate, and we can control each aspect of them, we can implement contention aware scheduling. (***Is changing something like the schedule still scheduling?***) For testing purposes, 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 can be taken further. We leave implementing some algorithm for automatic parameter tuning for future work.

***Refine from here**

Now that we have contention aware scheduling, we can experiment with multiple

programs running on a system at once. An example of how plastic programming and contention aware scheduling is given in figure

As an example of how this would work in practice is in figure 3.3.

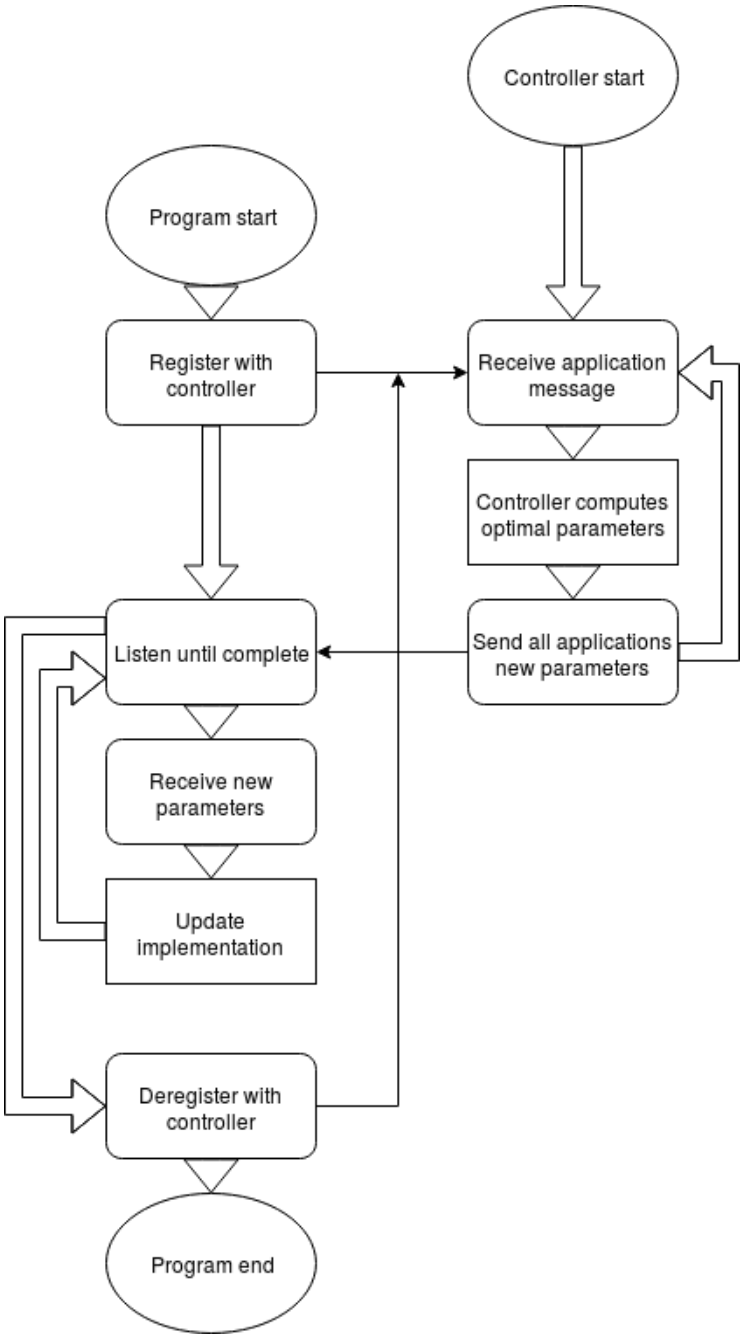

Figure 3.3: How applications communicate with the controller ***Not too happy with this diagram, could be clearer***

3.1.4 Evaluation

To properly evaluate the outcome of this project, we also need points of comparison in terms of performance. So in addition to our skeleton library and controller application, we also implement a dummy program which can run the library with several different parameters to perform experiments. a sequential and an OpenMP version, in which we can vary similar parameters and produce comparable statistics. (*** If complete, add that they utilize the same testing framework)

3.2 Designing For Future Applications

A real-world version of our library would include 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). It's feasible that the system could asses both the tasks presented and the environment (e.g. the particulars of the machine), and automatically allocate the resources of the machine so we perform in the most efficient manner.

This 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. 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.

(Could be used on multiple nodes in a distributed system?)

Chapter 4

Implementation

4.1

Explain implementation details, (C++ etc), librarys used, cross platform support? Is it easy to modify for the future? So on.

threads communicate with shared memory, utilizes bag of tasks etc.

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

4.1.1 Skeleton Foundation

4.1.2 Adding Plasticity

4.1.3 Contention Aware Scheduling

4.1.4 Evaluation

Chapter 5

Evaluation And Methodology

Discuss the systems used briefly, runtime of experiments? evaluation methodology (done with nothing else running?, Different array access patterns? How did you create a synthetic workload?)

To properly evaluate the system,

5.1 Evaluation Methodology

The experiments were run on an Ubuntu virtual machine, with four cores, 4096MB of memory, and the rest of the system at idle.

To evaluate the performance of the system, we need to synthesize a *** CHECK HERE *** real-world 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.

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

assortment of programs

test machine repeats

future work - array access patterns

Chapter 6

Results

6.1

In this experiment, the total run times of three implementations are measured. One sequential implementation, a standard modern parallel implementation utilizing OMP, and our plastic implementation. 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.

The OMP and our implementation are using a dynamic chunks schedule, with a chunk size of 500. All programs were compiled at optimization level 3.

Fig. 6.1 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 baseline implementation performs on a par with current parallel implementations, providing a good baseline performance.

pthread/openMP vs us(different schedules?)

pthread/openMP w/ 2threads vs us w/ 2 threads then 4 (plasticity!) - highlights importance of parameters

pthread/openMP w/ fixed schedule vs us switching schedules (plasticity!) - highlights importance of schedule choice

Above with skewed task distribution

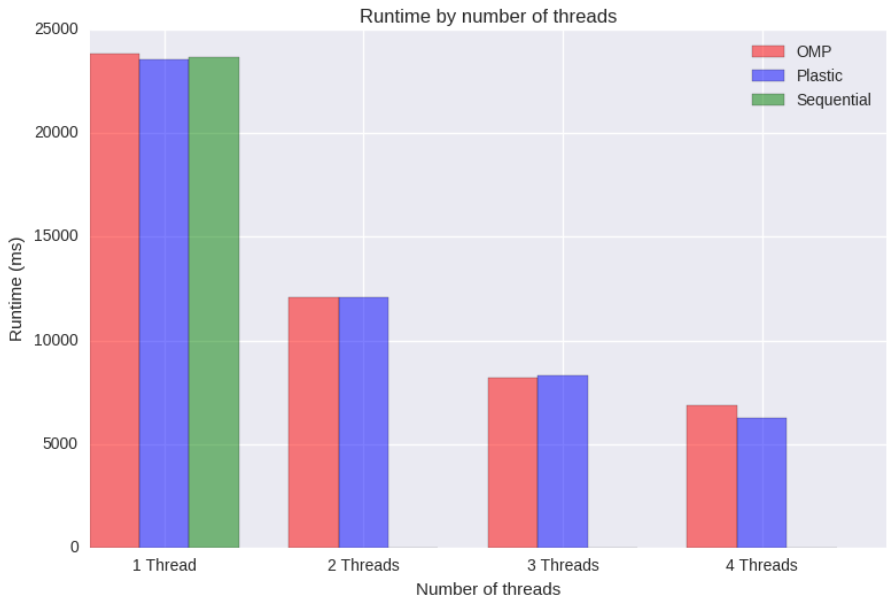

Figure 6.1: Total run times for an assortment of implementations and thread counts. Plastic

Chapter 7

Discussion

7.1

Discuss the findings of the results, (Mention weird runtimes with many small tasks!)

Chapter 8

Future Work

8.1

Overview of possible future work here, including work for next year and after that

Currently optimizing for performance, can optimize for other things e.g. energy efficiency for mobile applications etc.

Utilize GPUs, more exotic hardware.

Test on XXXII and other hardware.

further patterns left for possible future work. Producer consumer?

task stealing?

distributed system?

implementing some algorithm for automatic parameter tuning for future work.

Chapter 9

Conclusion

9.1

Provide overall conclusion, and discuss future work (next year)