Plastic Parallel Programming

Luke Blackbourn

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2016

Abstract

The rise in multi-user time-sharing systems over the past few decades has introduced a challenge in how to efficiently run multiple applications on the same hardware at the same time. While operating system schedulers have been designed with a variety of tricks to eke out as much performance as possible, the onus is largely on the programmer to design their applications to make optimal use of the available resources, presenting them with numerous algorithm and thread management choices, a task which can be daunting to anyone not accustomed to writing multi-threaded applications.

In this dissertation we present a novel solution to this problem, the concept of *Plastic Parallel Programming*, which provides a framework for non-specialist programmers to write correct efficient parallel programs that react to their environment in order to ensure that they are continually using the most optimal strategies to solve a given problem. Using the well-known example of the *task farm* parallel design pattern, a reactive framework with a high level interface is developed, which allows the user to ignore the vast majority of decisions that are needed to write correct parallel programs, while providing the possibility of complex optimisations. This framework is then tested using a variety of different application types, showing that in the right circumstances plastic parallel programming can improve the runtime performance of applications running in a shared enviromnent. Specifically we show that, for running applications that combine a variety of memory access types, making dynamic changes in reaction to the starting of a second application can result in lower average runtimes for both applications than can be acheived using standard techniques.

Acknowledgements

Firstly I would like to thank my supervisor Murray Cole for taking me on for this project, for his guidance and helpful suggestions on both the code base and this dissertation. I would also like to thank the University of Edinburgh for giving me the opportunity to work on this MSc, along with all of the lecturers and tutors who have taken me through the past year. Thanks should also go to Helen Colhoun for her understanding, and allowing me time off my new job in order to get the project and write-up finished. Last, but certainly not least, a massive thank you to my fiancée Emma for continually supporting me through a busy year, including almost single-handedly moving both of us into a new house. Without your love and understanding this piece of work would never have been completed!

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Luke Blackbourn*)

Table of Contents

Chapter 1

Introduction

It has long been understood that the problem of writing correct parallel programs is a hard one, having a multitude of complications that are nonexistent in classical sequential programs, such as race conditions when multiple parts of an application may try to modify the contents of a single memory address at the same time, and the related challenge of often not being able to rely on *happens-before* relationships between parts of your code [1]. While there have been a number of attempts to allow easier access for the non-specialist to the possible benefits of parallel programming, the assumed difficulty of doing so is known to put off many in the scientific fields who would benefit from it the most [2, 3]. This means that the easily available multi-core and multi-socket commodity hardware is being massively under-utilised, even while computer simulations are becoming ever more important in just about every scientific discipline [4–7]. Even when researchers do have access to these resources and have the tools available to build multi-threaded applications, resource contention becomes an issue when the hardware is shared between multiple users, this being a problem for anyone who has to share computers, including university students. While large-scale properly managed clusters such as the EPCC ARCHER supercomputing service [8] have sophisticated systems such as the Open Grid Scheduler [9] to manage tasks and resources, more basic shared computing resources are often only protected by a request to "please `nice` your programs", with stories of runaway tasks taking all of the CPU time being common.

There have been a number of attempts to rectify some of these problems, to try to make parallel programs easier to write, easier to optimise and more efficient to run in shared environments. One of these is LIRA, a dynamic scheduler which is discussed in §2.3, which uses intelligent thread pinning to minimise the interference between multiple applications running on the same multi-socket hardware. Another is PetaBricks, discussed in §2.2, a framework and programming language which presents a high-level interface to the programmer, making compile-time choices as to the details of the algorithms and data structures that it uses, in order to make optimal use of the hardware it is compiled on.

In this dissertation we introduce the notion of *plastic parallel programming* in §2, which combines some of the ideas from LIRA and PetaBricks, as well as the concept of algorithmic skeletons as defined by Cole [10] and discussed in §2.1, into an overarching framework that is designed to allow non-specialists to write correct efficient parallel programs that dynamically adjust themselves to their environment. We then present a summary of the goals of the summer project and this work in §3, followed by a description of the implementation of a plastic task farm that was written, in §4. The experimental framework that was used for testing this implementation is described in §5, with details of the experiments that were carried out and their results given in §6. These results are then reviewed in §7, with the dissertation giving its final conclusions in §8.

Chapter 2

Background

The past half century or so has seen dramatic changes in the way that computers are used, with the ever increasing demand for computing resources constantly providing new challenges. Early operating systems in the 50s were batch systems, where multiple people would submit jobs, generally on punch cards, which would then be run one at a time by an operator. There were a number of problems with this approach, with one being the fact that there was no direct interaction between the programmer and the computer, so if a program failed the programmer would not be informed until much later, in which case they would have to change, then resubmit it. Another problem was that, since only one process was run at any one time, with the next job not started until completion of the previous one, if the process depended heavily on I/O, which was, and still is, significantly slower than processing speed, a computer could spend a large amount of time waiting for the I/O to complete. This design was improved in the 1960s with the rise of multiprogramming, in which many programs could be loaded into memory at the same time [11]. If the currently running program had to wait for some external event, another idle program could be executed until that event had completed, dramatically reducing the idle time of the computer. This decade also saw the rise in *time-sharing* computers [12], where multiple users could directly connect to the computer, running programs almost in real time. With many people using a computer with a single processing element simultaneously, and expecting the system to react quickly to their input, the importance of proper scheduling of programs on the CPU became apparent. Originally multitasking schedulers, such as those used in the earlier versions of Windows and MacOS, often used *cooperative multitasking*, where a program would execute until it voluntarily gave up control of the CPU, either when waiting for a response from some external device or because it decided that it had been executing for long enough. There was always, however, the possibility that a program could hold the CPU indefinitely, for example if it entered an infinite loop. The development of *pre-emptive* multitasking, in which the operating system will interrupt a process if it is deemed to have had its 'fair share' of processing time, removed the problem of badly behaved programs keeping the

CPU, paving the way for ever more complex schedulers. These include the Linux Completely Fair Scheduler (CFS) [13] which holds information about each process, such as whether it is CPU bound or I/O bound, as well as allowing processes to be prioritised.

Recently, the scheduling problem has been even more complicated as CPUs started to develop more cores, with high-end machines having multiple CPUs through multiple sockets, often with Non-Uniform Memory Access (NUMA). Much work has gone into making schedulers that aim to allow all tasks equal access to the processing elements, however they tend to come with some downsides. For example schedulers do not know the details of a program, so do not know in advance how a program may behave. They may try to make a guess, for example noting that a certain program spends a large amount of time in memory accesses, or is heavily CPU bound, and compensate accordingly, however a program's behaviour may change suddenly, leading to massively sub-optimal scheduling decisions. The scheduler is also restricted to decisions that the application programmer makes, such as the number of threads that a program has, and while the scheduler can decide when and where to run the threads it has no control over how the application utilises the threads. There have been a number of attempts to try to rectify these problems, for example through better adapting the program to the hardware that it is running on or taking more control over the scheduling decisions, although these solutions are often too complex for non-specialists to fully take advantage of.

2.1 Parallel Design Patterns and Algorithmic Skeletons

As discussed above, the large variety of choices that are necessary to produce correct and efficient parallel programmes may be expected to dissuade researchers without a background in software engineering from attempting to do so, even though a large proportion of the complex software used today is written by exactly this demographic [14]. If they do attempt to write such programs there are a host of different bugs that can occur, the proper debugging of which often requires the use of more sophisticated methods than the standard `printf` method [15]. Even when written by experts these bugs often crop up, sometimes with harmful results, such as the race condition in Nasdaq's system that delayed Facebook's initial IPO by 30 minutes, resulting in the loss of millions of dollars [16]. In fact the problem is so widespread that around 60% of respondents to a survey of Microsoft's technical staff reported having concurrency issues with their software [17]. It is thus not surprising that researchers are often put off writing parallel programs.

A standard method that is used to help less experienced programmers choose the correct implementation to solve a particular problem involves the concept of a *design pattern*. Originally conceived as a way for ordinary people to understand and be able to use designs that are frequently seen in professional architecture [18], design patterns give a solution to a com-

monly recurring problem within a particular context [19]. While design patterns gained popularity in the object oriented programming community they were rapidly adopted by scientific high-performance programming groups, in particular the concept of *parallel design patterns* [20], as a way of removing the need to re-invent the wheel whenever a scientist is faced with a problem of a standard sort. While undoubtedly a step in the right direction, since they are high level concepts parallel design patterns still leave programmers having to make many low-level design decisions themselves, such as details of the choice of algorithms to use, numbers of threads, or what programming languages and libraries to use. These problems were addressed by Cole [10] with the idea of *algorithmic skeletons*. The idea behind this concept is that the algorithmic skeleton, which is written by an expert in parallel programming, provides a high-level interface for some design pattern that a researcher can use without worrying about the low-level details. This interface can then hide a lot of complexity that can go on behind the scenes, with the skeleton making decisions on optimal strategies to use, as well as protecting the researcher from many platform-dependent decisions, making their code highly portable.

2.2 PetaBricks

The concept of hiding implementation details in order to allow portability and low-level optimisation, as discussed in the previous section, was put into practice by Ansel et al. [21] in their *PetaBricks* programming language and compiler [22]. A programmer provides methods, known as *rules*, with which state changes can be implemented in the application to produce the result that they desire. The framework then does compile-time analysis of the system in order to choose the most efficient rules for carrying out those state changes, which includes combining many rules together to achieve certain goals, or varying free parameters.

One downside of the PetaBricks system is due to the fact that all of the optimisation decisions are made at compile-time, meaning that they may in fact have the opposite effect to that intended if the environment that the application is running in is changed. These changes could range from another application starting up, using up resources, to changes in the physical configuration of the hardware, such as the hot-swapping of a broken CPU that can be done on some modern high-end systems such as those in the Intel Xeon E7 family [23]. In order to address this shortcoming, we take inspiration from another system that aims to take on some of the complexity of creating efficient parallel programs in a shared environment, LIRA, which is discussed in the next section.

2.3 LIRA

One of the major causes of non-optimal program execution comes about due to the limited amount of fast on-chip memory. Program data must be loaded into caches from main memory, which may take hundreds of clock cycles. This is mitigated somewhat by clever cache coherence protocols, however the movement of threads between cores, or even worse different sockets, can practically negate these. In order to counteract the migration of threads, which are moved around by the scheduler to try to ensure that as many threads are running as possible, it is possible to implement *thread pinning*, where threads are restricted to running on a subset of cores. This can limit the number of times that data must be loaded from memory, at the expense of restricting the available processing elements. It is also possible to pair together programs with different characteristics, such as one that is I/O bound and one that is CPU bound. This allows the CPU bound process to run while the I/O bound process is waiting for the much slower memory access, allowing both to make the maximum use of the available resources. It is this idea which underlies LIRA [24], which controls thread placement in an adaptive contention-aware manner. In one incarnation called LIRA-static, initial sets of tests are done on the applications that will be run, working out the optimal positions of each, while LIRA-dynamic uses hardware performance counters to to categorise each application according to its recent behaviour, and changes scheduling as the application goes through behavioural phases. This categorisation depends on the load instruction rates of the running applications, with the scheduler choosing application pairings that minimise the sum of the differences between the load instruction rates of the applications on each socket.

While Collins et al. [24] were able to show that intelligent thread placement could reduce the overall runtime of various applications, LIRA still has the disadvantage that it can only work with the threads that it is given, so a badly written program may not be able to run efficiently regardless of the scheduling policy that is used. Plastic parallel programming aims to combine the ideas behind LIRA, which show that context-dependent thread pinning and dynamic strategy choices can improve program runtimes in a shared environment, with the ideas of algorithmic skeletons and PetaBricks, that shows that using high-level interfaces to hide implementation details allows for behind-the-scenes optimisation, to create a framework that allows non-specialist programmers to create efficient correct parallel applications that intelligently react to their environment.

Chapter 3

Project Description and Goals

In this chapter we introduce plastic parallel programming, along with the goals that is is hoped that this project would achieve.

3.1 Plastic Parallel Programming

As has been seen in the previous chapter, a number of approaches have been taken to try to reduce the complexity associated with writing correct efficient parallel programs, while at the same time increasing the ability of an application to pick the optimal strategies for the environment in which it is running. While LIRA has the advantage that the application writer does not have to worry about any of the scheduling details, and LIRA-dynamic is able to make changes on-the-fly as the environment changes, it still has limitations in that it fully relies on the application programmer to pick the best algorithms for the job, and correctly implement them. PetaBricks, on the other hand, chooses the algorithms to use from a selection provided to it by the programmer, however once the application has been compiled these choices are fixed, so environmental changes, such as another application starting up, can render the choices suboptimal.

In this dissertation we introduce the concept of *plastic parallel programming* that encompasses many of the ideas from algorithmic skeletons, LIRA and PetaBricks to provide a high-level programming interface for classic design patterns, while hiding a complex mechanism for altering strategies at runtime to give optimal performance. Inspiration is taken from the idea of algorithmic skeletons and PetaBricks to provide a simple programming interface, representing a frequently used design pattern, with implementation details hidden in a way that allows optimisation that the programmer does not need to worry about. This concept is developed further by adding the dynamic element from LIRA-dynamic, allowing runtime changes in the choice of algorithm used, as well as intelligent thread placement to optimise applications where memory access and cache behaviour play key roles in the overall runtime.

3.2 Project Goals

The goal of this dissertation is to implement and test a plastic version of a well-known parallel design pattern, which provides a high-level interface to the programmer and makes changes to the way that it implements that design pattern behind the scenes depending on the environment in which it is running. It is of course important to test that the introduction of plasticity to a parallel design pattern actually has a positive effect on the runtime of the applications under its control. This project aims to provide results from a large number of tests in order to ascertain in which cases it would be most useful to use plastic parallel programming, and which forms of application react best to added plasticity. A description of the test setup that is used is given in §5, with the results presented in §6. While there was a strict limit on the time that was available for testing, it is the intention here to provide as much statistical justification as possible for any claims made, meaning that the tests that are carried out should all be justified, with enough samples to give statistically reliable results.

Chapter 4

Project Implementation

In this section we introduce the classical task farm parallel design pattern, illustrating the scenarios in which it is useful as well as the difficulties associated with implementing it, before describing our adaptation of the design pattern into a *plastic task farm*, which is able to modify its behaviour depending on the environment in which is it running.

4.1 The Classic Task Farm Design Pattern

The design pattern that was chosen to be adapted for this project was the *task farm*, which is a classic parallel design pattern that while relatively simple has a very large number of possible variations [25–28]. As the name suggests the task farm depends on the concept of a *task*, which is a unit of work that can be independently scheduled, meaning that tasks in a task farm must be self-contained and not dependant on the results of any other tasks, unless explicit barriers are put in place. Tasks are usually generated by a master thread then collected in some sort of data structure, with worker threads taking tasks whenever they are free. Task farms are very useful for problems such as parameter sweeps and Monte Carlo simulations, where tasks may represent, for example, the modelling of a system with various possibly random initial conditions, and the results aggregated to produce statistical information about the system. One advantage of a task farm is that it is easy to convert serial programs into tasks, since there is no intra-task concurrency, indeed this is the basis for many batch processing systems, which essentially manage load balancing for multiple individual processes on multi-core or multi-socket hardware [29, 30]. There are also many implementations using libraries such as MPI [31], however this project takes a rather lower level approach, dealing directly with the threads through the `pthreads` library [32], as this allows substantially more control and a larger variety of possible optimisations.

4.2 Task Farm Strategies

When implementing a classical task farm, there are many choices that have to be made with regards to implementation details, ranging from the design of data structures to runtime scheduling decisions. The optimal decision to make in each case usually depends on both the hardware that the application is going to be executed on, and the environment in which it will be running. The choices that are made in an attempt to optimise the program are known here as *strategies*.

4.2.1 Data structures

A standard example of a strategy choice that has to be made is the form of the data structure that holds the tasks to be run. The simplest case, which can be used when there is a single shared-memory process, is to have a single list of tasks, with protection provided in the form of a mutex whenever a thread wants to take a task off the list, or else having a master thread that alters the data structure and hands out tasks. This setup however will start running into problems if there are many threads, at which point there will be a large amount of contention for the mutex or master thread leading to high overhead, or if the application is running on a NUMA architecture, so threads on a different socket from the task list have to wait a significant amount of time before getting their next task. An example strategy to improve performance in this case is then to spread the tasks over multiple lists, with each thread taking tasks from the list that it is closest to in memory, possibly transferring tasks over if one list becomes empty, a process known as *work stealing*, similarly to the work stealing that occurs in operating system schedulers [33].

4.2.2 Threading

Another set of strategies, that we have been concentrating on during this project, involves the runtime decisions around thread numbers and scheduling. There is a vast literature on the effects that the numbers and placement of threads has on application runtime [see e.g. 34–37, and contained references]. The standard rule is to ensure that threads that access the same data structures run on the same CPU to maximise cache efficiency, while on NUMA architectures they should access local memory as much as possible over remote memory. One way of controlling this is through *thread pinning*, which tells the operating system scheduler to restrict the threads to a subset of the available cores, accessed through the Linux `taskset` command, or the pthread `pthread_setaffinity_np` function call. As is often the case there are downsides associated with thread pinning, as it reduces the scheduler's ability to load balance, so it is possible for an entire CPU to sit idle while another is overworked if all the threads are pinned.

As well as thread placement one set of strategies is to control the number of active threads that is used by the task farm. If all of the threads are CPU bound then it may not make sense

to have more threads than processing elements, since the remaining threads will just sit idle and result in more work for the scheduler. However as was discussed in §2.3 where LIRA was introduced, if threads have different characteristics, such as some being I/O bound and some CPU bound, having more threads than cores may allow the CPU bound threads to execute while those that are I/O bound wait for the hardware. The optimal number of threads also depends on the environment in which the application is running, since if another application is running at the same time that application's threads may contend for resources, even if the total number of threads in the initial application is less than the available CPU cores. This means that at these points it may be appropriate to reduce the number of active threads. There are different ways to control the number of threads, with the most obvious being to spawn and join them as necessary. This however tends to have high costs associated with it, meaning that implementations often use a *thread pool*, where threads are spawned at the beginning of the process and wait idle, using minimal resources, until they are needed, at which point they are woken up and carry out their work, before returning to sleep.

4.2.3 Task Granularity

One important topic that is often overlooked when describing a task farm is the question of how a task itself is defined. In batch processing systems this is generally simple, as a task is an individual process with the task farm effectively acting as a scheduler or load balancer, however when the task farm is contained within a single process the job of partitioning work into individual tasks becomes a lot more flexible. An example of this occurs when the job consists of iterating over some data structure such as an array or linked list, and carrying out some work on each data element. This sort of job is common in parameter sweeps [38], where the initial array could consist of structures containing sets of initial conditions of interest, or Monte Carlo simulations, where repeated random sampling builds up statistical results. In this scenario the entire program could be a single large task, referred to here as *course-grained* or *low granularity* tasks, or each piece of work on a single array element could be a task, called *fine-grained* or *high granularity* tasks, as long as the pieces of work are independent of each other. There are also intermediate levels of granularity where each task takes a subset of the array to work on.

As with thread pinning there are advantages and disadvantages for any chosen granularity. For low granularity tasks there is less overhead and less contention for shared data structures, although if there are fewer tasks than available processing elements there is obviously wasted CPU time, and since each task will tend to run for longer there is a higher likelihood that there will be idle time at the end when there are only a couple of tasks left to run. Conversely if there is a large number of fine-grained tasks there is a lot more likelihood of contention for access to shared data structures, and since each task will run for less time the ratio of time

spent getting new tasks versus running tasks will be increased. The optimal task granularity itself will depend on the architecture that the application is being run on, and while, similarly to threads, there are general rules of thumb that can be applied to decide on granularity in advance, it is only really through testing that the ideal granularity can be found.

4.3 Controller Application

One of the features that was present in LIRA-adaptive and allowed dynamic changes to the system was a thread that each application had which periodically woke up, checked various pieces of information that have been collected from hardware counters during the course of the execution of each application, and using this data notified the main threads of each program as to which schedule should be used. This ability to do runtime monitoring lies at the heart of plastic parallel programming, with the monitoring threads being promoted to having a process of their own, known here as the *controller*, which assesses the environment in which the applications are running and tells each application which strategies it should use at any given time. There is essentially no limit to how complex the controller could become, or the factors which it could take account when making its decisions, although it should be remembered that the more work the controller does the more CPU time it will itself take up, meaning that the benefits to the other applications must be greater to make it worthwhile. For the sake of this project, due to time constraints and the need for the tests to be repeatable, the controllers' decisions were hard-coded into simple scripts, implementing pre-decided strategies depending on which stages the applications were at and how many applications were running. The design of the controller was however made such that this could easily be extended if desired for future work.

Since the controller necessarily has to communicate with the applications in order to both get details as to the applications' current state and to inform them of which strategies to use, some sort of messaging protocol is necessary. It was decided to use the Zero Message Queue (ØMQ) lightweight messaging library [39] for this purpose, which has the facility for simple client-server communication using standard TCP sockets. The controller binds to a given port on `localhost`, with any participating application connecting to that port, registering with the controller when it starts, getting strategies to use when it reaches certain phase points in its lifecycle, and de-registering when it exits. This use of standard sockets means that the controller could easily be converted to run on a separate machine in a distributed system. The behaviour of the controller is largely reactive, responding to messages from the applications, which means that it spends most of its time sleeping, waiting for the next incoming message, and does not take up much CPU time. This was verified by a set of tests in which applications ran either with or without the controller, using the same strategies each time apart from the controller

communication. As seen in §6.1 the controller used an insignificant amount of CPU time, however care should of course be taken as the addition of more applications and more complex decision making by the controller would likely lead to a noticeable increase in resource usage.

4.4 The Plastic Task Farm

The idea behind the plastic task farm is to take the variability in implementation details that was mentioned in the previous section, and make it available at runtime, so that the task farm can choose what strategy to use depending on the environment at that time. This variability is hidden behind the high-level algorithmic skeleton interface, so that the programmer does not even have to be aware that changes are being made behind the scenes. In this project we concentrated on strategies involving thread placement, in particular number of threads and thread pinning, since there was limited time to implement the task farm, and in order to run a thorough set of tests the space of available strategies could not be too large. The details of the strategies that were investigated are described in §6.4.

4.4.1 Initialisation and running

Using the API for the implemented plastic task farm requires four main stages: initialisation; adding tasks and phase points; running; and finalisation. A general picture of the system is given in figure 4.1, which shows the lifetimes of two applications and the controller. The initialisation phase merely consists of a call to `tf_init`, passing the port number with which to communicate with the controller and a character string denoting the file to write output data to, and returning a `tf_context_t`, which is an opaque struct that is passed to any taskfarm functions and keeps track of the state of the system. During the initialisation phase the application sets up the data structures that it will use, and registers with the controller on the given port. The controller will pass back an initial strategy to use, in particular details of the thread pool size. In this implementation the thread pool size cannot change after it is initially set, however it would not be too hard to add variable thread pool sizes in. When it has the initial strategy the task farm then sets up the initial data structures that are used to store and distribute tasks, and spawns the initial threads. When instructed to, the spawned threads will execute all of the tasks, with the initial program thread acting as the master, doing all of the communication with the controlling application and directing the worker threads. After initialisation, tasks and phase points can be added, a process which is described in §4.4.2.1 and §4.4.2.2. When all tasks have been added the `tf_run` function is called, which instructs the master thread to set the worker threads running, executing all of the tasks that were added. The final action is a call to `tf_finalise`, which cleans up the data structures and outputs any final metrics, which are described in §4.4.4.

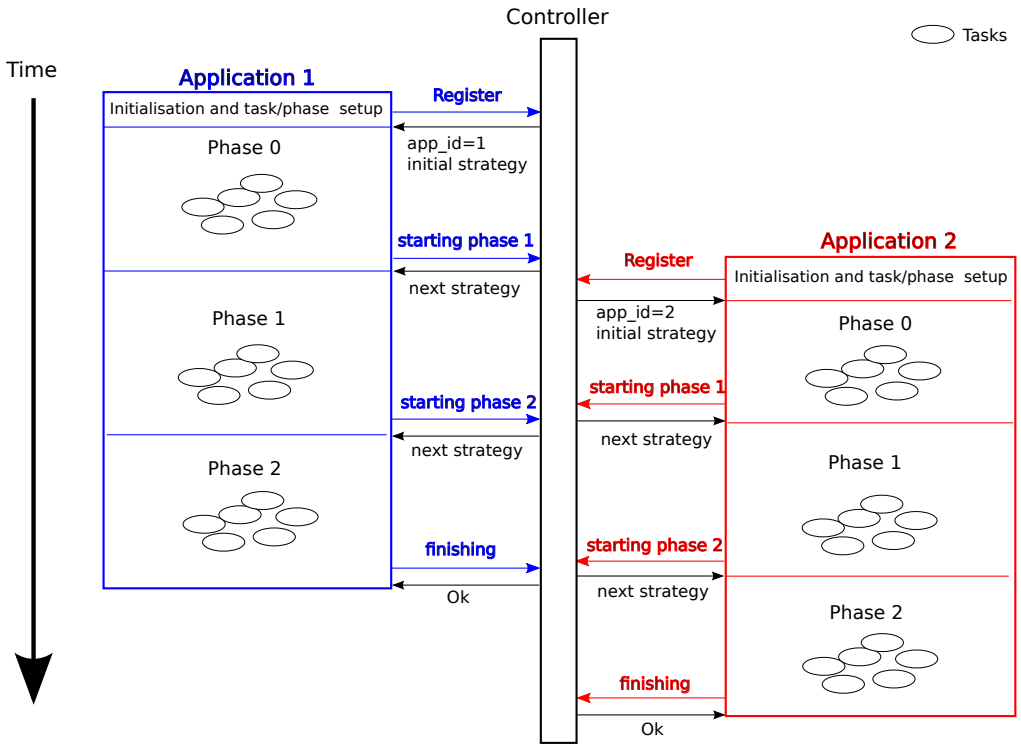


Figure 4.1: System diagram showing two applications starting, registering with the controller, running tasks in three phases, then de-registering and exiting.

4.4.2 Tasks and Phases

Two of the most important parts of the plastic task farm are the concepts of *tasks* and *phases*. These are fundamentally intertwined, and described in the next two sections.

4.4.2.1 Tasks

The most fundamental part of a task farm application is the process of adding tasks. In this project a task is always a function that takes a void pointer to some user-defined data structure, taking any input from and putting any input into that structure or memory referenced by it. It is the job of the user to control any memory access restrictions in memory referenced in this structure. The act of adding a task to the parallel task farm is as simple as calling the `tf_add_task` function, which tasks as its arguments the `tf_context_t` struct returned from the initialisation routine, a pointer to a work function, which must be of type `void (*)(void *)`, and a pointer to a data structure to pass in to the work function.

As described in §4.1, one of the standard uses of the task farm parallel design pattern is for parameter sweeps, where the same piece of work is repeated multiple times, with the only differences being input parameters, a scenario which is well suited to automation over an array containing input values and pointers to where to put results. This is the principle behind the `tf_add_foreach` function, which allows an extra layer of indirection between the programmer and the implementation decisions. As an example of its use, consider a data structure

```
struct {
    double input, output;
} data_struct;
```

with the working array

```
data_struct work_array[SIZE];
```

where the `input` values in the array are set to some initial value. A call to `tf_add_foreach` would then be

```
tf_add_foreach(tf_context, work_array, sizeof(data_struct),
SIZE, num_phases, flags, work_func);
```

where `tf_context` is the structure returned by the initialisation function, `num_phases` is the number of phases that the loop over the array should take, `flags` gives the types of phase points to use, as described in §4.4.2.2, and `work_func` is the function that is called on each of the elements of the array. The decision on how to distribute the array iterations over different tasks and how to organise the tasks into the number of requested phases is left entirely up to the plastic task farm, giving it a large amount of flexibility. Ideally the number of phases to use would also be set by the framework rather than the programmer, however since this was one of

the factors that was varied in the experiments it was easier to include it in the API than to build it into the framework at this stage. In this project's implementation the distribution is arranged such that there is one task per phase for each thread in the thread pool, so increasing the value of `num_phases` increases the task granularity. For example if there are 5000 elements in the array that is being looped over, there are 16 threads in the thread pool and `num_phases` has the value 1, then there will be 16 tasks, that is each thread will only have one task to run in the entire program, with each task carrying out 312 or 313 pieces of work from the work array. If on the other hand `num_phases` has the value 100, then there will be 1600 tasks, with each task doing three or four of the work array elements, with a phase point after every 16 tasks. In the first case there is obviously significantly less overhead, since the threads are distributed all of their work at the beginning and spend the rest of the time doing that work, however once the work starts there is no possibility for strategy changes until the program has finished. On the other hand the second case has the threads continually carrying out small pieces of work, meaning that the time spent retrieving tasks may be a significant fraction of the overall runtime. The numerous number of phases also means that it will be in contact with the controller frequently, possibly leading to large overheads, although also allowing the application to be quickly notified of changes in strategies if the environment changes.

4.4.2.2 Phases

Since the controller is designed to spend most of its time waiting for messages, in order to reduce its impact on the system, it is up to the running applications to decide when to contact the controller in order to receive instructions as to what strategies they should be taking. The frequency of these communications could potentially have a powerful effect on the efficiency of the program for similar reasons as the task granularity. If the application communicates with the controller too frequently then it could spend more time communicating with the controller than carrying out its tasks, and the controller could become swamped with messages, leading to it using more of the valuable resources itself. On the other hand not contacting the controller frequently enough means that the application cannot react quickly to changes in environment. The correct frequency for communication itself depends on the environment, since a stable environment would mean that a lower communication frequency is necessary.

In this project the communication between application and controller is determined by the existence of *phase points*, which delimit groups of tasks. Phases are numbered sequentially, with phase points being global synchronisation points, meaning that there is a strict *happens-before* relationship between the phases, although not necessarily between the tasks in each phase, as described below. When the final task that was added to the task farm before a given phase point has started, the application's master thread initiates a controller communication, notifying it of the change of phase, and receiving the strategy to use for the next phase. There

are different types of phase points, depending on the needs of the tasks before and after them. By default when moving from one phase to another the application will communicate the transition to the controller, receiving in reply the strategy to adopt for the next phase. The most basic phase type just does this communication, with tasks from the next phase able to start before the next strategy is received, in fact the next task may start before the previous tasks from before the phase point have finished. This is most useful when the controller should know that a new phase has started, but it is not expected that there will be a change in strategy, and it is desired that the next set of tasks should finish as soon as possible. The controller communication can be turned off by using the `PHASE_BYPASS_CONTROLLER` flag, which also ensures that the phase number is not incremented. By itself it is no different to not having a phase point, but it can be combined with other flags such as the `PHASE_WAIT_FOR_TASKS` flag. This creates the second sort of phase point which is similar to a `barrier` in OpenMP [40], in which it is guaranteed that all tasks from before the barrier are finished before any tasks are started after the barrier. Another phase point type comes by using the `PHASE_WAIT_FOR_STRATEGY` flag. In this case the application does not continue with the post-phase tasks until it receives a reply from the controller and implements the next strategy. This final form of phase point is used for all of the tests that are described here.

Phase points are added via the function `tf_next_phase` which takes as arguments the `tf_context_t` struct and flags representing the phase point type that is desired, or via the `tf_barrier` function which is an alias for a phase where there is no controller communication, but all tasks started before the phase point must finish before the next tasks start.

4.4.3 Threading

Thread control in the plastic task farm comes about through a mix of `pthreads` and the ØMQ communication library. When the initial thread pool is spawned each thread sets up a communication channel with the main thread using ØMQ in-process communication. With the communication channel set up the thread then waits for the startup message from the master thread, at which point it will start removing tasks from the task list and running them. When a worker thread finds a phase point it communicates this to the master thread, then either waits on a phase barrier, as described in §4.4.2.2 or carries on running tasks from the next phase. Between each task threads check whether there is a message from the master thread, either telling the thread to block until an unblock message is received, or telling the thread to terminate. The thread blocking is the mechanism that is used if fewer threads are required than were originally spawned in the thread pool, with threads using up minimal resources while they wait on a waking signal, while being easy to resurrect if more threads are required. This allows a highly efficient and flexible mechanism for adjusting the number of threads in the application, and could be easily extended if different strategies were needed.

4.4.4 Measurements

At various points in the lifecycle of the plastic task farm various timings are taken, in order to give as much information as possible about how each thread behaves. These timings all depend on wall clock times, rather than cumulative CPU time, since it is this time that is most relevant for most users, although cumulative time can easily be calculated by adding up the timings for each thread. The reported times include the time that each thread was running for, the time that a thread was running tasks for, the total time that a thread was blocked on a mutex for, and the total time that a thread was blocked by the master thread, waiting for an unblock message. These results could be output at the end of each phase for debugging, or at the end of the application run for standard execution. During all of the testing that is described in the results section the percentage of time any thread spent mutex blocked never rose above 0.17%, with the vast majority of thread runs spending effectively no time mutex blocked, with a percentage so low that it registered as zero when printed to six decimal places. This suggests that the task farm implementation was highly efficient for the circumstances in which it was used. This overhead would however be expected to rise as the number of threads increases, meaning that a different strategy for holding and distributing tasks could become necessary.

Chapter 5

Experimental Framework

5.1 Introduction

While there is good reason to believe that combining the best parts of PetaBricks and LIRA would produce a system that can increase application performance, it is of course vital to test this, particularly as there are often effects such as caching that are very difficult to predict. Different types of application will require different strategies to get the most out of the hardware, while there may be applications in which the added overhead of plastic parallel programming outweighs any benefit that may be gained through the flexibility offered. It is thus vital to carry out comprehensive testing to show that there are circumstances where plastic parallel programming can produce significant optimisations, as well as finding the best strategies to implement in those cases. There are of course a huge range of possible tests, with an essentially unlimited number of application types and strategies to try, but due to time constraints it was only possible to test a small subset of these, although there were enough to demonstrate that plastic parallel programming has the potential to have a significant positive impact.

In this chapter we describe the setup for the tests that are run to try out plastic parallel programming. In §5.2 we describe the environment in which the tests take place, in particular the relevant details of the hardware setup on which the tests are run, with §5.3 describing the testing protocols that were used, as well as the types of application work that were tried, in order to find the optimal strategies for various different types of application. We attempt to cover a large range of different behaviours in a relatively short period of time. Finally in §5.4 a description is given of the data analysis routines and presentation methods that are used, along with some predictions based on simple heuristic arguments.

Motherboard	Dell PowerEdge R810 05W7DG quad socket [43]
RAM	64GB NUMA, 16GB per socket DDR3 1333MHz
CPUs	4 × Intel Xeon L7555 Octa-core 1.84Ghz
L1 Cache	32K per CPU, split into data and instruction, core private
L2 Cache	256K per CPU, core private
L3 Cache	24M per CPU, shared between cores
Hyperthreading	disabled

Table 5.1: Hardware details

5.2 Hardware

All of the tests that were done were run on the informatics machine `xxxii`, which has a four-socket NUMA architecture [41], meaning that each socket has part of the memory which it can access very quickly, with access to the other non-local memory taking significantly longer. Each socket contains an octa-core Intel Xeon L7555 CPU, giving a total of 32 cores, each running at 1.86GHz [42]. More details are given in table 5.1. The core IDs are distributed so that the first socket has cores with IDs $0, 4, 8, 12, 16, 20, 24, 28$, and similarly with the other cores.

5.3 Testing Protocols and Work Types

All of the tests that were run were restricted using the Linux `taskset` command to only run on half of the available cores, either the cores on sockets 0 and 1, or the cores on sockets 2 and 3. This allowed two sets of runs to be done at the same time without interfering, or let another user run processes on the other sockets, which was necessary since this was a shared machine. The first set of tests that were done, described in §6.3, verified that running two sets of tests on the two sets of sockets did not make a noticeable difference to one set of tests pinned to the first set of sockets.

All of the tasks done during the tests shared a common underlying theme, involving access to a shared array, which consists of $2,000,000$ doubles. This gives an array with a total size of 16MB, meaning that the entire array can fit into the L3 cache of one of the CPUs, however if two applications are running on the same socket there will be cache contention. This means that caching effects will be very important in the interaction between multiple applications. Each piece of work in an application accesses this shared array in some form, with an entire application consisting of 5000 pieces of work, a number chosen because it gave a runtime that was long enough such that results could be statistically significant over the background noise, but runtimes were short enough, at around a minute per application, that a large number of tests

could be run. This also allowed a large range of different granularities to be tested, since for 16 CPUs each task could consist of between 1 and 312 pieces of work.

Four different memory access patterns were used when accessing the shared array, since they each produce different caching effects, which are the main effects that were expected to be important in this project. These covered memory read/write, and random/sequential access, with the sequential access moving over the entire array, while the random access used the re-entrant random number generator to generate indices, although accessing the array the same number of times as the sequential access. This did pose a problem when comparing these two types of work, in that calling the random number generator takes a significant amount of time, in fact in a set of initial tests work using the random number generator took many times longer than work that accessed the array sequentially, even when caching effects were taking into account. In order to more meaningfully compare the effect of caching behaviour on different work types calls to the same random number generator were added to the sequential access work, with the returned value ignored. Since the programs were all compiled with the `-O2` flag, which among other optimisations tells GCC to remove dead code, the variable the number was stored into was marked `volatile` to ensure that the random number generator was called as expected. When this change was made the timings for sequential and random memory access were essentially the same when cache coherence effects were accounted for.

5.4 Result Presentation and Analysis

In this section we describe the analysis that we perform on the results of the tests, the way that we present the results, and give an introduction to the main statistic that we use when reporting the data, the Average Normalized Turnaround Time (ANTT).

5.4.1 Data presentation

There are numerous ways in which it is possible to present experimental data, which include methods that do not accurately represent the data, or skew the presentation so as to attempt to force a result, a fact that is well known through the saying popularised by Mark Twain: ‘lies, damned lies and statistics’. There has been a resurgence in recent years in attempting to ensure that experiments are properly designed and results fairly reported, with numerous studies reporting problems with many previous research articles [44–46]. While the author cannot pretend to be an expert at statistics, it will be attempted to present the results here in a way such that it is clear when results are significant, and when they are purely the result of chance or noise. The graphical methods of presentation will thus vary slightly depending on the aim of the test, trying to toe the line between being comprehensive and looking overly cluttered.

One common method of result presentation is via a *box-and-whisker* plot, which gives a representation of the spread of the data, as well as statistics about the data. A standard plot consists of a box which encompasses the first and third quartiles (that is the box contains half of the data points, with the first and last quarter of the sorted data points outside the box), with the median value denoted by a thick black horizontal bar. It is known that the median is less susceptible to outliers [47], which are a particular problem in these test since all outliers tend to lie in the same direction, with occasional tests having dramatically increased runtimes due to outside interference such as system processes or other users on the system. The 'whiskers' on the box plot show the range of data points that lie within 1.5 standard deviations of the mean, with any points outside this range viewed as outliers and represented using circles. While the box plot does a good job of giving the layout of the data, it does not by itself give any indication of how significant the results are. In order to visualise these the box plots include *notches*, which represent confidence intervals around the mean, with the relevance that if the notches of two plots do not overlap then that is 'strong evidence' that there is a significant difference between the two medians [48]. The details of the calculations that are used to calculate the notches can be found at [49] but are not discussed further here.

As well as the box plots it is also often useful to display other information in a graphic. Where it is deemed useful a horizontal red line is overlaid over the plot, representing the mean of the distribution. The graphics also make use of *violin plots*, laid in gold under the box plot, which represent the distribution of the data in the same way as a histogram. The violin plot includes a thick black vertical line which represents the interquartile range, as well as a white dot at the median.

All of these elements are combined in figure 5.1, which gives sample results for the runtimes of two applications, running separately in isolation. Some results to note are that the first application has a more widely spread distribution, largely caused by the presence of the outlier, which pulls the mean value, represented by the horizontal red line, away from the median. This reinforces the decision to use the median rather than the mean as the measure of central tendency. The second feature to note is that there is a large amount of overlap between the notches in the two bar plots. This means that it is not possible to state that the average runtimes are different in any meaningful sense.

5.4.2 Average Normalized Turnaround Time (ANTT)

As mentioned in the previous section, the main statistic that is going to be used to make comparisons across different tests is the Average Normalized Turnaround Time (ANTT). This statistic represents the amount by which multiple applications interfere with each other when running at the same time, as opposed to each one running in isolation. This is the statistic that was used by Collins et al. [24] to demonstrate the efficacy of LIRA, and is based on a suggestion

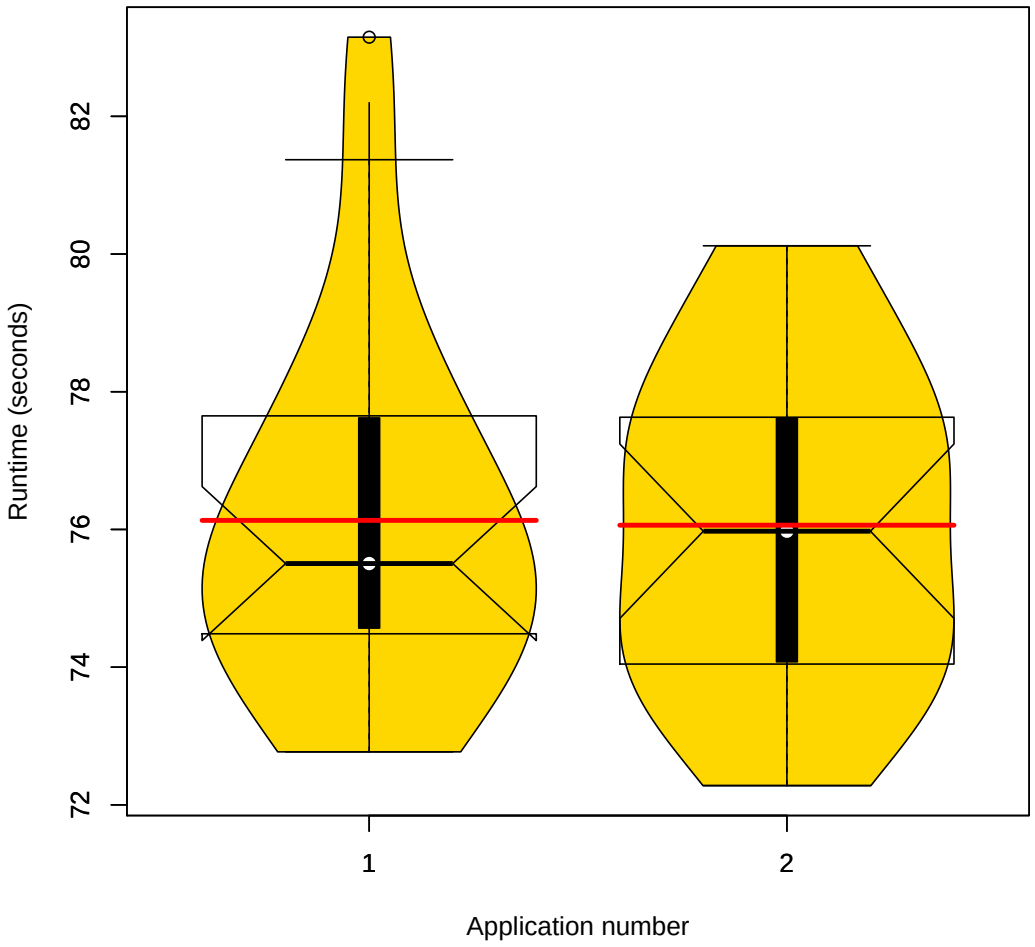


Figure 5.1: Example distribution plot

made by Eeckhout [50] as a statistic to evaluate performance in multi-user systems. Given n applications running simultaneously, the ANTT can be defined as

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i^{MP}}{T_i^{SP}} \tag{5.1}$$

where T_i^{SP} is the runtime for the i^{th} application when it is running in isolation, and T_i^{MP} is the runtime for that application when it is running in the shared environment. As applications interfere with each other, increasing their runtimes, the ANTT increases from the "ideal" value of 1. The definition of ANTT gives a clear direction for testing, with applications first being run in isolation in order to calculate T_i^{SP}, then run together, either without plasticity or with plasticity and using various different strategies, in order to find the scheme that gives the lowest ANTT, which will be the optimal strategy to use. Since the ANTT involves ratios of runtimes it is also possible to compare different work types, even if they have different base runtimes, in order to decide which strategies work best for which types of work.

5.4.3 Theoretical Analysis

The relative simplicity of the applications that were run during the testing phase means that it is possible to make various estimates of the ANTT, given simplifying assumptions about the behaviour of the applications. These can then be used to compare against the actual results as a help to understanding the behaviour of the applications in the real world, and possibly help to work out what assumptions were not valid, shining a light on what is happening behind the scenes.

Let us suppose that a program has perfectly divisible tasks, so that a running application can be described as running at a certain number t_1 of tasks per second, with the entire program consisting of T_1 tasks, taking a total time of $S_1 = T_1/t_1$ seconds. When it runs with another application the scheduler is assumed to assign it and the other task equal access to the resources, so it will then on average run at a speed of $t_1/2$ tasks per second. Suppose now that the first task starts up, with the second task starting after s seconds, that is a fraction p_1 of the time through the isolated run, where $p_1 = s/S_1 = st_1/T_1$. At this point the first application will have done st_1 tasks, so it will have $T_1 - st_1$ tasks remaining. Since it will now run at $t_1/2$ tasks per second, assuming that the second application runs for the entire remaining time, it will take an extra $(T_1 - st_1)/(t_1/2) = 2((T_1/t_1) - s)$ seconds, giving the first application a total runtime of $2((T_1/t_1) - s) + s = 2T_1/t_1 - s$, which is equal to $s(2-p_1)/p_1$. The runtime, when compared to the run in isolation, has then increased by a factor of $T_1^{MP}/T_1^{SP} = (s(2-p_1)/p_1)/(s/p_1) = 2 - p_1$. This means, for example, that if the second application starts half way through the first application's run and carries on running until the first application has finished, the runtime of the first application will increase by a factor of $3/2$.

Suppose similarly that the second application runs at t_2 tasks per second, with the program consisting of T_2 tasks, giving an isolated runtime of $S_2 = T_2/t_2$ seconds. If it starts up while the first application is running then it will then initially run for $2((T_1/t_1) - s)$ seconds at the reduced speed of $t_2/2$ tasks per second, carrying out $t_2((T_1/t_1) - s)$ tasks. Since the first application has now finished, it will carry out the remaining $T_2 - t_2((T_1/t_1) - s)$ tasks at the full speed t_2, taking a further $(T_2/t_2) - (T_1/t_1) + s$ seconds. This then gives the second application a total runtime of $(T_2/t_2) + (T_1/t_1) - s$, so the runtime will have increased by a factor $T_2^{MP}/T_2^{SP} = 1 + (T_1t_2/T_2t_1) - st_2/T_2$.

It is now simple to calculate the ANTT for this simplified case, giving

$$\begin{aligned} ANTT &= \frac{1}{2}\left(\frac{T_1^{MP}}{T_1^{SP}} + \frac{T_2^{MP}}{T_2^{SP}}\right), \\ &= \frac{1}{2}\left(2 - p_1 + 1 + (T_1t_2/T_2t_1) - st_2/T_2\right), \\ &= \frac{1}{2}\left(2 + 1 - p_1 + (T_1t_2/T_2t_1) - p_1(T_1t_2/T_2t_1)\right), \\ &= 1 + \frac{1}{2}\left(1 - p_1 + (T_1t_2/T_2t_1)(1 - p_1)\right), \\ &= 1 + \frac{1}{2}(1 - p_1)\left(1 + \frac{T_1t_2}{T_2t_1}\right), \\ &= 1 + \frac{1}{2}(1 - p_1)\left(1 + \frac{S_1}{S_2}\right). \end{aligned} \tag{5.2}$$

It is easy to put some rigorous bounds on this, under the assumptions that the second application starts part way through the first application, that is $0 \leq p_1 \leq 1$, and assuming that the first application finishes before the second application does, which means that $1 - p_1 \leq S_2/S_1$, then the ANTT is bounded by $1 \leq ANTT \leq 2$. This gives us a range of values in which the assumptions we made above may be reasonable, with an ANTT value outside this range meaning that one or more of the assumptions are not valid.

One final set of assumptions that can be made are allowed by the restrictions that were put in place during testing, namely that in this project all of the applications that were running at any one time were of the same type, meaning that they each had the same isolated runtime, at least in the statistical sense. This means that $S_1 = S_2$ and equation (5.2) simplifies to

$$ANTT = 2 - p_1. \tag{5.3}$$

If p_1 take the value $1/2$, the ANTT takes the simple value of $3/2$, assuming that all of the assumptions listed above hold. It was decided for the majority of runs to start the second application half way through the first application's run, in order to have a consistent baseline with which to compare ANTT values, both one set of tests with another and compared to the theoretical results. The only set of runs that this was not carried out on was the final set, discussed in §6.6 where the second application was started after a random time in order to better simulate real world behaviour.

Chapter 6

Experimental Programme and Results

In this chapter we discuss the experiments that were run in an attempt to verify the efficacy of plastic parallel programming. The tests were split into three main phases. The first phase of testing was run to check that plastic parallel programming was feasible in that there wasn't excess overhead generated by having an extra controller process, having communication with the controller, or distributing tasks as described in §4.4. In all of these cases the applications were run in isolation, apart from the possible existence of the controller process, so these tests were also used to get the baseline runtimes T_i^{SP} for use in the ANTT calculations. It was also checked that it was possible to conduct two set of tests simultaneously on two different sets of sockets, using `taskset`, without interfering with each other. In the second phase a large number of tests were run with two simple applications running in contention, in order to gauge the interactions within the four different work types. These were run over a range of work loads, strategies and granularities in order to try to get a handle on how plastic parallel programming could best be used. The final phase used slightly more real world conditions, with the second application starting a random time after the first application started and workloads that are slightly more reminiscent of those found in real life applications, in order to ensure that plastic parallel programming could work in more complex situations.

6.1 Test for Controller Use of Resources

The first set of tests that was run had a single application, which used the sequential reading work type, running in isolation. The first case had the application running without any communication with the controller and without a controller application running, while the second case had the application communicating with the running controller application, but the only strategy that was used gave the application full access to the available resources. The results are shown in figure 6.1. As can be seen the addition of the controller has no statistically significant effect on the runtime of the single application. The figure also supports the decision to use the

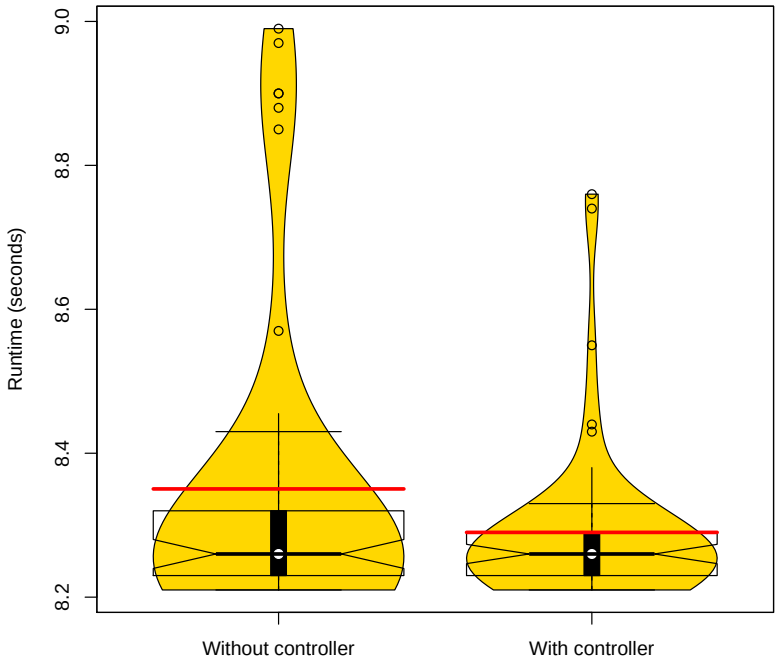


Figure 6.1: Isolated application running with and without controller

median rather than the mean runtimes, since the means are clearly skewed by the few outliers, as discussed in §5.4.1.

6.2 Baseline isolation runtimes

The same tests were then done with the other memory access types, sequential memory writes and random memory read/write. Similarly it was found that there was no significant difference between running without the controller and running with the controller but not changing strategies, so the non-controller results are not shown here. Figure 6.2 shows the results for the applications using the four different memory access types to the shared array, sequential read/write and random read/write. The horizontal blue line marks the value of the median of the results for each work type, which is the value that is used as T_i^{SP} in the following sets of results.

There are some interesting aspects in the results that are worthy of note. First, as may be expected, the different work types give significantly runtime, even though, as described in §5.3, an effort was made to ensure that time differences do not come about through different numbers of instructions or calls to functions such as `rand_r`. This means that the differences in runtime are largely the result of different caching behaviour, which is backed up by the measured runtimes. The fastest run, with a median runtime of 8.26 seconds, is in the sequential read, the memory access type where caches are most efficient. Work involving random reads take slightly longer, around 9.31 seconds, presumably largely through lower-level cache effects, since the entire array fits into the $L3$ cache of each socket. Sequential writes take significantly longer, at around 13.25 seconds, due to the fact that writes to the first element of a cache line on one socket will necessitate that the corresponding cache line on the other socket is invalidated. The random write work takes a lot longer, at around 56.67 seconds, since almost every array access will be for a different cache line and result in a cache invalidation on the other socket, resulting in vast amounts of communication between the sockets, and thus the greatly increase runtime. One point of interest is the bi-modal distribution that occurs in the sequential write. In order to check whether this was just an anomaly the set of tests for sequential write were repeated 150 times, which showed the same bimodal distribution, strongly suggesting that there is some underlying reason for this distribution, although the precise reason is not clear, and would warrant more investigation if there were time.

The results here were also used to check how many times the test should be run to ensure that the results are statistically significant. If it is desired that there is a 95% confidence that the sample mean is within a margin M of the actual mean, in a distribution with a standard deviation σ, then at least

$$n = \left(\frac{1.96\sigma}{M}\right)^2$$

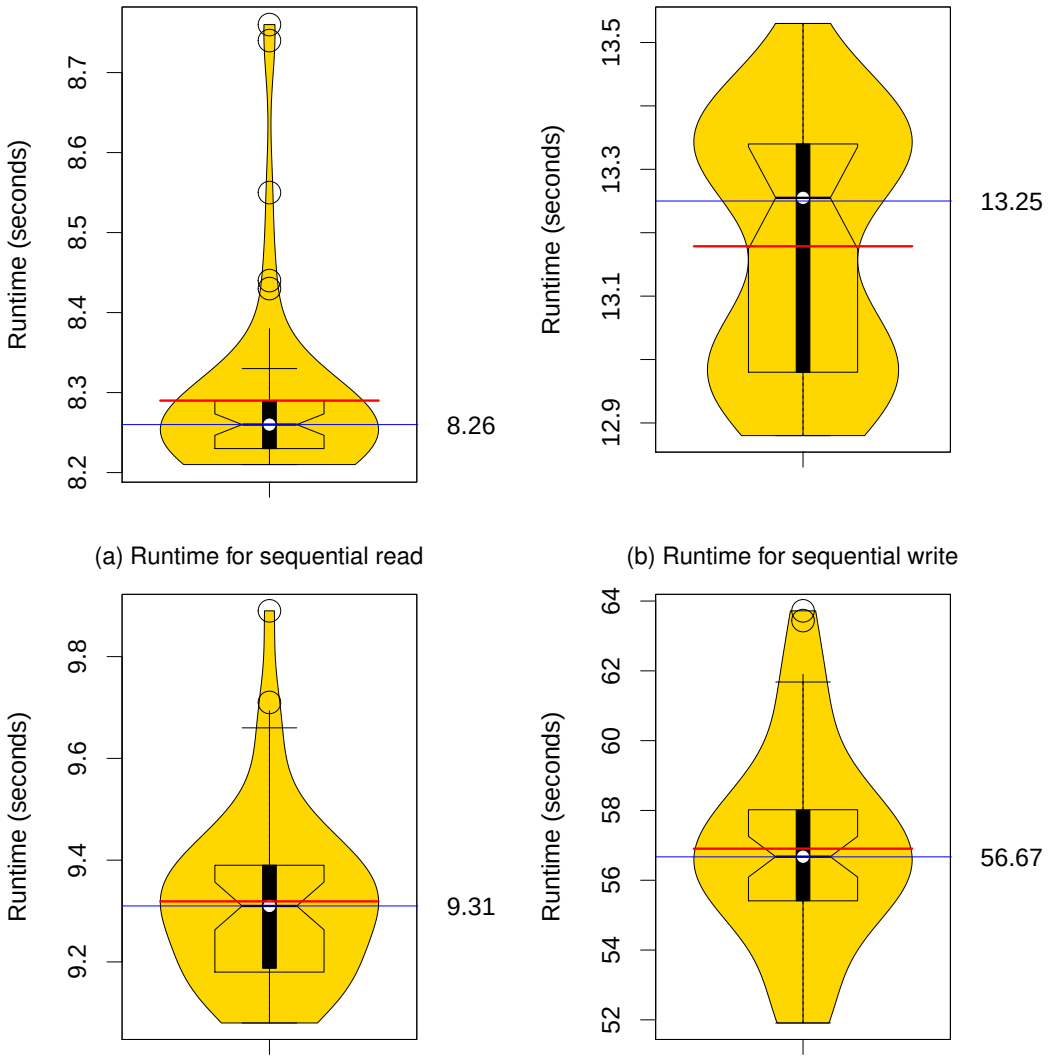
samples are needed [51]. Using this calculation with the four distributions in figure 6.2, with the desire that the margin M is within 1% of the mean, the number of samples needes for the sequential read, sequential write, random read and random write was calculated to be 8, 9, 12 and 89 respectively, rounding up. Given the times taken to run the tests it was decided that it was not feasable to run every test 89 times. It was decided to use 20 runs for the set of tests described in §6.4.3 that included granularity tests, since there were are large number of granularities to test. Other tests were to be run 50 times. This gives sample sizes that are significantly more than needed in all cases apart from the random writes, in which case there is a 95% confidence that the actual mean should lie within a distance of $M = 1.18$ from the mean when $n = 20$, and within a distance $M = 0.75$ of the mean for $n = 50$. While the distributions will of course change the required sample size, these numbers give an idea of the significance of any results that are presented here.

6.3 Test Isolation Check

The next set of tests that were run were designed to check that it was possible, using the `taskset` command, to restrict sets of applications to sockets in a way such that they did not interfere with each other, as described in §5.3. This was done by running the same application, with the sequential read workload, along with the controller, without anything else executing on the machine. Then two sets of the application and controller were run concurrently, with one set restricted to the first two sockets, and the second set restricted to the second two sockets. The results of these tests are shown in figure 6.3. The gold left-hand plot shows the runtimes when the applications was run by itself, with the middle and right-hand plots showing the runtimes of the two tests that were run together. Interestingly running two set of tests together actually seemed to slightly reduce the runtime of each, however overall there is little difference between the two, suggesting that the test setup that was used was sound.

6.4 Multi-application Runs and Strategies

Having done the baseline tests, the multi-application tests were then run. As mentioned previously each multi-application test involved two instances of the same application, with the second instance starting half way through the first application's run, using the isolation tests to calculate the timing for this. The first application was generally started with full access to the available resource, then when the second application was started a strategy decision had to be made as to how the first application would react, and the set starting conditions for the second application. When the first application finished the second application was then given access to all of the resources.



(a) Runtime for sequential read

(b) Runtime for sequential write

(c) Runtime for random read

(d) Runtime for random write

Figure 6.2: Distribution of runtimes for each work type, running in isolation apart from the controller, and not changing strategies. The blue line marks the value of the median, which is used in later ANTT calculations.

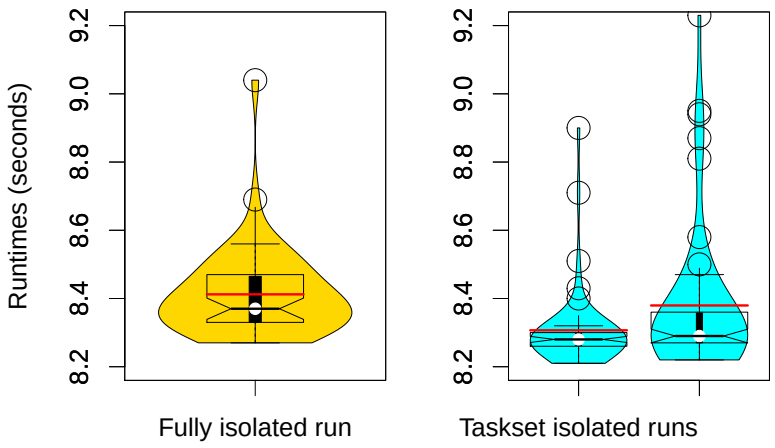


Figure 6.3: Test runs in full isolation (left) and with two sets of runs separated using `taskset` (right)

6.4.1 Implemented Strategies

As has been mentioned previously one of the major factors that is expected to affect runtime is the way that threads are scheduled among the sockets, with the workloads described above designed to measure this. This is the main characteristic that was considered when deciding what strategies to test. The first strategy, if it can truly be called that, allows the plastic task farm to spawn the maximum number of threads, that is 16 in the setup as described above, with all thread placement decisions left to the operating system scheduler.

The second strategy is designed to minimise the work that the scheduler has to do by ensuring that there are always the same number of threads as cores, so threads do not lie in the runnable state waiting for CPU time. In this case when the second application starts, the number of active threads in the first application is reduced by half, with the second application then starting using only 8 threads. Since this implementation of the plastic task farm does not have the ability to change the thread pool size, the second application actually spawns 16 threads, but 8 are immediately put to sleep. When the first application then finishes the 8 sleeping threads in the second application's thread pool are woken up and can get to work.

The third and final strategy, following from the success that LIRA had, adds thread pinning to the reduction in the number of threads. When the second application starts, the threads of the first application are reduced in number and pinned to the cores of the first socket, while the second application has its threads pinned to the second socket. When the first application finished the second application is allowed free reign, with any blocked thread woken up and free to be scheduled over all 16 cores.

6.4.2 Task Granularity

As well as the four different memory access types and the three different strategy types, one other aspect of the plastic task farm that was tested was to find the optimal task granularity, as described in §4.4.2.1. For each of the twelve combinations of work type and strategy type tests were run for a large range of values, with the number of phases in the main `foreach` loop ranging from 1 to 100, with each task consisting of between 312 and 3 elements from the work array.

6.4.3 Results

Figure 6.4 shows the results for the ANTT values for runtimes for the tests that were done over the four different memory access types, three different strategies, and 22 different task granularities. As was mentioned above each set of tests here was only repeated 20 times due to time constraints. The columns represent the different memory access types, with sequential read on the left, followed by sequential write, then random read and random write on the right. The rows give the different strategies, with the 'use-all' strategy at the top, the thread restriction strategy in the middle, and the thread pinning strategy at the bottom. The figure only uses box-and-whisker plots so as to not overly crowd the picture.

The first thing to note from these results is that for the sequential reads and writes neither the strategy used nor the task granularity seem to have a significant result, with the ANTT value staying very closely to the theoretically predicted value of $3/2$ from §5.4.3. This suggests that, while the use of processing resources is diminished by being shared between the two applications, there is very little cache contention, suggesting that in this, admittedly restricted, case plastic parallel programming may not have any benefits. For sequential writes it can be seen that the ANTT value stays pretty consistently around the value 2, with the main effect of the different strategies seeming to be to greatly increase the runtime variability. This high value of the ANTT supports the understanding that the isolated runs make efficient use of the cache, while the inability to fit the entire data array into the L3 cache when two applications are running sequentially means a runtime increase by more than may be expected.

One final point of note is the sharp drop in ANTT for the random write work with the third strategy, with the ANTT decreasing to almost 1. This is a significant result, which suggests that the thread pinning strategy is highly effective when doing the random writing work, and is discussed in more detail below.

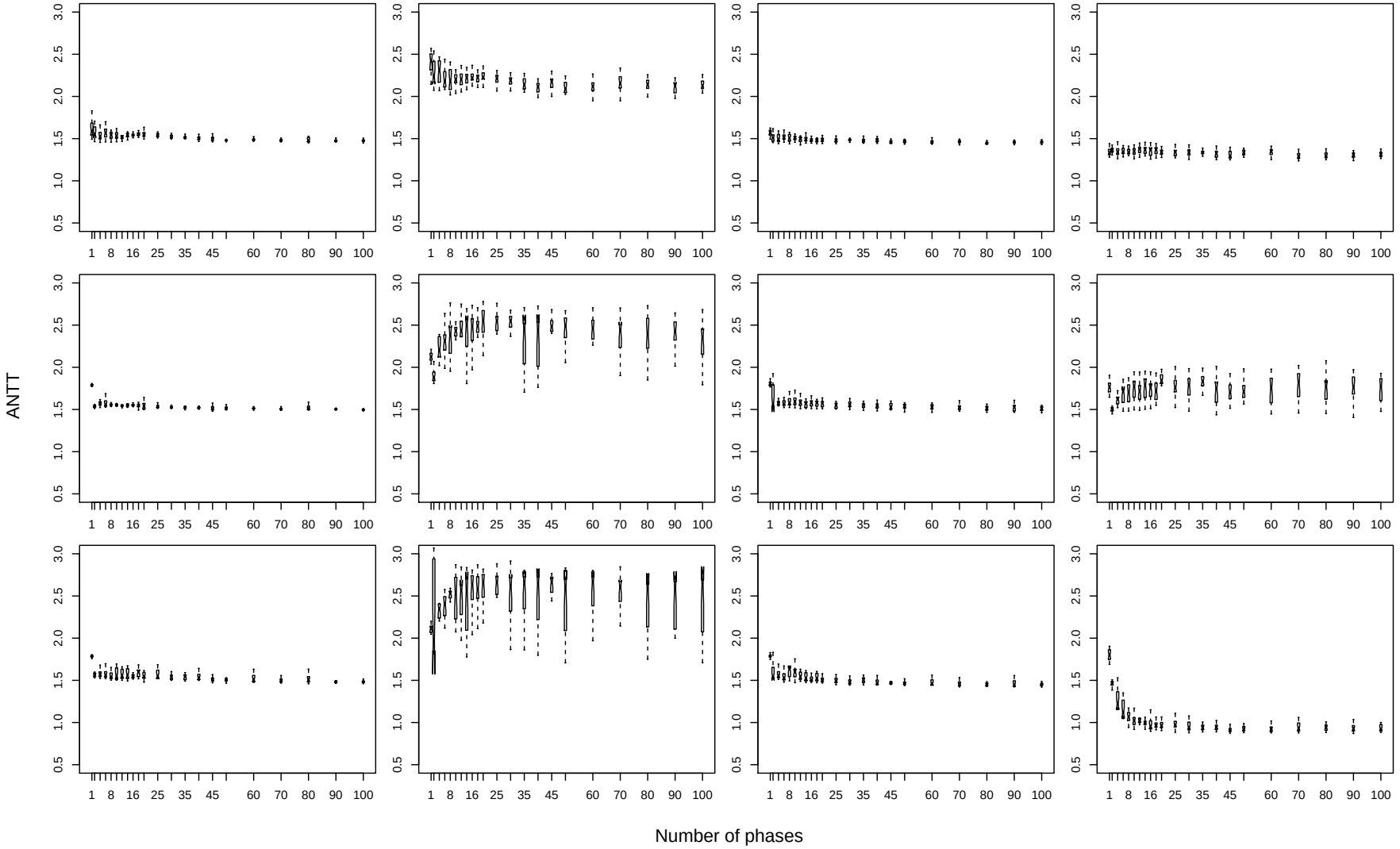
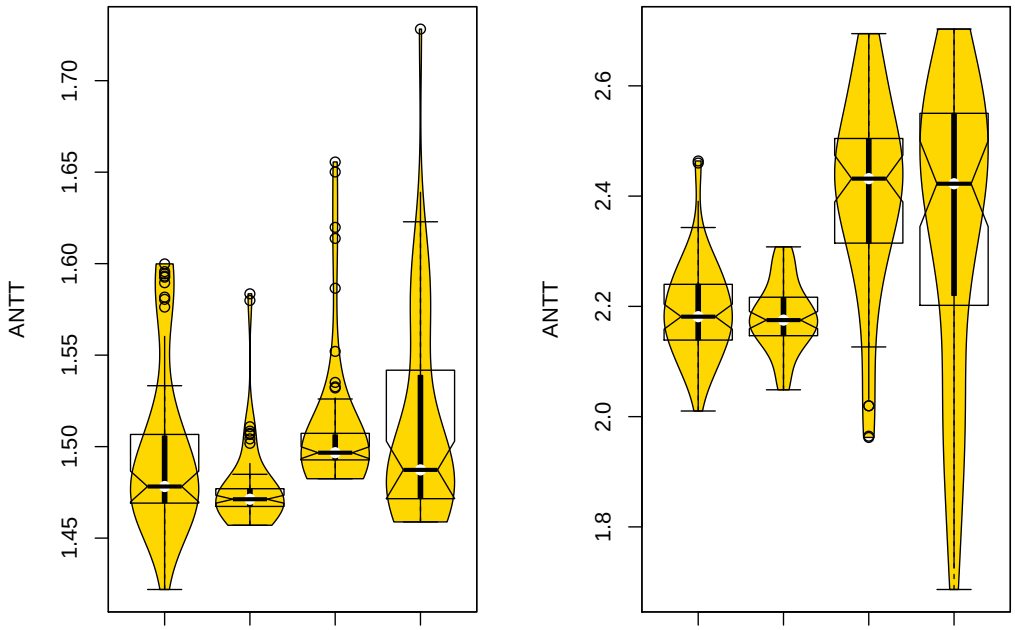


Figure 6.4: ANTT values for different memory access types, strategies and task granularities. The memory access types are sequential read (left column), sequential write (second column), random read (third column) and random write (right column). The strategies are unconstrained (top row), restricted number of threads (middle row) and restricted threads with thread pinning (bottom row)

6.4.4 Closer Inspection

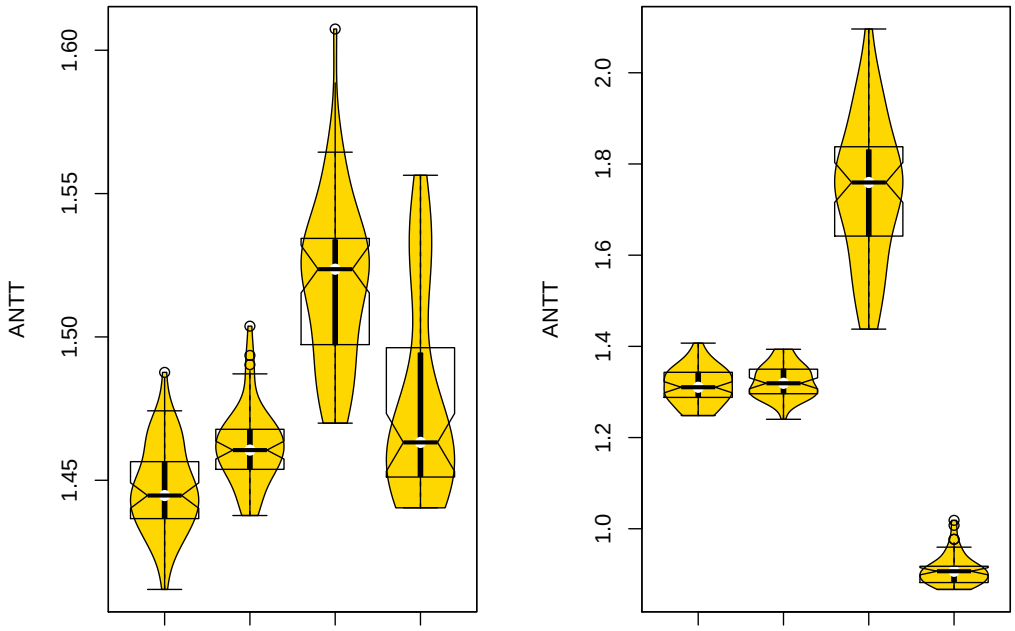
Since it is hard to note anything apart from general trends in figure 6.4, we present here a comparison of the results, only at the highest granularity, from a set of tests that were re-run, with each combination of memory access type and strategy done 50 times in order to bring up the statistical significance of the results, as described in §6.2. The results from this set of tests, including a non-plastic set where the controller was not used, are presented in figure 6.5. From here we see, as described before, that for sequential read there is little real statistically significant deviation from the theoretically expected value of $ANTT = 3/2$, for any of the runtime strategies as well as the case without a controller. For the sequential write we can see that the runtimes without the controller are almost identical to those with the controller but allowing the task farm freedom to spawn the full number of threads and use all of the cores, while both of the thread-restriction strategies give a small, but statistically significant, increase in runtime for the majority of the time, although it is interesting to note that in both cases the outliers tend to lie at *shorter* runtimes rather than longer, suggesting that the restrictive strategies could potentially offer more optimal runtimes, but the scheduler does not often provide the correct conditions.

For the random array access one element that jumps out is that for both the read and the write, the second strategy performs significantly worse than either of the other strategies or the non-plastic case. The reason for this may be that, with the large number of cache misses that are expected with the random reads and writes, threads spend a fair proportion of their time blocked waiting for the cache line to be read in. In these circumstances it may be more efficient to have more threads waiting in the background that can be scheduled while this is happening, with a restricted number of threads removing this option, leading to longer runtimes. A final point, that is possibly the most important in terms of this project, is that for the work that involves random writes to the array, the thread-pinning strategy gives a significant runtime performance improvement of around 30% when compared to the non-plastic case, with a resulting ANTT that is actually below 1. This is remarkable, since it means that on average the applications ran faster when they were running together than when they were in isolation. This suggests that the scheduling that was used when the application was run in isolation was largely sub-optimal, with caching effects resulting in larger runtimes than was necessary, while the restrictions when using the third strategy forced the scheduler to use more optimal thread placement. The importance of thread placement in this case suggests that it is likely to be the most fruitful type of work for this implementation of the plastic task farm, and is investigated more thoroughly in the next section.



(a) ANTT for sequential read

(b) ANTT for sequential write



(c) ANTT for random read

(d) ANTT for random write

Figure 6.5: Distribution of ANTT values for each work type, with second application starting halfway through first application's run. In each plot the left-hand figure is the test with no controller, the next figure has the 'use-everything' strategy, the third figure has the strategy with reduced number of threads and the fourth figure uses the thread pinning strategy.

6.5 Random Write Restricted Run

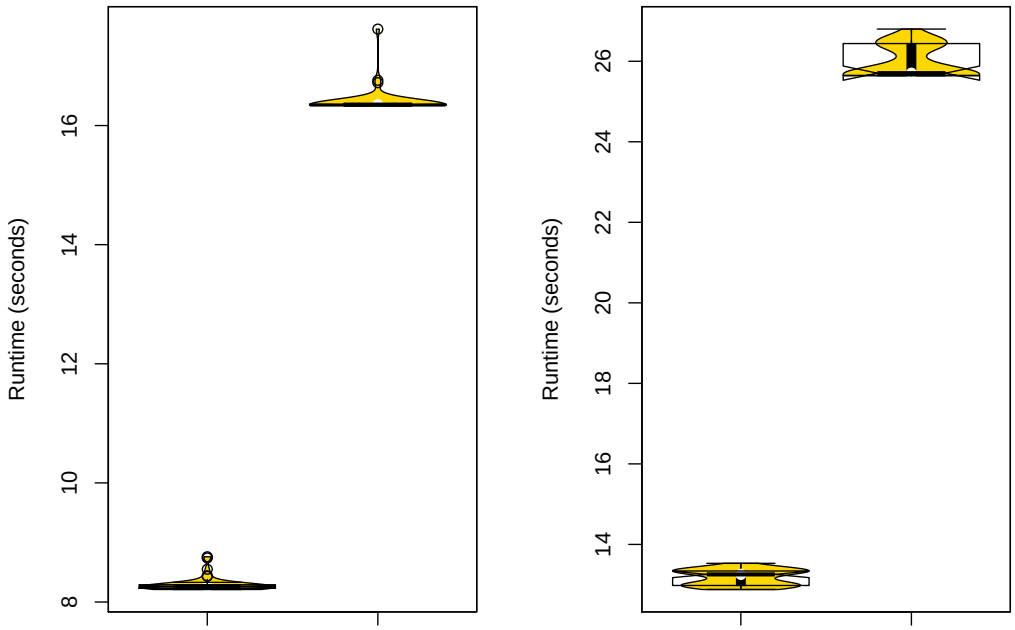
As was noted in the above section, with the runs whose memory access pattern consisted of writes to random addresses, there was effectively no slowdown when two applications ran overlapping using the thread pinning strategy, as opposed to a single application running by itself. This strongly suggests that the single application was in fact running suboptimally, even when in isolation. This can be understood because when it was running in isolation the application was spread over two NUMA sockets, and whenever a memory location on one socket was written to the entire cache line containing that memory location on the other socket had to be marked as invalid. This obviously entails a large amount of overhead, suggesting that it could even be faster to restrict the isolated application to a single socket for such runs.

In order to test this, and confirm that the results were in fact unique to the random write work type, the applications were again run, once again in isolation, but this time with their threads pinned to the cores of a single socket. While this meant that the applications only had half the number of working threads, which naively one may expect would mean they took twice as long to run, from the results in the previous section it may be expected that for the random write workload, inter-socket effects may dominate this difference. The results are shown in figure 6.6, with each plot showing the original run, using sixteen threads spread over two sockets, on the left and the second run, with eight threads on a single socket, shown on the right. Clearly from this figure the random write workload is unique in that it actually runs faster when restricted to a single socket, suggesting that the overhead associated with cache invalidation is greater than any gains made by running with more threads.

6.6 Combined Work Type Application

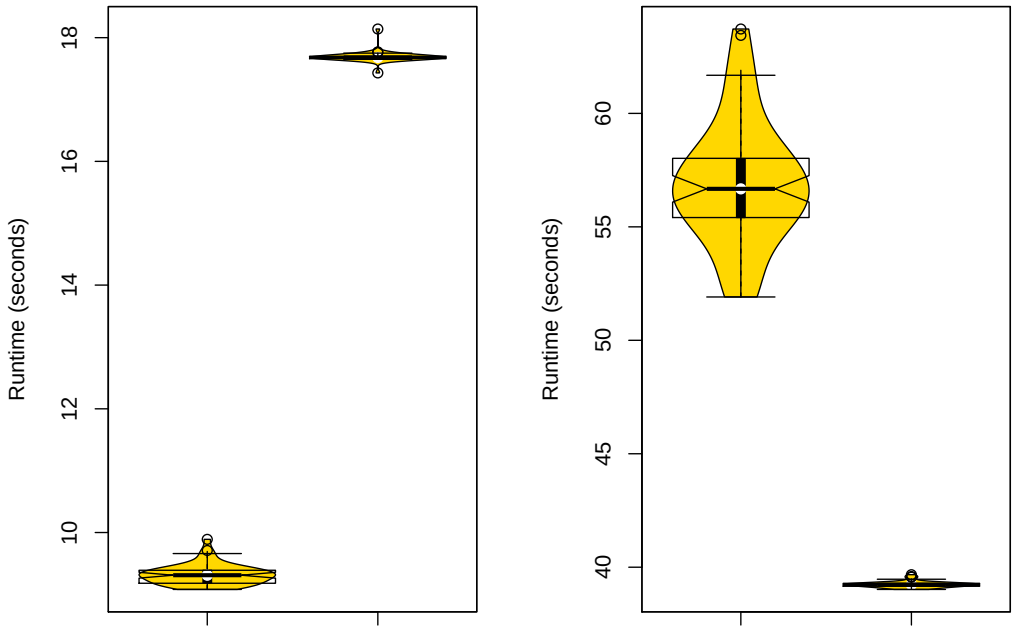
So far we have been looking at highly simplified workloads, consisting of a single memory access type, and have found that while most workloads benefit from having access to more threads and sockets, when random writes to an array are involved it is actually better to restrict the application to running on a single socket, meaning that there is no need for costly inter-socket communication. These workloads do not however really demonstrate the power of plastic parallel programming, since even with the final workload that showed an improvement when the pinning strategy was used, the strategy would have been unnecessary if each application had been pinned to a single socket in the first place.

In order to demonstrate the true power of plastic parallel programming a final set of tests were run, using an application that used a mixture of different workloads. This application spends around the first half of its runtime doing work that involves sequential reads, and the second half of the time doing work with random writes. As we have seen these two types of work have very different conditions for optimal running, with sequential reads best with more



(a) Runtime for sequential read

(b) Runtime for sequential write



(c) Runtime for random read

(d) Runtime for random write

Figure 6.6: Runtimes for the four different work types, each with a single application running isolation. In each plot the left-hand figure gives the application allowed access to two sockets, running with sixteen threads. The right-hand figure shows the application restricted to a single socket with eight threads.

threads and spread over multiple sockets, and random writes best restricted to a single socket. This means that an optimal strategy may be to run each application with as many threads as possible during the first work set, then restricted to a single socket during the second work set.

6.6.1 Baseline Tests

An initial set of tests were run on this application in isolation, in order to get baseline values, in the same way as previous tests. In all of the cases each test was repeated 50 times. The first set of tests had the application pinned to a single socket, that is 8 CPU cores, which we have seen is ideal for the second random write work type. The second set of tests had the application allowed access to two sockets, which is best for the sequential reads. Both of these cases were run without any controller. The third set of tests involved the controller, which instructed the application to switch from using two sockets for the first workload to using a single socket for the second workload, giving fully optimal conditions. The results of these three sets of tests are shown in figure 6.7 as the three left-hand plots, with median values 61, 49 and 40 seconds respectively. As can be seen the slowest run was when the application was restricted to one socket, suggesting that the runtime improvement that comes about using two rather than one sockets for the first workload is greater than the penalty that is imposed on the second workload. This is not surprising since as we saw in figure 6.6 that the sequential read ran twice as quickly when using two cores rather than one, while the random read only ran around 43% faster when running on one socket as opposed to two.

While previous results have been demonstrated in terms of the ANTT, there is a problem here in that the ANTT needs a single baseline runtime for the calculate. While the previous tests were done with a baseline of the application running using full resources, here we would like to make as comprehensive a comparison as possible, so ideally the multiple-application runs should be compared with all forms of the single-application runs. For this reason rather than the ANTT, the average runtime is used. Since both of the running applications are identical it is easy to work out what the ANTT would be for any given baseline run in this case. As a guideline the plots for the individual application runs have been marked with a green horizontal line at $3/2$ times the median runtime, which represents the expected ideal runtime for the concurrently run applications given the heuristic arguments from §5.4.3.

6.6.2 Concurrent Applications

Having run the baseline tests, tests were done with two sets of the application running concurrently. In order to add more realism to the tests, in each case the first application was started, then the second application started some time uniformly distributed between 2 and 48 seconds later. In the first set of runs there was no controlling application, so both applications ran spread over the entire two sockets. In the second set the strategy was used that each application was

restricted to a single socket upon finishing the first piece of work, a strategy which proved optimal in the isolated runs. The average runtime for the two applications in each case is shown by the fourth and fifth plot in figure 6.7, with median values of 73 and 72 seconds. It is rather surprising that using the strategy that was optimal in the single-application cases does not produce any significant improvement over not using the controller at all. When the reason for this was investigated it was noted that the second application almost always took significantly longer to run than the first application. The reason for this seems to be the following. It has been noted that the runtime decrease by doubling the number of sockets from one to two in the first sequential reading workload is significantly greater than that gained by halving the number of sockets, from two to one, for the second random write workload. Since, on average, the second application starts half way through the first application, then on average the first application has two sockets on which to run its first workload, then a single socket on which to run the second workload, thus fully optimal conditions. However the second applications has to share the sockets while running its first workload, and has no advantage during its second workload as it cannot utilise the extra socket when the first application exits. This means that there is no significant different between the runs without the controller and with this first strategy.

This understanding of the reasons for suboptimal behaviour suggested that a more complex strategy would be needed, which was implemented as follows. When the first application started, it got full access to the two sockets, with optimal conditions for running its first workload. If the first application reached the second workload before the second application had started then it was restricted to the first socket, the same as when it was running in isolation. If the second application started while the first was still doing its first workload, then both of them were allowed full access to the resources, letting the operating system schedule them as it willed. If, however, the first application started its second workload while the second application was running, or the second application started running while the first was doing its second workload, the first application was restricted to only four of the cores on the first socket, allowing the second application to use the remaining twelve cores, giving it a significant speedup of its first workload. When the second application then started the second workload, each application was pinned to a single socket, which once again was the most optimal setup. The results from this set of tests are shown in the right-hand plot of figure 6.7, with a median of 56 seconds. As can be seen there is a substantial improvement in runtime compared with either of the other multi-application cases, with the average runtime of the two applications actually being around the same as the runtime for the single application when constrained to one socket. This seems to represent a case where plastic parallel programming produces a more optimal result that can be achieved through any static means.

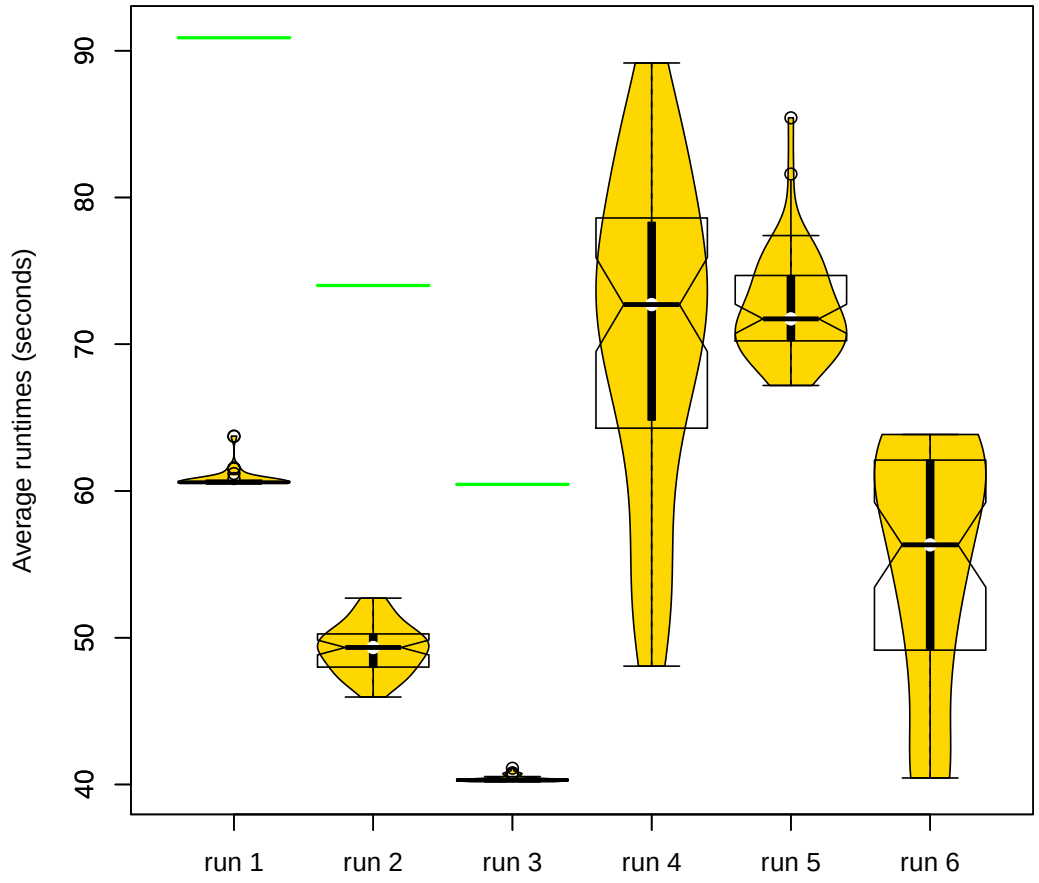


Figure 6.7: Average runtimes for the tests described in §6.6. The three left-hand figures are runtimes for individual applications, the three right-hand figures are average runtimes for the concurrent applications. The green lines show $3/2$ times the median runtime for the individual applications, representing the ideal ANTT values as described in §5.4.3.

Chapter 7

Results overview

In the previous chapter we presented the results from a large number of tests, some more successful than others, which were aimed at finding situations in which plastic parallel programming could be reliably shown to improve overall runtime performance of concurrently running applications. We showed how different work types required different strategies in order two run optimally, with caching behaviours being the main area that was focused on. It was seen how workloads that involved sequential access to an array, as well as random reading, which have the ability to efficiently make use of cache lines, are generally well served by being allowed more resources, up to a certain point, with the operating system scheduler giving efficient thread placement. In cases where the cache could not be relied up however, in particular the workload the involved writing to random array elements, tailoring the scheduling policy to the details of the system become much more important. For simple applications that largely involve one work type it may be possible to do this statically, however as the final set of results showed, with more realistic applications that involve multiple types of work in an environment with random starting and stopping of other applications, having dynamic control becomes much more important. The final strategy of the previous section managed to give a reduction of around 23% to the average runtime of the two applications, which is a significant result in a world where vast amounts of research are going into methods to squeeze every last drop out of the available resources.

Chapter 8

Conclusion

This dissertation has discussed the creation and testing of an implementation of a plastic task farm. This extends the well-known task farm parallel design by adding plasticity, that is the ability to dynamically alter details of the implementation to best suit the environment that it is running in. By presenting a high-level interface the plastic task farm is able to hide many of the details of its implementation, thus allowing it to pick optimal strategies depending on the hardware and environment. A large number of tests were run in order to verify that this method of optimisation is useful, with results presented in a way that is aimed to convince the reader that any differences are in fact statistically significant.

The set of tests described in §6.4 was done on highly simplified applications, each of which did work involving a particular memory access type and pattern, with either sequential or random access, and involving either reads or writes. From these tests it was found that the operating system scheduler manages very well in the majority of cases, however there are situations where explicit control of the program threads is important, that being the random writes workload in this case. While optimal behaviour in all of this first set of applications could in fact be achieved through simple thread affinity settings throughout the lifetime of the application, the tests gave valuable insights that were used later on. With the more complex application that was introduced in §6.6, which combined the sequential read and random write workloads, it has been shown that these 'one-size-fits-all' strategies no longer work, and demonstrate when plastic parallel programming really comes into its own. By carefully tailoring the strategies to the application and the hardware that the application ran on it was possible to significantly reduce the overall runtime.

As we have seen from the results in the previous chapter, the efficacy of plastic parallel programming depends strongly on the details of the application that is being run. While this project has only looked at a couple of different application varieties, concentrating on different memory access types, there are countless other categories of application, including CPU bound vs I/O bound, embarrassingly parallel vs inherently sequential, distributed vs local etc. Since

incorrect strategy choices can easily decrease performance rather than increasing it, and the interaction of different application types with the hardware is very hard to predict, it would take a lot of work to find strategies tailored for every case that is likely to be encountered. The positive results that have been shown in the last set of results however suggest that this is a worthwhile task, and could provide large benefits in a world where the efficient use of computing resources is becoming ever more important.

8.0.3 Potential for Future Work

In this dissertation we have dipped our toes into the possibilities that are available with plastic parallel programming, showing that the plastic task farm has the ability to provide runtime improvements in a small subset of toy applications under highly controlled test conditions. There are thus a multitude of ways in which plastic parallel programming could be extended. There are many design patterns, each aimed at a different parallel situation, that could be implemented with plastic parallel programming, including geometric decomposition patterns, or the divide and conquer pattern among many others [see e.g. 20].

As well as a multitude of design patterns to choose from, there are an essentially unlimited number of strategies that could be used for different application types. While it may be possible to deduce optimal strategies theoretically for some simple cases, it may be necessary to do many simulations in order to find more general categories of application for which a given strategy is optimal, especially when more than two applications are involved. It may be possible to use something similar to the autotuner that was used by PetaBricks [21], although the complexity may become far too great and require much more sophisticated methods. As well as thread placement strategies, there are many other types such as data structure and distribution strategies, some of which were mentioned in §4.1.

One more way in which plastic parallel programming may be adjusted is by altering the quantity that is being optimised for. The current iteration optimises on what is pretty much the simplest metric, the runtime, which in many ways is no longer the factor that is most important nowadays. The rapid increase in large server farms has brought the problem of dissipating the heat they produce to the forefront, with some estimates suggesting that around 40% of the energy consumed is used for removing heat [52]. Power usage and heat dissipation in mobile devices is also a major concern, severely limiting the ability to create ever faster, smaller gadgets [53]. This means that strategies that minimise power consumption could have massive positive environmental and economical impacts. While being even more challenging to determine than optimal strategies for minimising runtime, the benefits of plastic parallel programming in this area could potentially be huge.

Bibliography

[1] Edward A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42.

[2] Krste Asanovic et al. “A View of the Parallel Computing Landscape”. In: *Commun. ACM* 52.10 (2009), pp. 56–67.

[3] James Larus and Dennis Gannon. “Multicore Computing and Scientific Discovery”. In: The Fourth Paradigm: Data-Intensive Scientific Discovery (2009), pp. 125–129.

[4] John C Wooley and Herbert S. Lin. “Computational modeling and simulation as enablers for biological discovery”. In: *Catalyzing Inquiry at the Interface of Computing and Biology [online]*. National Academies Press, 2005, pp. 117–202.

[5] Alfio Quarteroni. “Mathematical models in science and engineering”. In: *Notices of the AMS* 56.1 (2009), pp. 10–19.

[6] Isabella von Sivers et al. “Humans do not always act selfishly: social identity and helping in emergengy evacuation simulation”. In: *Transportation Research Procedia* 2 (2014), pp. 585–593.

[7] Philip Esper. “The role of computer modelling and E-engineering in civil, structural and geotechnical engineering”. In: *2006 2nd International Conference of Information & Communication Technologies*. Vol. 1. IEEE. 2006, pp. 7–11.

[8] *ARCHER national supercomputing service*. URL: `https://www.epcc.ed.ac.uk/facilities/archer` (visited on 08/12/2016).

[9] *Open Grid Scheduler*. URL: `http://gridscheduler.sourceforce.net` (visited on 08/12/2016).

[10] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-53086-4.

[11] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[12] C Gordon Bell. “Fundamentals of time shared computers”. In: *Computer Design* 7.2 (1968), pp. 44–59.

[13] Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. 2009. URL: http://www.ibm.com/developerworks/library/l-completely-fair-scheduler (visited on 03/29/2016).

[14] Steve M. Easterbrook. "Climate change: a grand software challenge". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 99–104.

[15] Peter Welch, Jon Kerridge, and Fred Barnes. "Classification of Programming Errors in Parallel Message Passing Systems". In: *Communicating Process Architectures 2006: WoTUG-29: Proceedings of the 29th WoTUG Technical Meeting, 17-20 September, Napier University, Edinburgh, Scotland*. Vol. 64. IOS Press. 2006, p. 363.

[16] Joab Jackson. *Nasdaq's Facebook glitch came from race conditions*. 2012. URL: http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html (visited on 08/12/2016).

[17] Patrice Godefroid and Nachiappan Nagappan. "Concurrency at Microsoft: An exploratory survey". In: *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*. 2008. URL: https://www.microsoft.com/en-us/research/publication/concurrency-at-microsoft-an-exploratory-survey/.

[18] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Vol. 2. Oxford University Press, 1977.

[19] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2005. ISBN: 9781405837309.

[20] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[21] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 38–49.

[22] *The Petabricks System*. Massachusetts Institute of Technology. 2012. URL: http://projects.csail.mit.edu/petabricks (visited on 01/04/2016).

[23] *Intel® Xeon® Processor E7 Family: Reliability, Availability and Serviceability*. Intel. 2014. URL: http://www.intel.co.uk/content/dam/www/public/us/en/documents/white-papers/xeon-e7-family-ras-server-paper.pdf (visited on 08/12/2016).

[24] Alexander Collins et al. “LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems”. In: *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM. 2015, p. 2.

[25] Horacio González-Vélez. “Self-adaptive skeletal task farm for computational grids”. In: *Parallel Computing* 32.7 (2006), pp. 479–490.

[26] Marco Danelutto. “Adaptive task farm implementation strategies”. In: *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*. IEEE. 2004, pp. 416–423.

[27] Ranieri Baraglia et al. “An optimized task-farm model to integrate reduced dimensionality Schrödinger equations on distributed memory architectures”. In: *Future Generation Computer Systems* 15.4 (1999), pp. 497–512.

[28] Jan Dünnweber et al. “Making a Task Farm Component Parallelize Loops for the Grid”. In: *Integrated Research in Grid Computing CoreGRID Integration Workshop*. 2006, p. 93.

[29] *Slurm task-farming*. URL: `https://www.tchpc.tcd.ie/node/1127` (visited on 08/13/2016).

[30] *Irish Centre for High-End Computing: Task Farm*. URL: `https://www.ichec.ie/support/documentation/task_farming` (visited on 08/13/2016).

[31] *A simple MPI task farm*. URL: `http://www.inf.ed.ac.uk/teaching/courses/ppls/farm.c` (visited on 08/13/2016).

[32] *pthreads - POSIX threads*. URL: `http://man7.org/linux/man-pages/man7/pthreads.7.html` (visited on 08/14/2016).

[33] Robert D Blumofe and Charles E Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[34] Radhika Thekkath and Susan J Eggers. “Impact of sharing-based thread placement on multithreaded architectures”. In: *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*. IEEE. 1994, pp. 176–186.

[35] Guy E Blelloch and Phillip B Gibbons. “Effectively sharing a cache among threads”. In: *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2004, pp. 235–244.

[36] David Ott. *Optimizing Applications for NUMA*. 2011. URL: https://software.intel.com/en-us/articles/optimizing-applications-for-numa (visited on 08/13/2016).

[37] Robert L McGregor, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. "Scheduling algorithms for effective thread pairing on hybrid multiprocessors". In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 28a–28a.

[38] Rajkumar Buyya et al. "Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost–time optimization algorithm". In: *Software: Practice and Experience* 35.5 (2005), pp. 491–512.

[39] *ØMQ*. iMatix Corporation. 2014. URL: http://www.zeromq.org (visited on 03/14/2016).

[40] *omp barrier*. URL: https://software.intel.com/en-us/node/524510 (visited on 08/13/2016).

[41] Christoph Lameter. "Numa (non-uniform memory access): An overview". In: *Queue* 11.7 (2013), p. 40.

[42] *Intel® Xeon® Processor L7555*. URL: http://ark.intel.com/products/46494/Intel-Xeon-Processor-L7555-24M-Cache-1_86-GHz-5_86-GTs-Intel-QPI (visited on 08/13/2016).

[43] *Dell PowerEdge R810*. URL: https://www.dell.com/downloads/global/products/pedge/pedge_r810_specsheet_en.pdf (visited on 08/14/2016).

[44] John PA Ioannidis. "Why most published research findings are false". In: *PLoS Med* 2.8 (2005), e124.

[45] Carol Kilkenny et al. "Survey of the quality of experimental design, statistical analysis and reporting of research using animals". In: *PloS one* 4.11 (2009), e7824.

[46] Michael J Marino. "The use and misuse of statistical methodologies in pharmacology research". In: *Biochemical pharmacology* 87.1 (2014), pp. 78–92.

[47] Analytical Methods Committee et al. "Robust statistics: a method of coping with outliers". In: *Technical brief* 6 (2001).

[48] John M. Chambers et al. *Graphical methods for data analysis*. Wadsworth, 1983.

[49] R Core Team. *boxplot.stats - R documentation*. R Foundation for Statistical Computing. 2013. URL: https://stat.ethz.ch/R-manual/R-devel/library/grDevices/html/boxplot.stats.html (visited on 08/08/2016).

[50] Lieven Eeckhout. “Computer architecture performance evaluation methods”. In: *Synthesis Lectures on Computer Architecture* 5.1 (2010), pp. 1–145.

[51] *Issues in Estimating Sample Size for Confidence Intervals Estimates*. URL: http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Power/BS704_Power2.html (visited on 08/15/2016).

[52] Z Song, X Zhang, and C Eriksson. “Data Center Energy and Cost Saving Evaluation”. In: *Energy Procedia* 75 (2015), pp. 1255–1260.

[53] Arden L Moore and Li Shi. “Emerging challenges and materials for thermal management of electronics”. In: *Materials Today* 17.4 (2014), pp. 163–174.