Design And Analysis Of A Plastic Parallel Programming System

Mark Jenkins

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

Abstract

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

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

Contents

Chapter 1

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 their implications. Then we will cover what was done in the first phase of this project [4], and the intentions for this second phase.

The main idea in this project is dynamic contention aware scheduling and optimisation. It has been shown to be an important factor in multiprogramming systems with performance implications [2]. The intention of this project is to investigate if we can exploit plasticity to mitigate this contention, and what benefits this could bring. Integrating plasticity into parallel programs results in complex code, making it hard to ensure correctness. Since the ideas of this project requires the cooperation of multiple different programs, it is in our interest to make using these techniques as easy as possible, to encourage their use by programmers. Thus, to abstract this complexity away from the programmer, we employ skeletal programming. It also has the beneficial side effect of dividing the challenge into a pattern-by-pattern basis.

In the first phase of this project [4], we investigated a basic parallel programming pattern, map-array. We incorporated this into a feasible estimate of a contention aware plastic parallel programming library, and measured the performance. The main conclusion was that there were performance gains to be had, but more investigation was necessary.

Another conclusion form the previous phase of this project is that the overhead of the contention aware plastic parallel programming library was not significant. To build on these findings, in this phase, we focus on the analysis of a more complex parallel pattern, without integrating it into a complete contention aware plastic parallel programming library.

1.1 Current Solutions

PThreads MPI OpenMP

SkePu

1.2 Contention Aware Scheduling

1.3 Plastic Programming

1.4 Skeleton Programming

1.5 Previous Work

In the previous phase of this project, we designed, built, and tested a contention aware plastic parallel programming library. This was to analyse the potential gains from combining contention aware scheduling with plasticity, and to assess the overhead of using the library.

Design

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

- Plasticity

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

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

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

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

- Communication

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

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

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

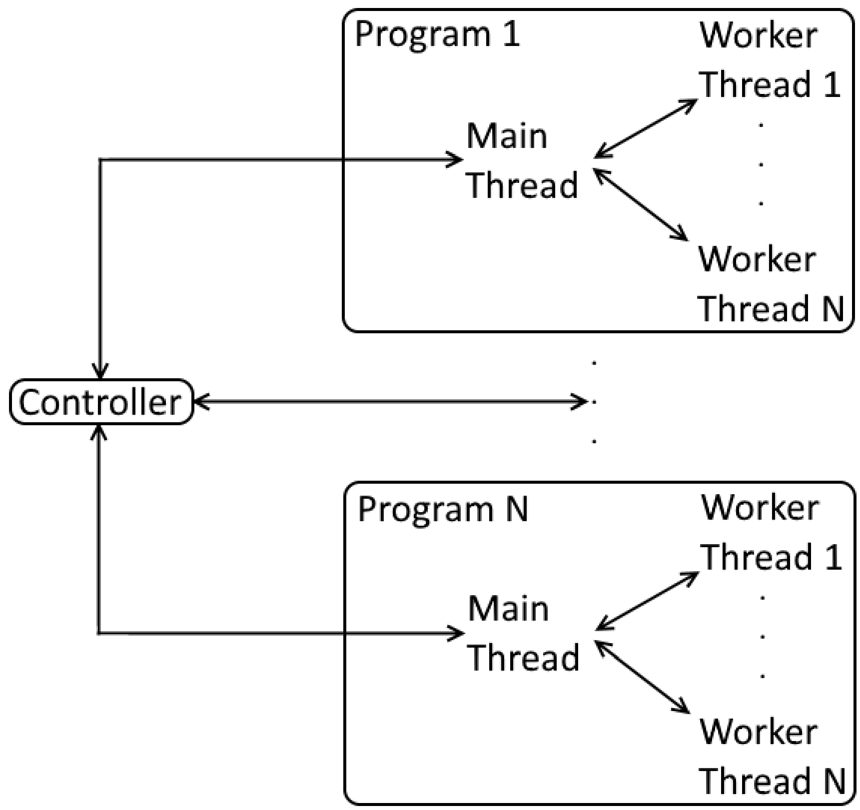


Figure 1.1: High level communication model of the system, with an arbitrary number of programs, with an arbitrary number of threads. Two way communication occurs between the controller and each main thread, and then between each main thread and its worker threads.

Results

The two main

Analysis

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

The map-array pattern is fairly straightforward in terms of complexity and performance characteristics, in that adding more cores/threads to the computation will generally improve performance. Since each program will effectively be able to saturate all the cores allocated to it, we have little room for improvement. Therefore, this year, we decided to focus on a more interesting parallel pattern which wouldn't necessarily improve with extra cores/threads.

Chapter 2

Design

****

Design of project:

- Pick pattern

- Understand performance characteristics

- Select interesting instances

- Refine investigation (Focused) Justification: unrealistic to investigate all instances in given time

- Investigate how each instance runs simultaneously

****

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

In this section, we cover the design of the project as a whole, leaving the details of the system produced for chapter 3, which covers the implementation of these ideas. We then detail the justification of and process of finding specific stencil codes with which to assess our system.

2.1 Project Architecture

The following is a high level description of the plan for this project. It consist of several distinct steps. For this to be completely clear, we first describe the aims of this project, so that they can be kept in mind when reading the plan.

2.1.1 Project Goal

The main goal of the project is to investigate if we can squeeze out some extra performance in parallel multi-programming systems by exploiting contention aware scheduling and plastic programming. Last year, we investigated the design and performance of a plastic parallel programming library, and the conclusion was that such a system would be viable, as the overhead introduced could be easily overcome / was minimal. This year, the focus will just be on whether we can achieve that extra performance. Therefore, we do not need to design and implement a complete programming library, we just need enough to let us carry out our experiments.

2.1.2 Pattern Selection

Last year, we focused on the map array pattern. This is an extremely common parallel programming pattern, however it's performance characteristics are fairly straight forward. If you provide more cores and corresponding threads, you will generally gain performance. This year we will focus on a more interesting parallel pattern. We decided upon stencil codes, which are a class of iterative kernels which update array elements according to some fixed pattern, called a stencil. Stencil codes are widely used, for example in computational fluid dynamics, solving partial differential equations, the Jacobi method, image processing, and cellular automata. In general, any algorithm which operates on finite grids can be formulated as a stencil code.

The particular stencil code we chose is the Jacobi method, and variations upon it to generate differing workloads. The Jacobi method is an algorithm for finding the solutions of a set of linear equations. It updates array elements according to a fixed pattern, then the process is iterated until it converges. The Jacobi method is used in many applications, and as such we use it as an example of a real world application. [6] [1] [5]

2.1.3 Performance Characteristics of the Jacobi Pattern

****

OLD CHAPTER HERE

****

2.2 Design of Experiment Platform

2.2.1 Basic Stencil Code

The first step was to implement a basic stencil code.

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

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

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

2.2.2 Adding Variable Configurations

variables numthreads etc kernels [3]

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

2.2.3 Adding Plasticity

plasticity

2.2.4 Experiment Scripts

experiment scripts

2.3 Interesting Use Cases

2.3.1 Reasoning

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

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

EXPAND UPON THESE

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

All of these can be changed from stage to stage. Beyond these parameters, we can have many different combinations of programs running simultaneously.

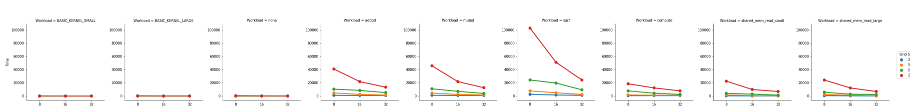
For the purposes of this project, we will select a few interesting use cases of the stencil pattern, and focus on testing these in a few different configurations.

2.3.2 Finding Interesting Use Cases

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

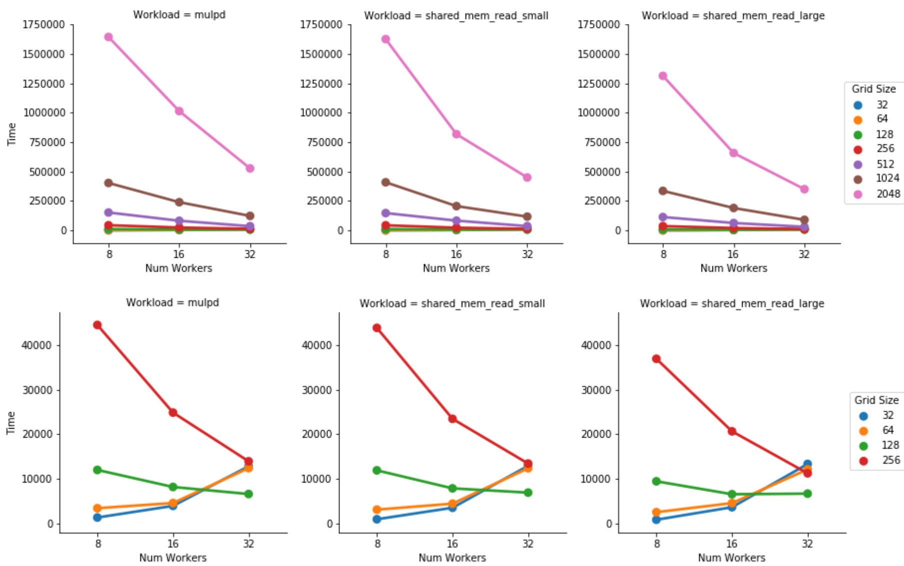
First

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



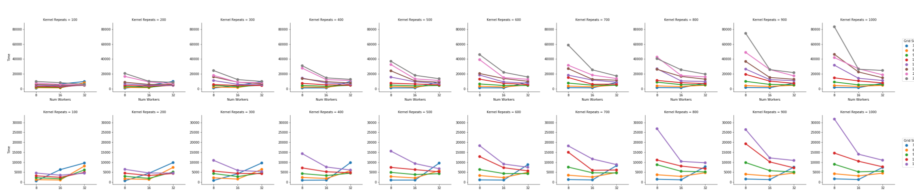
Second

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



Third

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



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

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

2.4 Selected Use Cases

List selected patterns

Chapter 3

Implementation

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

****

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

- Test harness/scripts etc

- Graphing

- ...

****

3.1

Chapter 4

Experimental Methodology And Program

4.1

Chapter 5

Results

****

- Results of each of the previous chapters

****

5.1

Chapter 6

Future Work And Conclusions

6.1

Chapter 7

Bibliography

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

[2] 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).

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

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

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

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