An Investigation Into The Performance Of A Plastic Parallel Programming System

Mark Jenkins

School Of Informatics
University Of Edinburgh
2017

Abstract

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.

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.

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).

This paper investigates if we can combine these ideas to create a parallel programming skeleton library where incarnations of its patterns will explicitly work together in this manner, and whether we can obtain any significant difference in performance. We then go on to discuss the ramifications, and how this could be taken further, and possibly used to implement a useful tool for programmers.

Contents

Chapter 1

Introduction

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

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, (unreproducible bugs due to race conditions, and difficulty debugging as need to decompose high level instructions into atomic assembly code to understand). The sole purpose of multi-threading is 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 that our performance gains disappear.

Even when we have an ideal parallel program, often hardware is shared between multiple programs and users, leading to resource contention. There may be messages to the effect of "Do not leave applications running on this machine" or "please nice your programs", which may be unseen or ignored.

The aim of this project is to combine parallel programming with three key ides:

- Co-Scheduling
- Plastic Programming
- Skeleton Programming

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

Chapter 2

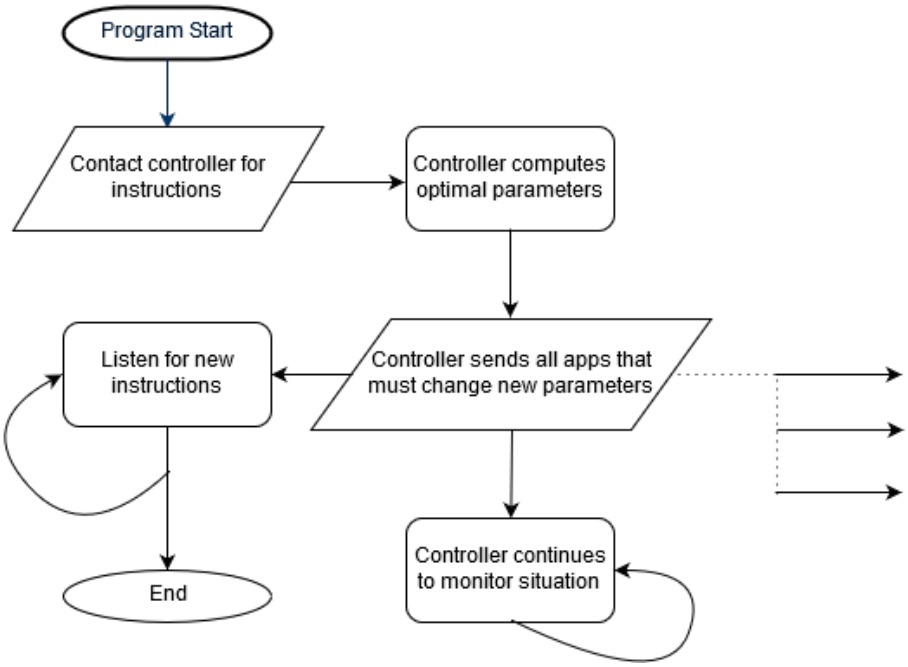
Background

In this chapter, we will explore the three key ideas requisite to this project, such that we can discuss how they are combined and the implications.

2.1 What Is Co-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.

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 aware scheduling in this manner is called co-scheduling. 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 what implementation the program is using. As an example of how this would work in practice, see the following flow diagram:

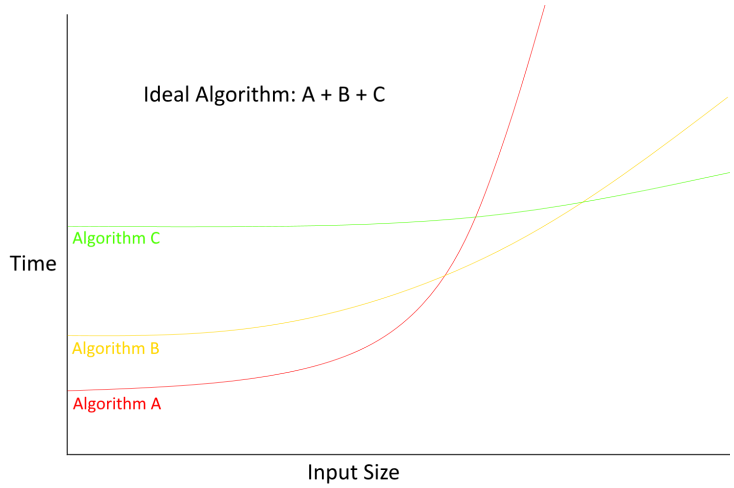


NOTE - CHANGE THIS GRAPH TO REFLECT FINAL IMPLEMENTATION

2.2 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 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.

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



2.3 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 co-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.

2.4 Summary

The main new idea in this project is that of co-scheduling. It is an important factor in multiprogramming systems with performance implications. 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 hide

this complexity 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.

Chapter 3

Design

3.1

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

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.

Chapter 5

Evaluation And Methodology

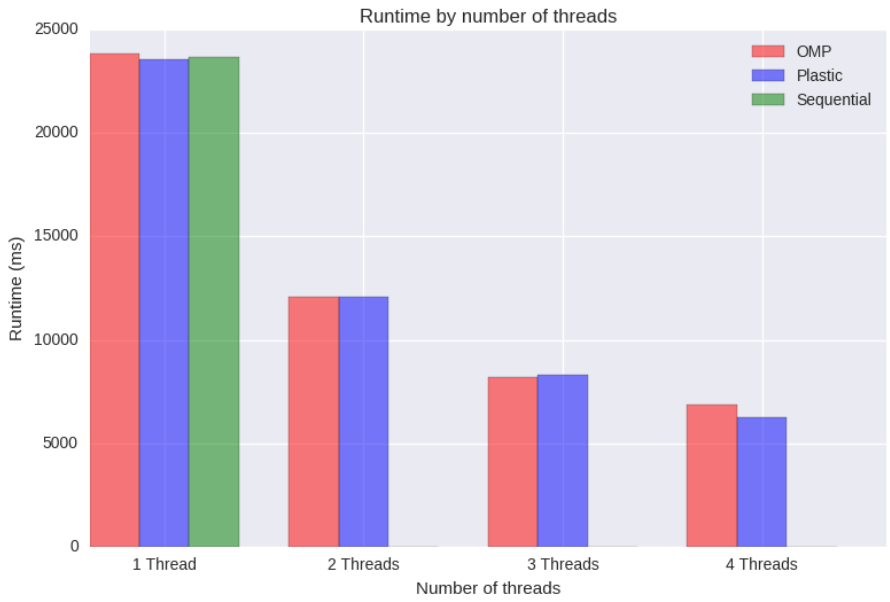
5.1

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?)

Chapter 6

Results

6.1



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

Chapter 7

Discussion

7.1

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

Chapter 8

Conclusion

8.1

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