NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Adaptive Scheduling in Heterogeneous Distributed Computing Systems

Andrew J. Page

Ph.D Thesis

Supervisor: Thomas J. Naughton

Department of Computer Science

Faculty of Science

National University of Ireland, Maynooth

Maynooth, Co.Kildare, Ireland

September, 2009

To Niamh.

Declaration

This thesis has not been submitted in whole or in part to this or any other university for any other degree and is, except where otherwise stated, the original work of the author.

Signed: _____

Andrew J. Page

Acknowledgments

I would like to thank Thomas J. Naughton for his many years of supervision. I would also like to thank Dr. Lukas Ahrenberg for reviewing this thesis and for his many helpful suggestions. Professor Enrique Alba provided very helpful feedback for future directions for this research. This research was funded by the Embark Initiative from the Irish Research Council for Science, Engineering and Technology under the National Development Plan.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The main focus of this research is in the area of adaptive scheduling for heterogeneous distributed systems. Given an unreliable, non-dedicated set of processing and communication resources, a scheduler is required to allocate tasks to processors. No information about the state of the system, which can vary over time, or the tasks to be processed, is known in advance and thus must be estimated dynamically. Current schedulers do not adequately address this dynamism. To address this, a property estimation method is presented, which utilizes a k-Nearest Neighbours algorithm, a smoothed average and an analytical benchmark. These estimated properties are then used by two different scheduling techniques, which make less restrictive assumptions than the current state-of-the-art methods. A multi-heuristic evolutionary method utilizes a genetic algorithm and eight simple heuristics to efficiently allocate tasks to processors. A deterministic method utilizes the error inherent in estimating the properties of the system and the execution time of tasks, to allocate tasks to processors. The algorithms have been implemented on a real-world heterogeneous distributed system with up to 150 processors. A set of real-world problems from the areas of cryptography, bioinformatics, and biomedical engineering were used as a test set to measure the effectiveness of the scheduling algorithms. Experiments have shown that both methods achieve better efficiency than other state-of-the-art heuristic algorithms. Finally, a low memory distributed reconstruction application for large digital holograms is presented, which has significantly increased the size of holograms that can be reconstructed, over the previous state-of-the-art.

Preface

Parts of this thesis have also been published in the following articles:

Journals

Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Scheduling in a dynamic heterogeneous distributed system using estimation error. *Journal of Parallel and Distributed Computing*, Vol. 68, Issue 11, November 2008, 1452-1462.

Andrew J. Page, Lukas Ahrenberg, and Thomas J. Naughton. Low memory distributed reconstruction of large digital holograms. *Optics Express*, 16(3):1990–1995, 2008.

Andrew J. Page and Thomas J. Naughton. Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. *Artificial Intelligence Review*, 24(3-4):415–429, Nov 2005.

Lukas Ahrenberg, Andrew J. Page, Bryan Hennelly, John McDonald, and Thomas J. Naughton. Using commodity graphics hardware for real-time digital hologram view reconstruction. *IEEE/OSA Journal of Display Technology*, Vol. 5, No. 1, 2009.

Thomas M. Keane, Andrew J. Page, Thomas J. Naughton, Simon A. Travers, and James O. McInerney. Building large phylogenetic trees on coarse-grained parallel machines. *Algorithmica, Special issue on Coarse Grained Parallel Algorithms for Scientific Applications*, 45(3):285–300, July 2006.

Conferences

Andrew J. Page and Thomas J. Naughton. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 189.1, Denver, CO, USA, 2005.

Andrew J. Page, Shirley Coyle, Thomas M. Keane, Thomas J. Naughton, Charles Markham, and Tomas Ward. Distributed monte carlo simulation of light transportation in tissue. In *proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, pages 1–4, Rhodes, Greece, April 2006. IEEE Computer Society.

Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Adaptive scheduling across a distributed computation platform. In J. P. Morrisson, editor, *Third International Symposium on Parallel and Distributed Computing*, pages 141–149, Cork, Ireland, July 2004. IEEE Computer Society.

Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Bioinformatics on a heterogeneous java distributed system. In *Proceedings of the 19th IEEE/ACM International Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 2005. IEEE Computer Society.

Thomas M. Keane, Andrew J. Page, James O. McInerney, and Thomas J.

Naughton. A high-throughput bioinformatics distributed computing platform. In *Bioinformatics and its Medical Applications Special Track, The 18th IEEE International Symposium on Computer-Based Medical Systems*, pages 377–382, Dublin, Ireland, June 2005.

Chapter 1

Introduction

Modern scientific research has ever-increasing computational requirements. Many of the large problems being tackled are ideal candidates for parallelization [19]. Distributed computing can provide a large amount of computational resources by utilizing the spare clock cycles of existing personal computers (PCs), without the cost of expensive dedicated parallel machines. Computers with different processor speeds and memory sizes can be brought together to form a virtual supercomputer. However, the distributed nature of the underlying resources presents problems not present in closely coupled systems, such as communication overheads, or heterogeneity of resources. A poor allocation of tasks to processors could nullify the benefits of using a distributed system by inefficiently utilizing the systems resources.

We wish to map tasks to processors in a dynamic heterogeneous distributed system where the resources are constantly varying and no knowledge of the system is available a priori. The task allocation problem (TA)

is NP-complete in the general case [100]. With the addition of a single dynamic element, which we refer to as the dynamic task allocation problem (DTA), it appears that a solution to the problem cannot be verified in polynomial time and that the DTA problem is not in NP (shown in Appendix A and B). Heuristics must be used to generate a solution in a realistic amount of time. Solutions from existing efficient algorithms (such as [15]) for problems classified as NP-complete (such as TSP and 3SAT) cannot be polynomially transformed to solve, in polynomial time, the problem tackled in this thesis.

Many scheduling algorithms (other than the most trivial) utilize knowledge of the available system resources and the tasks to be processed when deciding to allocate a task to a processor [7, 21, 23, 61, 92, 96, 98, 103]. How to best generate this knowledge is an open problem [98]. In general cases, all information used when deciding to allocate tasks to processors must be estimated. This, of course, is error-prone, with the errors in these estimations introducing inefficiency. The most common forms of estimating task execution times are by benchmarking a task or set of tasks offline in advance [7, 21, 92, 96, 98], or requiring a person to supply a directed acyclic graph with: task, communication information and precedence constraints in advance [56, 57]. The heterogeneous and non-dedicated nature of the resources in a loosely-coupled distributed system means that these types of estimation can be detrimental to accuracy and contribute to large margins of error, between the actual execution time and the estimated execution time of a set of tasks. One of the contributions of this thesis (explained later) is the introduction of a new approach to estimation in distributed computing

scheduling.

One technique used to address DTA problem, when designing scheduling heuristics, has been to simplify the problem by adding restrictive assumptions. For example:

- a priori knowledge of communication times and task processing times [1, 7, 10, 16, 21, 24, 38, 55, 56, 57, 92, 96, 98, 102, 106],
- homogeneous processing or communication resources [16, 34, 38, 52, 54, 91, 96, 102, 107],
- the state of the system does not change during run-time [1, 10, 38, 59, 98, 106],
- all messages are passed instantaneously, [96, 107],
- and resources are dedicated exclusively to the distributed system [1, 38, 45, 90, 98, 105, 106, 107, 108].

An overview of the properties of the schedulers referenced in this section can be found in Table A.1 and A.2. Schedulers have been classified by a number of properties, with each property limiting the generality of the scheduling technique in some way, which in turn limits the usefulness of the technique to a subset of problems. Next, each of these properties will be discussed.

Static scheduling refers to a schedule which is created before run-time and cannot change. The opposite is dynamic scheduling where the schedule can change during run-time, and thus can adapt to variations in available resources. Some schedulers were designed to use only homogeneous resources.

This restriction, in some cases, fundamentally changes the complexity of the problem being tackled to the P complexity class. The ability to use heterogeneous resources may open up a larger pool of resources, and more accurately models the available resources in loosely coupled distributed computing. Pre-emptive scheduling refers to the ability of the scheduler to move partially computed tasks from one processor to another. This flexibility may reduce the overall total execution time allowing for tasks to be moved from slow processors to idle faster processors as they become available.

A priori knowledge of communication times and task processing times limits the generality of schedulers, because advanced knowledge is needed about the state and operation of the system and the tasks to be processed. Some schedulers are specifically optimized to suit certain architectures, for example: Kwok and Ahmed [55], and Lee and Zomaya [59] require a fully connected network; Mohapatra [66] requires a hypercube network; Hamidzadeh *et al.* [34] assume a common shared memory is available; and Nagar *et al.* [68] require a specific characteristic of the Solaris kernel. These requirements limit the schedulers to certain operating systems, architectures and topologies, reducing their generality.

Assuming that the processing and communication resources are completely dedicated to the distributed system (e.g. a cluster) greatly simplifies the problem of scheduling, by creating a closed controlled environment. The behavior of the resources becomes predictable, which can be easily factored into a scheduling algorithm. The inclusion of non-dedicated resources, such as using the spare clock cycles of desktop PCs connected by the internet, can

greatly increase the amount of computational resources available. These non-dedicated resources are however unpredictable, and their availability can vary due to external events outside of one's control. These additional parameters greatly increase the complexity of the problem.

Kwok and Ahmed surveyed 27 different static scheduling algorithms, and proposed a taxonomy that classifies these algorithms into different categories [57]. Kwok and Ahmed also proposed a set of benchmarks to compare 15 different static scheduling algorithms on a homogeneous set of processors [56]. At the other extreme Maheswaran *et al.* surveyed 8 different schedulers for a dynamic heterogeneous distributed system [61].

Some distributed systems (detailed in Table A), such as SETI@home [52], ignore the resources of the system [52, 54, 91], or treat their heterogeneous resources as a homogeneous set [6, 22, 52, 54, 74, 91, 101] by ignoring variation in the available computational resources of the processors. Some of the assumptions made simplify the problem but fundamentally change the complexity class. This limits the applicability of the scheduling algorithms to specific special cases. It is our belief that if a scheduler is to be applicable to real-world distributed computing environments and problems, then it should not make any prior assumptions about resource homogeneity or availability.

Research has been done to address some of these restrictive assumptions. Sinnen *et al.* [92] look at a processor's involvement in communication and show that considering this involvement, when scheduling, leads to more efficient resource utilization in real-world distributed systems. Cohen *et al.* [13] focus on scheduling the communication between processors, to minimize the

data transfer overheads in a distributed system. Theys *et al.* [98] generate and store many scheduling solutions before run-time, then select the most suitable solution during run-time, which allows the scheduler to adapt to a variable task and resource environment. The dynamic level scheduling algorithm proposed by Dogan and Ozguner [23] addresses the variability of network and processor resources caused by failures, and attempts to minimize the probability of these failures adversely affecting the overall operation of the distributed system. Ali *et al.* [5] create a generalized robustness metric for unreliable parallel and distributed systems where the system resources may vary or the estimated task execution times may be erroneous.

Another method is to use complex evolutionary scheduling heuristics, such as genetic algorithms (GAs) [37], simulated annealing (SA) [51], Tabu [30] and Ant Colony search optimization [14]. This allows for the fast exploration of the search space of possible schedules. Near optimal solutions can be found quickly and the scheduler can be applied to more general problems [96]. Scheduling algorithms based on GAs have been shown to consistently generate more efficient solutions than other evolutionary strategies when applied to scheduling in heterogeneous distributed systems [10].

We have broken up the DTA problem into two parts: 1.) generating accurate estimates of the system resources and the properties of the tasks to be processed, and 2.) allocating tasks to processors.

We estimate the system properties and the resource requirements of the problems to be processed based on historical information. We use a k nearest neighbours method (k-NN) [17] combined with a smoothed average to

estimate the value of non-linear properties.

These estimates are then used as inputs into scheduling heuristics. We set out to create a robust scheduler, which could produce high quality solutions in unknown resource environments. GAs fulfilled this requirement, having been shown to work well [10] in dynamic heterogeneous distributed systems. A GA takes a set of different solutions to a problem, and in successive iterations, keeps the best solutions and uses them to generate a new set of solutions. This survival-of-the-fittest method closely models evolution in the natural world. The evolutionary nature of GAs allows for a fast traversal of the search space and for efficient solutions to be produced quickly. Eight simple heuristics are utilized to enhance the initialization of the GA based scheduler. This means that the GA scheduler is no worse than the best simple heuristic, allowing for efficient schedules to be produced in polynomial time.

While the GA based scheduler works well in many situations, it does have a number of disadvantages, notably predictability and verifiability. Since it contains randomness, the same set of inputs may not give the same output solution. It is not possible to definitively know the output solution in advance. Likewise, the running time needed to achieve a solution can vary, although given enough time it may evolve to a very good solution. This makes it unsuitable for situations with deadlines. Due to the difficultly in verifying the output, or even understanding why a particular solution has been reached, GAs are unsuitable for some applications, such as medical applications.

A simpler solution has been developed which addresses these disadvantages. The uncertainty in the estimation of the properties of the system is

utilized to produce efficient schedules. We seek to schedule either the tasks with the minimum uncertainty or the tasks with the most uncertainty earliest. When this is combined with different objectives, such as minimizing makespan (total execution time) and evenly distributing load, it naturally gives rise to a family of four different scheduling algorithms. It has been shown to produce solutions which are nearly as good as more complex evolutionary schedulers. It has a constant running time, so is suitable for applications with deadlines. There is no randomness, thus the same solution will be given for the same set of inputs. Its predictable and repeatable properties mean it can be verified, thus allowing for it to be used in situations not suited to an evolutionary algorithm.

Finally a real-world distributed application is presented. It is a low memory distributed reconstruction application for large digital holograms [3, 76] from the optical physics field. It has allowed for the reconstruction of 4.3 gigapixel digital holograms on low powered, desktop PCs; the previous largest recontructions where of the order of 0.2 gigapixels.

Chapter 2 specifies how to estimate task execution times and system properties. A heterogeneous distributed system and a set of problems are presented. These are used in the evaluation of the scheduling algorithms. Chapter 3 presents a multi-heuristic genetic algorithm scheduler. Chapter 4 uses estimation error to schedule tasks to processors. Chapter 5 describes a distributed application for reconstructing large digital holograms. Finally we conclude in Chapter 6.

Chapter 2

Estimating properties

Parts of this chapter have also been published in the following articles [49, 50, 78, 79, 80]. We will present a technique to estimate task execution times, and present a set of problems used to test our algorithms. These are then used in the following chapters.

The properties and availability of heterogeneous computational and communication resources can vary randomly over time, with unknown statistics. The computational requirements of problems and individual tasks are unknown a priori. A method is presented for estimating these properties and requirements, using a k-NN [17] combined with a smoothed average.

2.1 Estimating properties

Many scheduling algorithms, other than the most trivial, utilize knowledge of the available system resources and the tasks to be processed when deciding to allocate a task to a processor [7, 21, 23, 61, 92, 96, 98, 103]. How to best

generate this knowledge is an open problem [98]. In general cases, all information used when deciding to allocate tasks to processors must be estimated. This of course is error-prone, with the errors in these estimations introducing inefficiency. The most common forms of estimation include benchmarking a task or set of tasks offline in advance [7, 21, 92, 96, 98], or requiring a directed acyclic graph with task, communication information and precedence constraints in advance [1, 7, 10, 16, 21, 24, 38, 55, 56, 57, 92, 96, 98, 102, 106]. The heterogeneous and non-dedicated nature of the resources in a loosely-coupled distributed system means that these types of estimates can be detrimental to accuracy and contribute to large margins of error, between the actual execution time and the estimated execution time of a set of tasks.

Taking a simple average of past task execution times and using it to predict future task execution times is error prone when presented with a heterogeneous set of tasks and processors. An average of past task execution times can only properly model a uni-modal distribution or a close-to-homogeneous set of task execution times. Neural networks and support vector machines can be trained to model complicated task execution time distributions, but generally require a large set of previously observed data and training [11]. The k-NN algorithm can model complicated task execution time distributions, and does not require training, although it does require more time to generate a result [42]. The advantage of k-NN is that it can easily adapt to sparse or dense regions in the distribution.

Assuming that each previously executed task has a finite running time; past task execution times can be used to predict future execution times [94].

One common technique is to use an expected time to compute (ETC) matrix, where the expected execution time of each task on each processor is contained in a row, gives the estimated execution time in seconds to compute a particular task on each processor. The matrix is populated dynamically as needed. There are many techniques for generating the ETC matrix, ranging from a simple average of past execution times to more complicated methods such as model-based methods [29], neural networks, support vector machines, and k-nearest neighbours (k-NN) [42]. The performance of each of these techniques degrades, with various degrees of grace, as the statistics of past execution times becomes more uniform and less stationary.

The scheduling problem we address can be stated as follows: we wish to schedule a number of problems, where each problem contains a number of indivisible tasks. The tasks contained within a problem can have different heterogeneous processing requirements (time, memory). The scheduler is required to map these tasks to processors, which can have different heterogeneous processing speeds, memory, and interconnection properties, for processing. The computational requirements of problems and individual tasks are unknown a priori. Problems arrive dynamically for scheduling. The properties and availability of the processors can vary randomly over time, with unknown statistics.

Our distributed computing system consists of a server processor (that runs the scheduler) and a collection of processors connected by a communications link. For the remainder of the chapter, in our terminology, a problem is defined as a pair of algorithms that is required to be run: a `task manager`

algorithm and a `task algorithm`. The `task manager` runs on the server. The `task algorithm` is sent to each processor. A task is defined as a set of parameters for the `task algorithm`, where task i is characterized by the tuple of parameters $X_i = (x_1^{\mathrm{i}}, x_2^{\mathrm{i}}, .., x_q^{\mathrm{i}})$ and q is the number of parameters. The restriction $X_i \in \mathbb{Z}^q$ is placed on the user to allow for the parameters to be mapped to coordinates in q-dimensional space. This coding is not seen as restrictive. A string parameter could be coded as an index into a hard-coded look-up table in the `task algorithm`, for example. As the degenerate option, a parameter can be represented by bit strings cast to integers.

The `task manager` generates tasks and puts them on the schedulers queue. If all tasks can be executed independently, the `task manager` puts them all on the queue at once. If the `task manager` requires a staged computation (for example, if there are dependencies between tasks) then the `task manager` will put tasks on the queue over time as the results of previous tasks become available. The `task manager` switches between different functionality in the `task algorithm` for different stages in the computation through the parameter list.

The actual processing time t_i of task i can be expressed as

$$t_i = ETC(X_i, j) + \epsilon_j^i, \tag{2.1}$$

where ETC(X_i, j) is the part of the execution time, in seconds, estimated with input vector X_i, j is the processor that task i was processed on, and ϵ is the error of the estimation (in seconds). It is assumed that the previous n task execution times on each processor j, where they exist, are stored

along with the input variables in a set U, as a single observation. A set of observations is denoted by O and defined as

$$O = \bigcup_{j} (U_{i=1}^{n}(t_i, c_i, X_i)^j), \tag{2.2}$$

where c_i is communications overhead. A separate O is maintained for each problem and it grows as more execution times become available, up to a maximum size of n, after which the oldest observations are removed.

Two methods are used in this thesis to generate estimated task execution times, a k-NN [17] and a smoothed average combined with analytical benchmarking. Each time a task is returned the following steps are performed. On receiving results for task i and the computational benchmark results P_j from processor j:

1. Pass results to task manager.
2. Calculate t_i based on recorded start time for task i.
3. Add (t_i, c_i, X_i) to U_j. (Remove oldest observation if $|U_j| > n$.)
4. Incorporate $t_i P_j$ into smoothed task processing requirement for the problem.

These steps will be explained in the following two subsections.

2.1.1 k-nearest neighbours

The estimated task execution times are calculated using selected observations from the set O Eq. (2.2), in Alg. 2.1. To decide which observations to include,

and their weighting, we use the k-NN algorithm. The k nearest observations, based on Euclidean distance in the space defined by vector X, out of the set of n observations are selected. In general k should grow in proportion to n such that both $k \longrightarrow \infty$ and $\frac{k}{n} \longrightarrow 0$ as $n \longrightarrow \infty$ [20]. We use $k = \lceil n^{4/5} \rceil$ which is shown to perform well in [42], which balances the computation time to the number of data points used. For example, if $n = 100$ then $k = 40$, so 40% of the observations are selected, whilst if $n = 10000$ then $k = 1585$, so 15% of the observations are selected for use in generating an estimated task execution time. L-smoothing [35] is used to make the algorithm more robust to outliers. A fixed percentage L of the largest and smallest values of t_i from the set of k previously selected observations are deselected, which gives

$$y = k - 2\lfloor Lk \rfloor \tag{2.3}$$

observations. Fig. 2.1 illustrates the effect of the k-NN and L-smoothing algorithms when selecting observations.

Given the input parameters of the next task i to be processed X_i, the set $U_j \in O$ of n previous observations for that problem on processor j, the number k of nearest observations to select, and a percentage L for L-smoothing, an estimated execution time ETC$(\cdot)$ for this task on processor j can be calculated (see Alg. 2.1 for pseudocode of the algorithm).

The algorithm is explained as follows. First we must select the observations which will be used to generate ETC$(\cdot)$. The k smallest elements of U_j are selected. L percent of the largest and smallest values of t_a from the set of k previously selected observations, are deselected. The set of y (see Eq. (2.3))

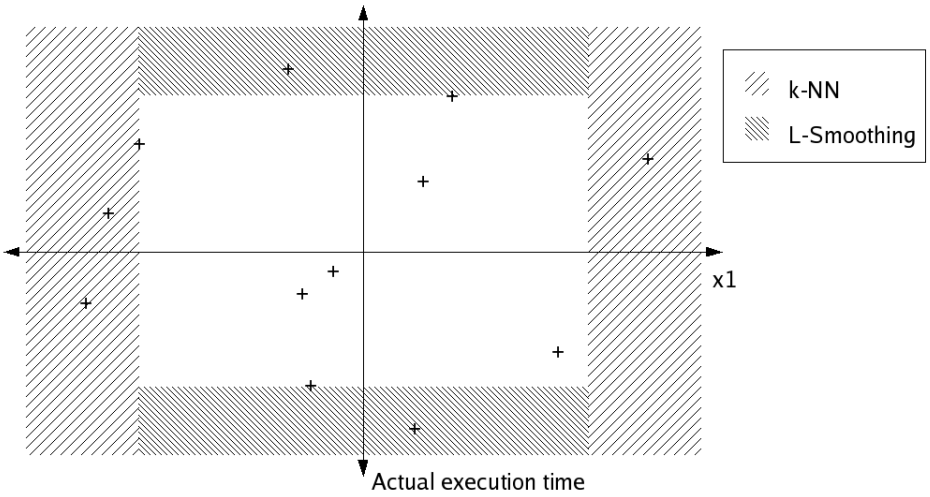

Figure 2.1: Illustration showing observations removed from consideration by k-NN and L-smoothing (shaded regions). The origin of the horizontal axis represents the parameter vector whose execution time is to be estimated. Each cross represents an observation, with the observations in the non-shaded region being used to generate an estimated execution time. In this example only 1 parameter (x_1) is used for each observation.

Input: $U_j \in O$ - Set of n past observations on processor j
X_i - Set of input parameters to task i
k - Number of nearest neighbours to select
L - Percentage of observations to deselect
Output: ETC
foreach *observation* $(t_a, X_a) \in U_j$ **do**
    Calculate Euclidean distance d_a between X_i and X_a (see Eq. (2.4));
Sort observations by distance;
Select k observations with smallest distances;
Deselect $L\%$ of observations with largest and smallest t_a;
foreach *Selected observation* **do**
    Calculate its influence in generating the estimated time ETC (see Eq. (2.5));
Calculate estimated time ETC from Eq. (2.6) ;
Return ETC;

Algorithm 2.1: Algorithm to estimate the execution time of task i on processor j. Individual steps are explained in the text and also shown in Fig. 2.1

remaining selected observations is called U'_j. The Euclidean distance d, from the input parameters X_i and X_a, is defined as

$$d(X_i, X_a) = \sqrt{\sum_{f=1}^{q} (x_f^i - X_f^i)^2}, \tag{2.4}$$

where q is the number of parameters in X. U_j is then sorted by distance. For notation reasons, let us order the set of parameters $\{X : (t, X) \in U'_j\}$ arbitrarily as $\{X_1^j, X_2^j, \ldots, X_y^j\}$. We define a weighting for each X_a^j that determines its influence on the estimated execution time of task i as

$$w_a^j(X_i) = \begin{cases} 1 & : \quad d(X_i, X_a^j) = 0 \\ \frac{\sum_{b=1}^{y} d(X_i, X_b^j)}{d(X_i, X_a^j)} & : \quad otherwise. \end{cases} \tag{2.5}$$

The set of running times $\{t : (t, X) \in U'_j\}$ in identical order is expressed $\{t_1^j, t_2^j \ldots, t_y^j\}$. Then the estimated execution time for task i on processor j is defined

$$\mathrm{ETC}(X_i, j) = \sum_{a=1}^{y} t_a^j w_a^j(X_i). \tag{2.6}$$

2.1.2 Smoothed average

If $U_j = \emptyset$ for a particular processor j an alternative estimation technique must be employed because the k-NN algorithm requires a minimum of one observation to generate an estimate. In such cases, a benchmarking metric is used to produce an estimate, without considering the input parameters, but by considering the other observations in O (for other processors). Benchmarks such as Linpack [25] and HPCC [39] can provide quite accurate information about system resources, in the context of particular types of computation. Linpack is used in this thesis to measure the execution rate of each processor in millions of floating point operations per second (MFLOP/s) [25]. This is a recognized standard used to benchmark systems for inclusion in the list of Top 500 Supercomputers [99].

The smoothed average algorithm makes use of each task execution time in each subset of O and which processor it relates to. Rather than estimating the task execution time, it estimates the computational requirement of the task, in MFLOP. The Linpack benchmark [25] is run periodically by each processor in the system which is used to calculate an approximate computation rate P_j of processor j in MFLOP/s. This benchmark result is sent to the server by each processor when requesting a task and when returning a processed

task. The server calculates a representative value $P_j = \Gamma^{\mathrm{Pj}}$, using Eq. (2.7), that uses the b benchmark results received from processor j up to that point. An approximate computational requirement, in MFLOP, for task i is then calculated from $t_i P_j$. This value is calculated for each returned task i and then incorporated into a single smoothed average task processing requirement $T_i = \Gamma^{\mathrm{tP}}$ for the problem using a smoothing function which will be described next.

For simplicity, the smoothing function strategy assumes that the gross features of the function to be smoothed will vary slowly over time. A smoothing function finds a single representative value for a sequence of values. As each new value is added to the sequence, this representative value is updated. For the first b values of a sequence of values $a_1, a_2, \ldots$, this representative value is denoted Γ_b^{a}, and defined recursively as

$$\Gamma_b^{\mathrm{a}} = \Gamma_{b-1}^{\mathrm{a}} + \nu(a_b - \Gamma_{b-1}^{\mathrm{a}}), \tag{2.7}$$

where the smoothness of the sequence of representative values is controlled by $\nu \in [0, 1]$, and where we let $\Gamma_0^{\mathrm{a}} = a_1$. The function allows one to vary the influence of more recent sequence values on the representative value, from no influence ($\nu = 0$) to complete dominance ($\nu = 1$). This method is less accurate than using k-NN, but can provide an estimate when less data are available.

Using the smoothed average method ETC is defined as

$$\mathrm{ETC}(i, j) = \frac{T_i}{P_j}, \tag{2.8}$$

where T_i is the most recent estimated computational requirement of task i in MFLOP and P_j is the most recent estimated execution rate of processor j in MFLOP/s.

If there are no observations at all for a particular problem (if each $U_j = \emptyset$), the scheduling mechanism defaults to round robin.

Next we will present an implementation of the task execution time estimation algorithm. It is tested on a real-world heterogeneous distributed system with a number of problems from the fields of bioinformatics, biomedical engineering and cryptography.

2.2 Heterogeneous Distributed System

A general purpose programmable Java distributed system, which utilizes the free resources of a heterogeneous set of computers linked together by a network, has been developed Keane *et al.*. [46] and extended by Page *et al.* [74, 75] . The system has been successfully deployed on over 800 computers, which were distributed over a number of locations, and has been successfully used to process bioinformatics [47, 48, 49, 50, 79], biomedical engineering [78], and digital holography [76] applications. Live statistics can be found at http://distributed.cs.nuim.ie.

The distributed system consists of 3 JAR files, a client, a server and a remote interface (see Fig. 2.2). A problem can be created for the system simply by extending 2 classes. The `Algorithm` class is run on the client and specifies the actual computation to be performed. We use a one-to-one mapping between a processor and a client in this chapter. The `DataManager`

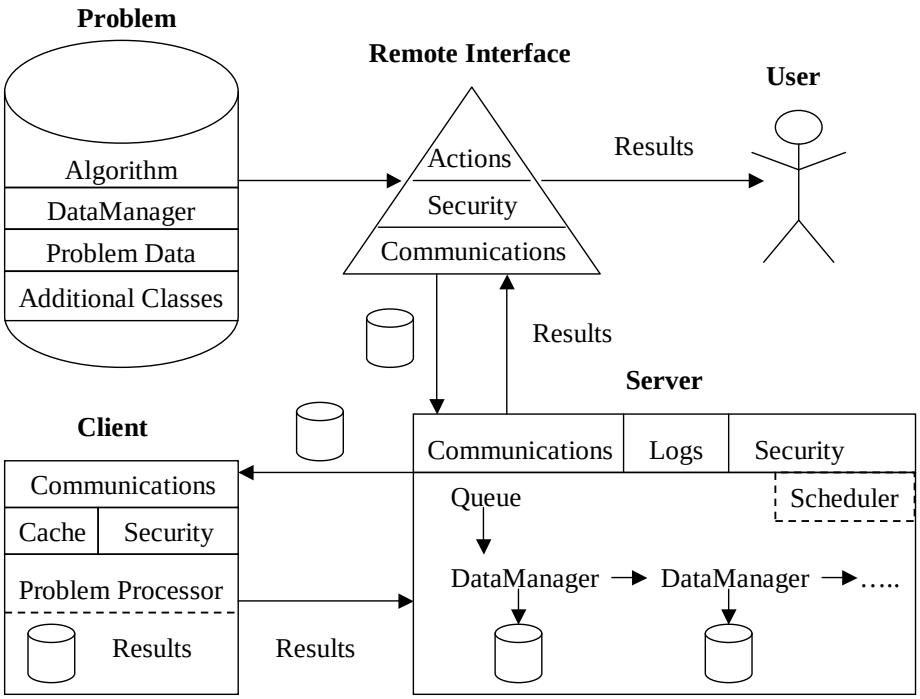

Figure 2.2: The major components of the Java Heterogeneous Distributed System.

class is run on the server and specifies how the problem is broken up into tasks and how the processed results are recombined.

The distributed system provides a simple scheduling interface, which allows the administrator of the system to select a scheduling algorithm using the remote interface. To create a new scheduler, a programmer only needs to extend the `SchedulerCommon` API and implement a single method called `generateSchedule`. This method simply takes in a list of tasks and maps them to processors. The system defaults to the simplest scheduler, round robin, although a number of more complicated scheduling mechanisms are available [75, 80, 82, 83].

2.3 Set of problems

A representative set of heterogeneous tasks for scheduling on a heterogeneous distributed system is an open problem [98]. To test Alg. 2.1 we have created a set of real-world problems from the fields of bioinformatics, biomedical engineering and cryptography. The problems are all easily parallelizable. Some problems are staged computations (DSEARCH and ElGamal) which requires all tasks from the current stage to be processed before processing can begin on the next stage, while the rest are have only a single stage. The mean processing-to-communication ratio (P-to-C ratio or CCR) is also different for each problem (see Table. 2.1), as is the amount of actual input and output data. Fig. 2.3 shows the P-to-C ratio after all problems have been processed. It forms a distribution with multiple uneven peaks, which is a non-trivial set of tasks to schedule. The processing time of the tasks (see Fig. 2.4) is heterogeneous, with large outliers, up to over 1000 seconds. Most of these problems can be efficiently executed individually on homogeneous distributed systems, thus we wish replicate this efficiency level when multiple problems are simultaneously scheduled on a heterogeneous set of processors. We will now look at each of the problems used to these the schedulers.

2.3.1 DSEARCH

The first problem is from the field of bioinformatics. Database searching for similar genomic sequences is one of the fundamental tasks in bioinformatics, but it is an NP-complete problem [9]. The DSEARCH application [79] performs a deterministic database search and significantly reduces the runtime

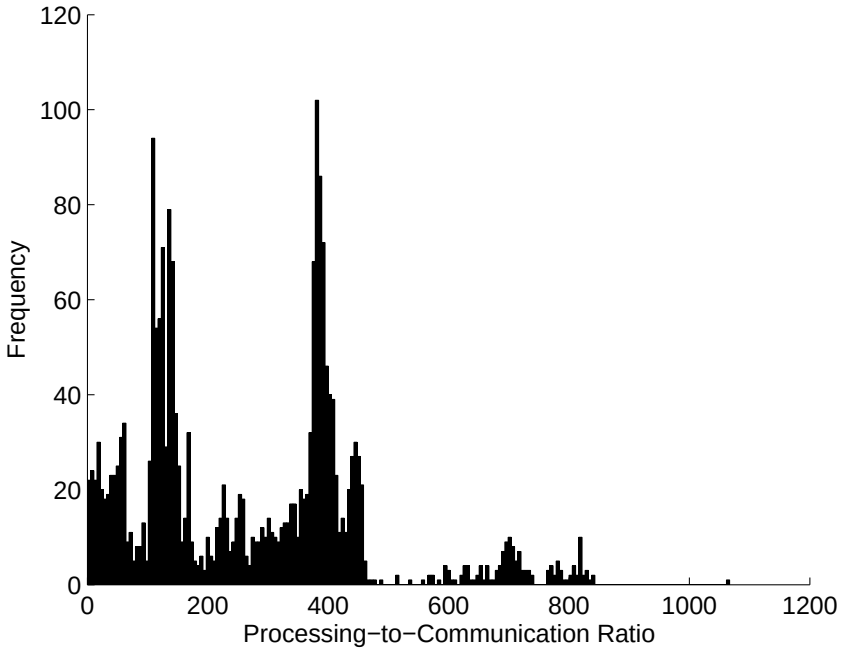

Figure 2.3: Histogram of the processing-to-communication (P-to-C) ratio of the tasks in the test set of problems. The number of tasks for each problem is given in Tab. 2.1.

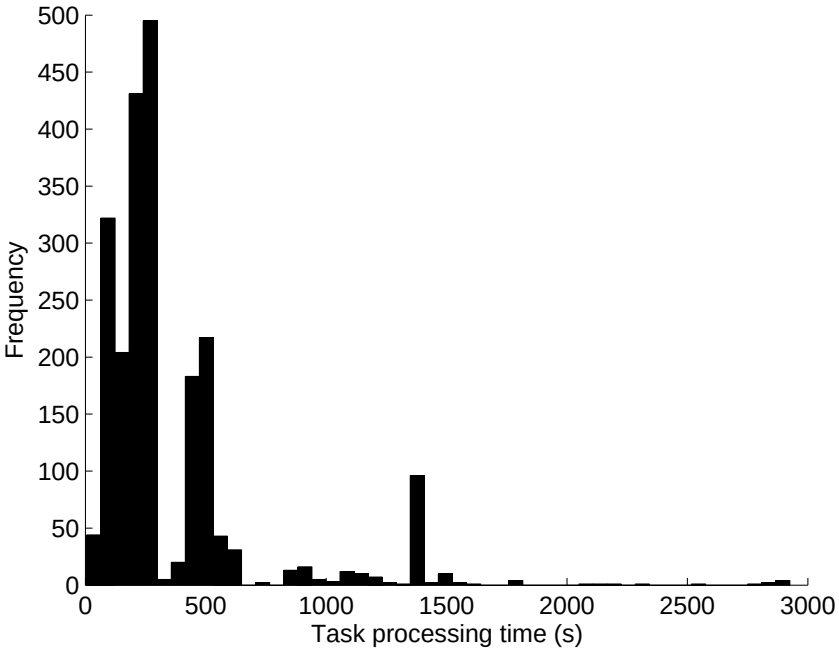

Figure 2.4: Histogram of the processing time of tasks in the set of problems

Problem	Avg communication time (s)	Avg processing time (s)	P-to-C	No. Tasks	Reference
SLTT	12.4	519.81	41.92	143	Chap. 2.3.4
DSEARCH	14.0	731.99	52.05	612	Chap. 2.3.1
MD5	14.4	235.52	16.36	800	Chap. 2.3.2
SHA1	64.5	543.02	8.42	900	Chap. 2.3.2
ElGamal	29.2	419.96	14.34	406	Chap. 2.3.2
TSP	9.5	353.72	37.04	121	Chap. 2.3.3

Table 2.1: Comparison of problem properties.

of large searches by using multiple processors. It requires the transmission of large amounts of genomic data when performing a search. Figure 2.5 shows how DSEARCH scales with up to 83 homogeneous of processors. Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm, and is the execution time of the sequential algorithm divided by the execution time of the parallel algorithm. Linear speedup indicates a 100% efficent parallelisation, with any sub-linear speedup indicating less than 100% efficiency, which is the normal case.

2.3.2 Cryptography

Three distributed cryptography applications have been developed. These are all very computationally intensive, each testing the strength of differ-

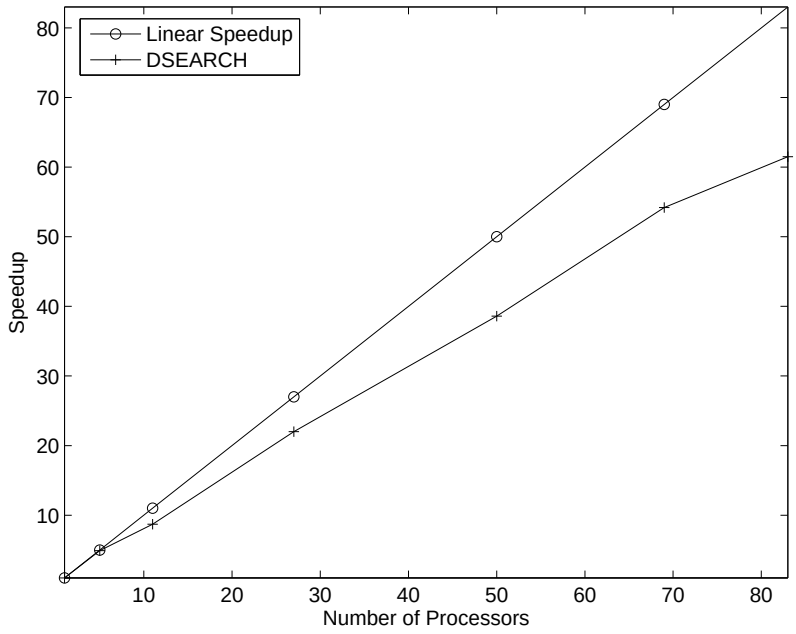

Figure 2.5: Speedup achieved by DSEARCH using up to 83 homogeneous Pentium III 1GHz processors.

ent cryptography algorithms. The first two test the strength of passwords hashed using the MD5 [65] and SHA1 [89] algorithms by using a brute force birthday attack [88]. A set of 2000 hashed passwords is sent to each client processor. Passwords are randomly generated, passed through the hashing function, and compared to the list of hashed passwords. Any matches are noted and returned to the server. Hashing passwords using these algorithms is a common practice in many applications, especially in on-line software. This method is however vulnerable to attack, due to the human desire for easy to remember passwords, which are quickly typed. This limits the total key space to such an extent, that it is computationally feasible to brute force

passwords.

The final cryptography application tests the strength of keys produced for the ElGamal [27] encryption scheme using the Pollard Rho method [85]. ElGamal is a public/private key encryption scheme based on the discrete log problem. We randomly generated public and private keys using the ElGamal encryption scheme. The public key was then distributed to 90 heterogeneous processors, where each performed a random walk [97], to attempt to find the private key. This was repeated 3345 times with a 64-bit ElGamal private keys. The time taken to find each private key was recorded and a frequency distribution was created, as can be seen in Fig 2.6. The time to break n-bit keys forms a Gaussian distribution.

2.3.3 Travelling Salesman Problem

A bruteforce distributed travelling salesman optimization application was created. It is a classic NP-hard problem. Given a graph of 14 cities, with weighted edges between cities, the goal was to perform a tour of all cities, with the shortest total path. Every possible permutation is checked by the application, to deterministically find the optimal tour. Each task is homogeneous, thus even the most trivial schedulers should be able to get near optimal speedup, when run on a homogeneous set of processing resources. The average speedup for this application using 86 homogeneous 600 MHz Pentium III processors was 95.6% as shown in Fig. 2.7. This application is has a high computation to communication ratio.

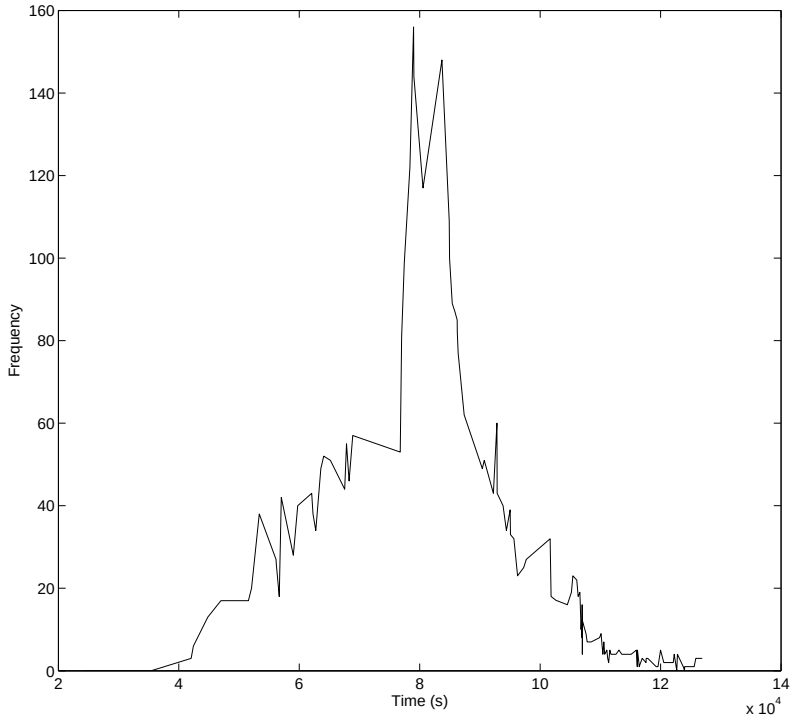

Figure 2.6: Histogram of time taken to break 3345 64-bit ElGamal private keys using 90 homogeneous Pentium III 600MHz processors.

2.3.4 Simulation of Light Transportation in Tissue

We have developed a distributed Monte Carlo simulation which models the propagation of light through tissue. It will allow for improved calibration of medical imaging devices for investigating tissue oxygenation in the white matter of the cerebral cortex. On a single processor these simulations would take an inordinate amount of time, limiting the potential usefulness of the model. To address this limitation we have developed a distributed application which allows for, in theory, an unbounded number of processors to perform

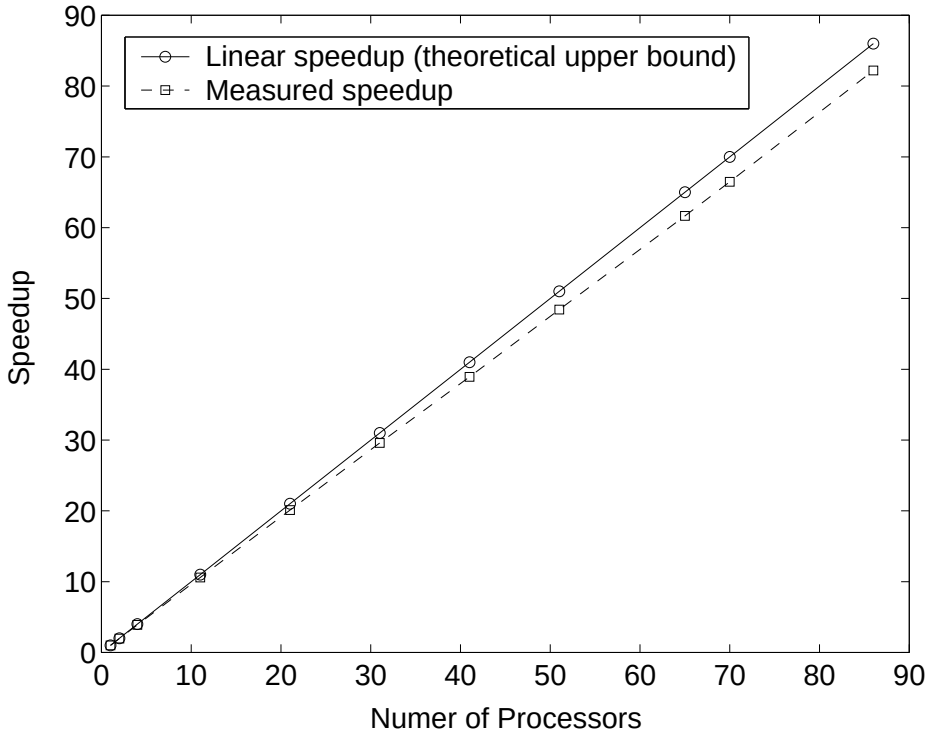

Figure 2.7: Speedup achieved for a bruteforce travelling salesman optimization application, using up to 86 homogeneous processors.

simulations in parallel. The application has been shown to achieve 97% efficiency when running on 60 homogeneous PCs (see Fig. 2.8).

A model of the different layers of tissue in and around the brain has been created. Fig. 2.9 shows the results of this simulation. We found that the source illumination footprint has an effect on the distribution of photons in the head and that lasers do produce a small, detectable beam in a highly scattering medium. The ability to model the statistics and distribution of the photon paths, which reach the white matter tissue of the human brain, allows for more accurate calibration of the imaging experiments, which previously

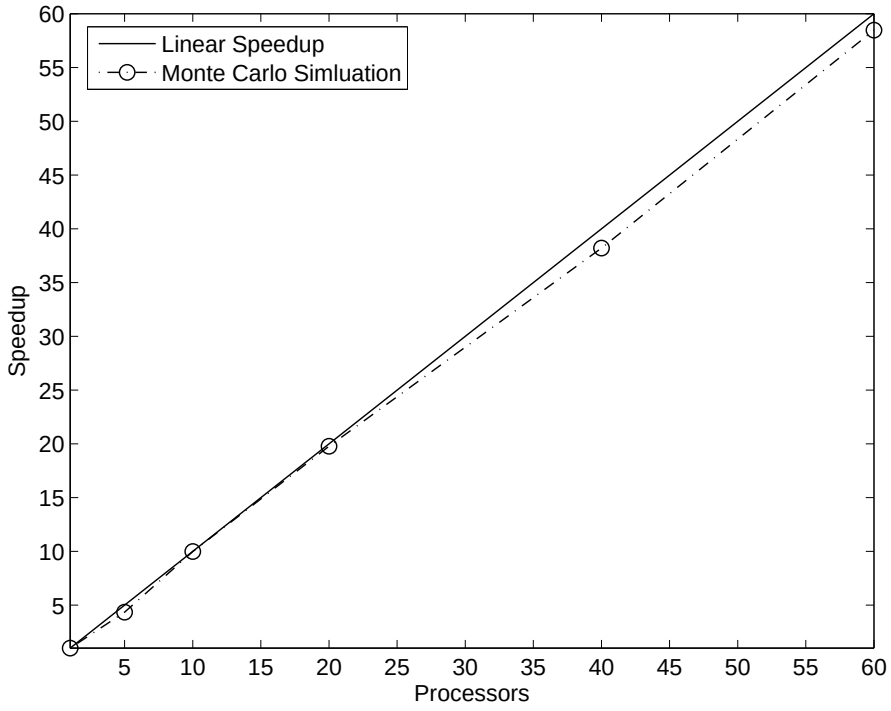

Figure 2.8: Speedup graph, with up to 60 homogeneous Pentium IVs with 512MB RAM, for the distributed Monte Carlo simulation.

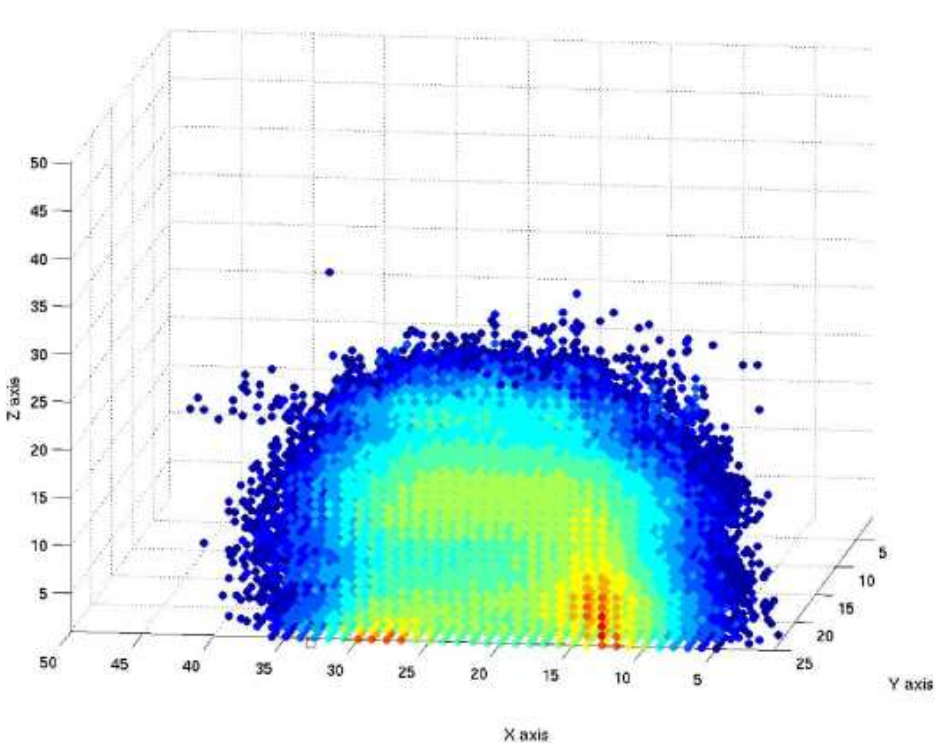

Figure 2.9: Simulated paths taken by photons with layers of human brain tissue. A near infra red source emits photons, which are detected by a senor 2cm away.

relied on trial and error. This research is part of a Brain Computer Interfacing (BCI) project. Further details about this application can be found in [78].

2.4 Experiments

We implemented and tested Alg. 2.1 on a real-world heterogeneous distributed system [46] with 90 PCs (Tab. 2.2). Two sets of processors where used, with 45 processors in each. The processors computational resources, measured in MFLOP/s, varied by up to 10%, due to slightly differing hard-

No. Proc	MFLOP/s	RAM (MB)	Bandwidth (Mb/s)	Processor
45	28-31	256	100	P3 600MHz
45	180-200	1024	100	P4 D820

Table 2.2: Client resources of a heterogeneous distributed system with 90 processors.

ware and software configurations. Machines with more RAM have the ability to store more intermediate data in memory, whereas an insufficient amount of RAM may lead to data being stored on the hard disk, which attracts a significant time penalty. All resources were non-dedicated, running Linux, and were connected by a 100 Mb/s network. The clients are connected to a dedicated server running Linux on a 3 GHz P4 with 1 GB of RAM. We used a single core on the Intel P4 D820 processors running a 32-bit version of Linux. This setup will be used for the experiments described in the next section.

2.4.1 Estimating system resources

Estimating the system resources and task execution times is difficult and error-prone. By accurately predicting the error in the estimation of these values, we can make better mapping decisions. We can also be more confident that the predicted makespan more accurately reflects the actual makespan.

The estimation of the systems processing resources is prone to error due to the dynamic nature of these non-dedicated resources. Fig. 2.10 shows that the predicted estimation error more closely tracks the actual estimation error of computational resources, as time progresses. The actual estimation error of processing resources is low overall. This is measured by periodically running

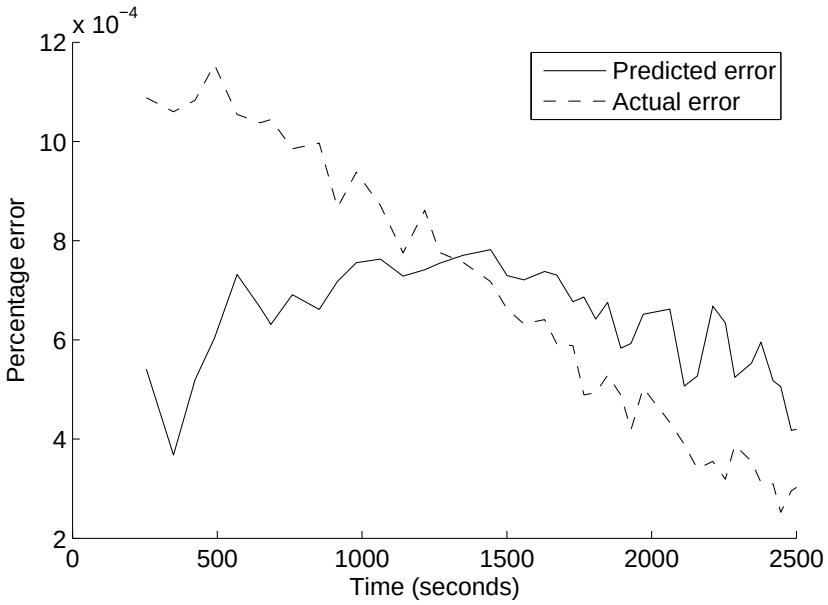

Figure 2.10: Predicted computational estimation error and actual computational estimation error over time

the Linpack benchmark on a 500 × 500 matrix. The processing resources used for these experiments do not vary greatly, because the processors are idle for most of the time. Thus the estimation error for the processing resources of the system never exceeds 1%. The estimation error for the communication times is also consistantly low, in the region of 1% due to the homogeneous nature of the communication resources used in this experimental setup, as detailed in Tab. 2.2.

2.4.2 Estimating task execution times

We look at the effect of estimating the task execution time using the smoothed estimate and a k-NN, and compare the results to the actual task execution

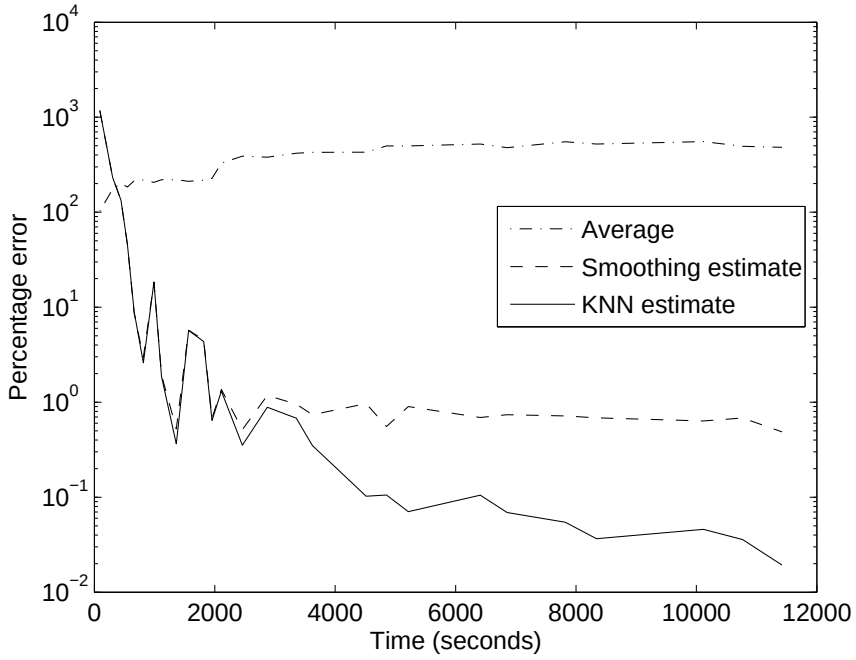

Figure 2.11: The absolute percentage error between the actual processing time of a task and the estimated processing time of a task over time, using a simple average of past task execution times, a smoothing estimate and a k-NN estimate. A log scale is used because the absolute values, between each plot, are quite large. The lowest point on the y-axis is $10^{-2}\%$.

times. Fig. 2.11 shows that as time progresses the error between the estimated execution time and actual execution time decreases for both the smoothed estimate and the k-NN estimate, but the k-NN estimate is approximately 10 times less error-prone that the smoothed estimate. As more observations are available to the k-NN algorithm, the error decreases, as can be seen in Fig. 2.12, which explains the continuous decrease in error from Fig. 2.11. Taking a simple average of all past task execution times to estimate future task execution times results in a high estimation error.

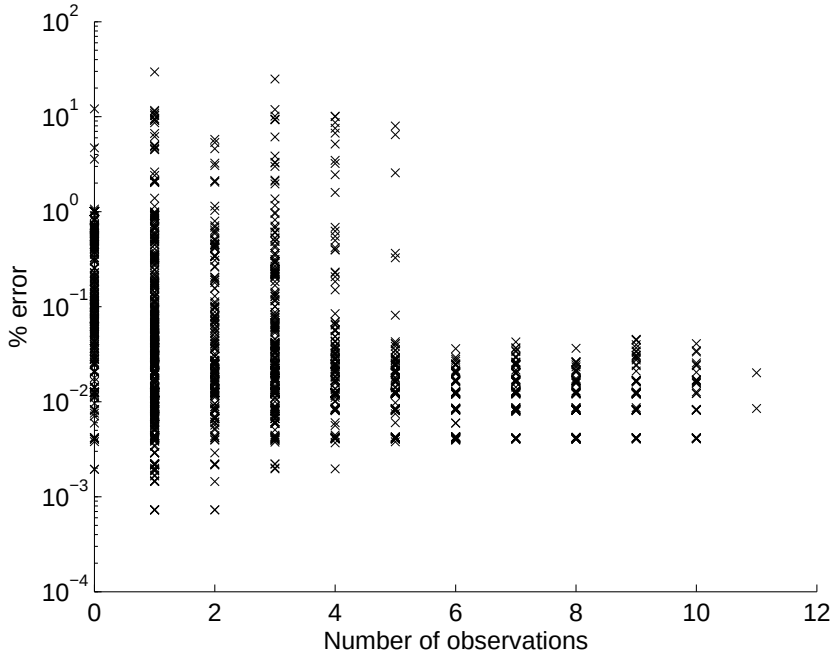

Figure 2.12: Absolute percentage error between estimated task execution time and k-NN estimate versus the number of observations used to generate the estimate. Each observation corresponds to a previously processed task.

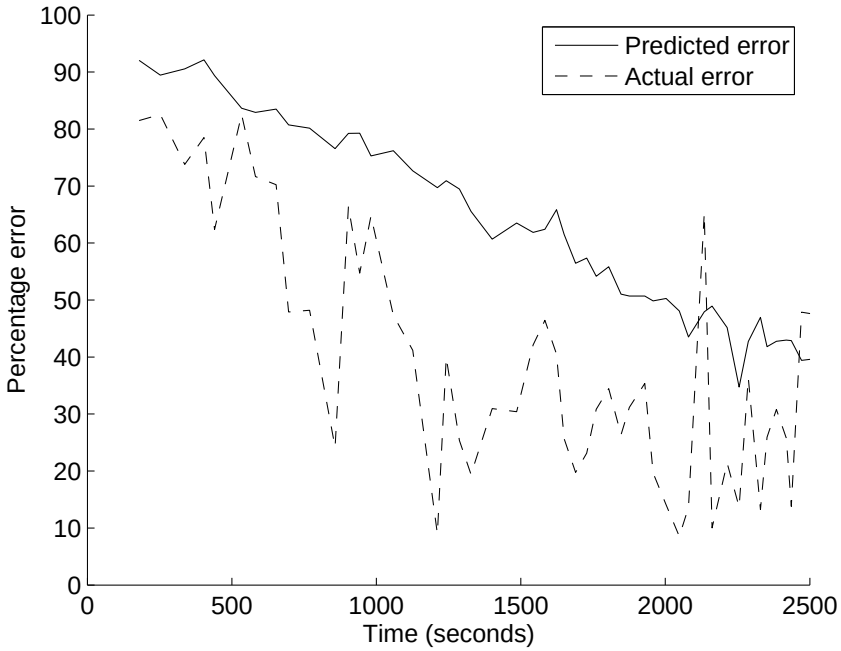

Figure 2.13: Predicted task estimation error and actual task estimation error over time, with absolute values shown.

In Fig. 2.13 the predicted and actual estimated task execution time error both approximately linearly decrease over time, with the prediction improving consistently over time. The error is still large, in the region of 50%, but this allows us to place a reasonable bound on the estimation error present in our estimations of the task execution times. This information aids the scheduling algorithm, providing an average upper bounds on the accuracy of the estimated task execution times.

An estimated communication to computation ratio (CCR) error can be generated for a task, once an estimation error is available for the task execution time and the processing resources. Fig. 2.14 shows that the average

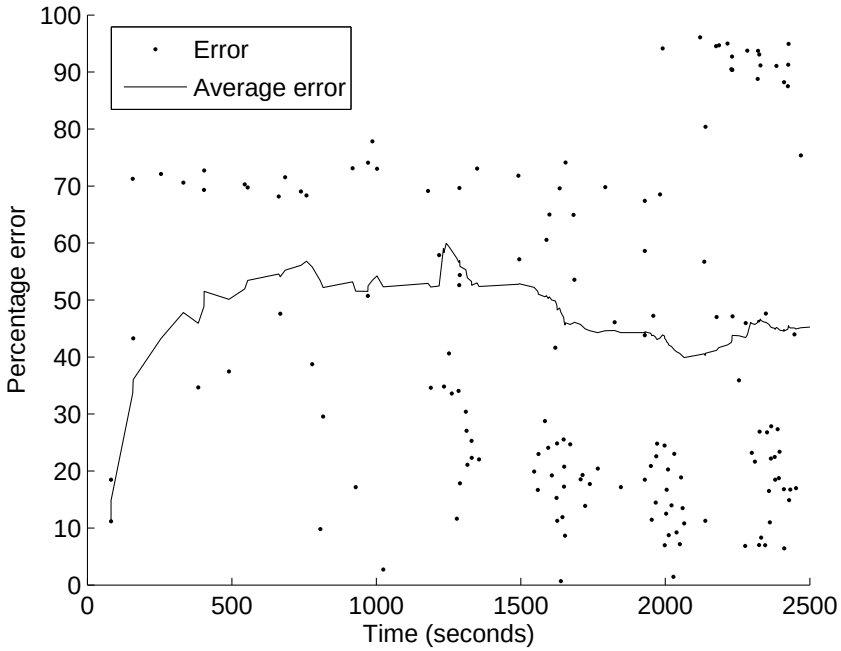

Figure 2.14: Absolute error between estimated CCR and the actual CCR of the tasks over time

estimated error of the CCR reduces over time.

2.5 Conclusion

In this chapter we have shown that non-linear task execution time distributions can be modelled using a k-NN algorithm combined with analytical benchmarking. The algorithm has been tested in a real-world, non-dedicated heterogeneous distributed system, with a diverse set of real-world problems from the fields of bioinformatics, biomedical engineering and cryptography. The estimates from the algorithm improve over time. This gives more accu-

rate information to use with a scheduling algorithm.

In the next 2 chapters we present 2 new scheduling heuristics which can adapt to the dynamic nature of a real-world heterogeneous distributed system, building on the work presented in this chapter. When there is an unknown amount of dynamism, a task allocation problem is not contained in NP (see Appendix B). In real-world heterogeneous distributed systems, the amount of dynamism is unknown, thus we cannot linearly transform efficient NP solvers.

Chapter 3

Task allocation using Genetic Algorithms

Parts of this chapter have also been published in the following articles [82, 83].

In this chapter we present a multi-heuristic evolutionary task allocation algorithm to dynamically map tasks to processors in a heterogeneous distributed system. It utilizes a genetic algorithm, combined with eight common heuristics, in an effort to minimize the total execution time. It operates on batches of unmapped tasks and can pre-emptively remap tasks to processors. The algorithm has been implemented on a Java distributed system and evaluated with a set of six problems from the areas of bioinformatics, biomedical engineering, computer science and cryptography. Experiments using up to 150 heterogeneous processors show that the algorithm achieves better efficiency than other state-of-the-art heuristic algorithms.

3.1 Introduction

Many heuristic algorithms exist for the task allocation problem, but most are limited to specific cases [45]. The use of evolutionary algorithms in scheduling, that apply evolutionary strategies from nature, allows for the fast exploration of the search space of possible schedules. This allows for good solutions to be found quickly and for the scheduler to be applied to more general problems. The genetic algorithm (GA) [37] evolutionary strategy has been shown to consistently generate more efficient solutions than other evolutionary strategies when applied to scheduling in heterogeneous distributed systems [10].

Many researchers have investigated the use of GAs to schedule tasks in homogeneous [38, 107] and heterogeneous [1, 10, 61, 98, 106] multi-processor systems with some success. However, the generality of these solutions are often reduced because of the assumptions made; i) calculating schedules off-line in advance [1, 10, 38, 98, 106], ii) a priori knowledge of communication times and task processing times [1, 10, 38, 98, 106], iii) instantaneous message passing [107], iv.) all processors are homogeneous [38, 107], and are dedicated to the distributed system [1, 38, 45, 90, 98, 105, 106, 107, 108]. All of these assumptions limit the applicability of a scheduler in a real-world distributed system. It is our belief that if a scheduler is to be made applicable to real-world distributed computing environments and problems, then it should not make any prior assumptions about resource homogeneity or availability.

In this chapter a scheduling strategy is presented that uses a GA to schedule a set of heterogeneous tasks on to a set of heterogeneous processors in an

effort to minimize the total execution time. It operates dynamically, allowing for tasks to arrive for processing continuously, and considers variable system resources, which has not been considered by other dynamic GA schedulers. To allow for efficient schedules to be produced quickly, the scheduler utilizes 8 heuristics, reducing the probability of processors becoming idle while waiting for a schedule to be generated. The scheduler has been implemented on a real-world distributed system and tested on 150 non-dedicated heterogeneous processors, with a variety of real-world problems from bioinformatics, biomedical engineering, computer science and cryptography.

3.2 Genetic Algorithm

We have created an algorithm which can adapt to varying resource environments utilizing a multi-heuristic GA (see Alg. 3.1), based on the homogeneous dynamic load-balancing algorithm in [107]. We wish to schedule an unknown number of tasks for processing on a distributed system with a minimal total execution time, otherwise known as makespan.

```
   Input: Set of tasks and processors
   Output: Mapping of tasks to processors
 1 foreach heuristic do
 2 |   generate schedule ;
 3
 4 while population not full do
 5 |   copy and mutate heuristic schedules ;
 6
 7 repeat
 8 |   cycle crossover ;
 9 |   random mutations ;
10 |   rebalance ;
11 |   roulette wheel selection ;
12 |   save best schedule (elitism) ;
13 |   update mutation rate ;
14 until stopping conditions met ;
15 return schedule with shortest makespan
```

Algorithm 3.1: Pseudocode for genetic algorithm. We refer to this algorithm as PN in the text.

The set of processors of the distributed system is heterogeneous. The available network resources between processors in the distributed system can vary over time. The availability of each processor can vary over time (processors are non-dedicated). Tasks are indivisible, independent of all other tasks, arrive randomly, and can be processed by any processor in the distributed

system.

When tasks arrive they are placed in a queue of unscheduled tasks. Batches of tasks from this queue are scheduled on processors during each invocation of the scheduler. The queue of unscheduled tasks can contain a large number of tasks. If all of these tasks where to be scheduled at once, the scheduler could take a long time to find an efficient schedule. To reduce the execution time of the scheduler and reduce the chance of processors becoming idle, we only consider a subset of the unscheduled tasks, which we call a batch. A larger batch will usually result in a more efficient schedule [107], but will incur a longer running time. To do this we dynamically set the batch size according to the estimated amount of time until the first processor becomes idle.

Each idle processor in the system requests a task to process from the scheduler, which it processes and returns. The scheduler contains a queue of future tasks for each processor, and when a request for work is received the task at the head of the corresponding queue is sent for processing. A processor does not contain a queue of tasks; because network resources are limited and processing resources are not dedicated. We also wish to avoid repeatedly issuing the same task multiple times, e.g., when a machine is switched off.

3.2.1 Encoding

Each schedule is encoded as a string of characters, using the same analogy as the encoding of DNA in nature. A single solution is referred to as a chromo-

5	6	2	-1	4	9	-1	1	3	7	-1	8

Figure 3.1: Encoding of a schedule within the GA, with -1 delimiting processors queues. Each number corresponds to a unique task ID, thus allowing for a mapping of tasks to processors.

some, and a set of multiple possible solutions is referred to as a population. Fig. 3.1 shows the encoding used within the GA. Each number represents a unique task identifier, with -1 being used to delimit different processor queues. This encoding allows for the execution order of tasks to be defined on each processor, which allows for precedence constraints (not covered in this research). If the execution order of tasks was not required to be defined, a simpler encoding can be used, where the index of each character corresponds to a task, and the character itself corresponds to a processor.

3.2.2 Fitness Function

A fitness function attaches a value to each chromosome in the population, which indicates the quality of the schedule. It comes from the evolutionary principle of 'survival of the fittest', where the organisms with the best characteristics for their environment have a better chance of surviving to the next generation than weaker organisms, which are less adapted to their environment. We use a localized makespan to delineate fitness. Simply taking the makespan of a solution only considers the total execution time, however a well balanced load distribution is also a desirable property, which will also lead to a lower makespan. Thus we have developed a fitness function which utilizes both. The localized makespan looks at when each processor will become idle

next, and adds on the time to process each task in the proposed schedule. The processors with the largest and smallest processing times are then identified. If these times are the same, it indicates a perfectly balanced schedule. As the difference becomes greater, so does the load imbalance, which also effects the efficiency of the resource utilization. The localized makespan of the y-th batch of tasks is $L_x = \max_{j=1}^{m}(\sum_{i=1}^{n_y} A_i^j + B_i^j) - \min_{j=1}^{m}(\sum_{i=1}^{n_y} A_i^j + B_i^j)$ where n_y is all of the tasks, up to and including the y-th batch of tasks, A is the processing time of a task, B is the communication overhead of the task, and x is a schedule from the population. All of the other variables are previously defined in Chap. 2.1. The fitness value of chromosome x is

$$F_x = \begin{cases} 1 & : \quad L_x = 0 \\ 1/L_x & : \quad otherwise, \end{cases}$$

and $F_x = [0, 1]$. A larger value indicates a better or fitter schedule.

3.2.3 Multiple heuristics

We use eight simple heuristics to create an initial population within the GA scheduler. The remainder of the population is generated using random permutations of these heuristics. The use of multiple heuristics in our initial population provides the GA with reasonable starting solutions, compared to starting with a completely randomly generated initial population. By employing elitism, the GA will always produce a solution which is equal to, or better than, the best heuristic solution in the initial population, because the best/fittest solution is always brought forward to the next generation.

The eight heuristics operate on batches of tasks, and each is presented with the same set of tasks. They are also all presented with estimated task execution times, estimated communication overheads, and execution rates of the processors in MFLOP. We will now present each of these heuristics.

The max-min (MX) heuristic begins with a set of unmapped tasks. The execution time of each task on each processor is added to an ETC matrix. For each task, the processor which will compute it with the minimum amount of time is selected and added to a set. The task-processor mapping with the largest completion time in this set is selected. This task is then assigned to the processors queue, and removed from the set of unmapped tasks. This process is repeated until all tasks are mapped to a processor. The MX heuristic attempts to schedule the longest running tasks as early as possible, to processors which will process the tasks as fast as possible. Tasks with shorter execution times can then be mixed with the longer running tasks resulting in an overall move evenly balanced load across the processors and a better makespan. The complexity is $\Theta(N^2)$, where N is the number of unmapped tasks and M is the number of processors.

The min-min (MM) scheduler [40] is similar to the MX heuristic, except that after the set of minimum completion times is found, the task with the overall minimum completion time is assigned to the corresponding processor. MM increases the probability that more tasks will get to execute on their first preference processor than with MX [61].

The max lightest loaded (LLX) heuristic scheduler considers the existing load on processors and the estimated MFLOP of the tasks. The set of

unmapped tasks is sorted in descending order according to their estimated size. The task with the largest computational requirement (in MFLOP) is then assigned to the lightest loaded processor. This is repeated until all tasks have been mapped to processors. LLX does not consider the time a task will take to execute on a given processor. It instead aims to put large tasks on lightly loaded processors, and small tasks on heavily loaded processors. If the estimated processing time of tasks has a high error, this heuristic will still provide a reasonably distributed load compared to MX and MM.

The min lightest loaded (LLM) heuristic scheduler operates in the same way as LLX, except the computational requirements of the tasks are sorted in ascending order. It attempts to schedule the smallest tasks first to increase the throughput of tasks.

Each of the heuristics above, MX, MM, LLX, and LLM assume there is no network overhead for scheduling a task on a processor. Where the processing to communication (P-to-C) ratio is very high, the network overhead may be negligible, but when it is low, or when their is limited network resources, the communications overhead must be considered for scheduling a task on a processor.

A variant of each of the above heuristics, MXC, MMC, LLXC and LLMC estimates the communication cost of mapping tasks to processors. Communication costs are estimated using the k-NN algorithm as described in Chap. 2.1.

Each heuristic is suited to different situations. MX performs well when there are more large tasks than small tasks, with MM performing better in

the opposite situation [61]. LLX and LLM are ideal heuristics for the situation where the size of tasks to be processed is not known, or the estimated processing time has high error. The variations of all the heuristics, which estimate communication costs, allows for efficient schedules to be produced in systems with high communications costs, such as massively distributed systems.

3.2.4 Crossover

The evolutionary phase of the GA is governed by the cycle crossover method [72]. Two parent (A and B) strings are randomly selected from the population. Index x_1 is randomly chosen. A_{x_1} and B_{x_1} are marked as having been visited. The value contained in B_{x_1} is noted. This value is then searched for in A and the index of this value is denoted as x_2. A_{x_2} and B_{x_2} are then marked as having been visited, and the value in B_{x_2} is searched for in A. This continues until an index in A is visited twice. A cycle has now been found. All indices visited are then crossed over to produce 2 new child strings. This ensures that the child strings generated are valid, e.g. only 1 task may be scheduled to 1 processor at any time. Since both parents contain the exact same character, just in a different order, a cycle will always be found.

3.2.5 Mutation

Two types of mutation are employed by the GA, one randomly swaps elements of chromosomes in the population, and the other is a rebalancing heuristic. Random mutations are an essential part of a GA, perturbing the

population to allow for new areas in the solution space to be searched. Every generation a percentage of elements in the population is randomly mutated. If the improvement in the makespan has not improved after 10 generations, the mutation rate is increased. Once the makespan begins to improve again the mutation rate is reduced. This reduces the probability of the GA getting stuck in a local minimum.

The other mutation operation utilizes a rebalancing heuristic to reduce the makespan. It achieves this by attempting to more evenly distribute the load on processors, by swapping tasks from heavily loaded processors on to lightly loaded processors. It has an average case complexity of $\Theta(M + N)$, where M is the number of processors, and N is the number of tasks. The solution generated by the heuristic will be discarded if it is worse than the starting solution, thus ensuring that the heuristic will only have a positive effect on the makespan.

3.2.6 Selection

The selection technique is based on the roulette wheel method [38, 90, 107]. The probability of a string going forward to the next generation is represented as a proportional sized slot on the roulette wheel, with a range from 0 to 1. Random numbers from 0 to 1 are then generated. The string which corresponds to the randomly selected slot is brought forward to the next generation. Since fitter strings have larger slots, they are more likely to be brought forward to the next generation. This process continues until a sufficient number of strings are selected.

3.2.7 Stopping conditions

When the stopping conditions are met, the evolution of the population will halt. This is to prevent the GA from running forever. Since this scheduler is intended for use in an on-line distributed system, it must produce schedules in a reasonable amount of time. Thus we use two stopping conditions: 1.) there is an upper bound on the maximum number of generations, to guarantee evolution will halt and 2.) if the makespan of the best solution has not changed after a set number of generations, then the GA will stop.

The GA algorithm described in this chapter contains research which was done over an extended period of time. Thus there are 2 slightly different algorithms used, which corresponds to different paper submissions over time. The first GA algorithm is enhanced by a single list scheduling heuristic (EF). This is evaluated with simulated sets of tasks and simulated configurations of distributed systems. The GA algorithm was then extended to utilize multiple scheduling heuristics, and is evaluated with a real-world set of problems on a real distributed system. We will present both sets of results in the following sections.

3.3 GA simulations

We have performed simulations using a GA scheduler, with a single list scheduling heuristic, presented in this chapter with simulated task distributions. A single heuristic was used to generate the initial population with up to 50 heterogeneous processors, and up to 10,000 randomly generated

heterogeneous tasks. Each experiment was repeated 50 times and an average result was calculated for each point on the resulting graphs.

A number of different experiments have been performed to demonstrate the effectiveness of the scheduling algorithm with varying communicating costs. We compare our scheduler to six other schedulers, and evaluate the results using two different, but related metrics, makespan and efficiency. An alternative method of evaluating evolutionary schedulers is to generate all possible solutions in advance and compare them to the solutions produced by the evolutionary algorithm [70]. The computationally intensive nature of this method makes it infeasible for the problem described in this chapter.

Determining a representative set of heterogeneous computing task benchmarks is an open problem as noted by Theys *et al.* [98]. Thus the task sizes are randomly generated using, uniform, normal, and Poisson distributions. By using different random distributions, which are commonly found in the real world, we can demonstrate the flexibility of our scheduling algorithm. For these experiments we will vary the communication costs and the task sizes.

3.3.1 Other scheduling algorithms

The performance of the PN scheduler has been compared to the performance of a number of different schedulers. Theses schedulers are the most commonly used schedulers in distributed computing (see Table 3.1). The earliest first (EF) scheduler [58] is an immediate mode heuristic scheduler. It schedules tasks on the processor which will finish processing earliest. The light-

Type	Key	Name	Ref.
Immediate	RR	Round Robin	[58]
	EF	Earliest First	
	LL	Lightest Loaded	
Batch	MM	Min-Min	[40]
	MX	Max-Min	[40]
Evolutionary	SA	Simulated Annealing	[51]
	ZO	Genetic Algorithm	[107]
	TA	Tabu search	[30]
	PN	This chapter	Alg. 3.1

Table 3.1: Taxonomy of schedulers.

est loaded (LL) scheduler is also an immediate mode scheduler, scheduling tasks on the most lightly loaded processors, without regard for the processing time of the task. MX is a batch scheduler which attempts to schedule the largest tasks first, and MM is the opposite, scheduling the smallest tasks first. We compare PN to three other evolutionary schedulers. A simulated annealing (SA) [51] based scheduler was created using the open source library Jannealer [43]. A tabu search (TA) based scheduler was created using OpenTS [73]. A GA scheduler (ZO) developed by Zomaya *et al.* [107] is used for comparison purposes.

The scheduling algorithms are of varying complexity (see Table 3.7), from the least complex, round robin (RR), to the most complex evolutionary algorithms. These schedulers represent the most commonly used heuristics and the state-of-the-art evolutionary schedulers.

3.3.2 Setup

We simulated the performance of our scheduler against the performance of six other schedulers EF, LL, RR, MX, MM, and ZO for these experiments. All of the tasks arrived for scheduling at the beginning of the simulation.

We mapped up to 10,000 heterogeneous tasks on to 50 heterogeneous processors. For these experiments each processor was assumed to have a fixed execution rate, measured in MFLOP/s. The aim of these experiments is to show that predicting the communication costs in advance will improve the overall efficiency and reduce the makespan, compared to heuristics which adapt to communication costs after they have occurred. All schedulers were presented with the same set of tasks for scheduling and all schedulers have the same information available to them.

We have decided to use a population size of 20, which is known as a micro GA [12] and used in [33, 107, 108], which speeds up computation time without impacting greatly on the final result.

3.3.3 Rebalancing heuristic

Fig. 3.2 shows the average percentage decrease in makespan after each generation of the GA, with points taken after every generation (1000 points in total). Each point on the graph is an average of 50 simulations, when scheduling 10,000 tasks which are normally distributed with a mean of 1000 MFLOP and a batch size of 500. The tasks are scheduled onto 50 heterogeneous processors with processing rates of between 30 and 150 MFLOP/s Each simulation used a different set of tasks. The largest reductions in makespan

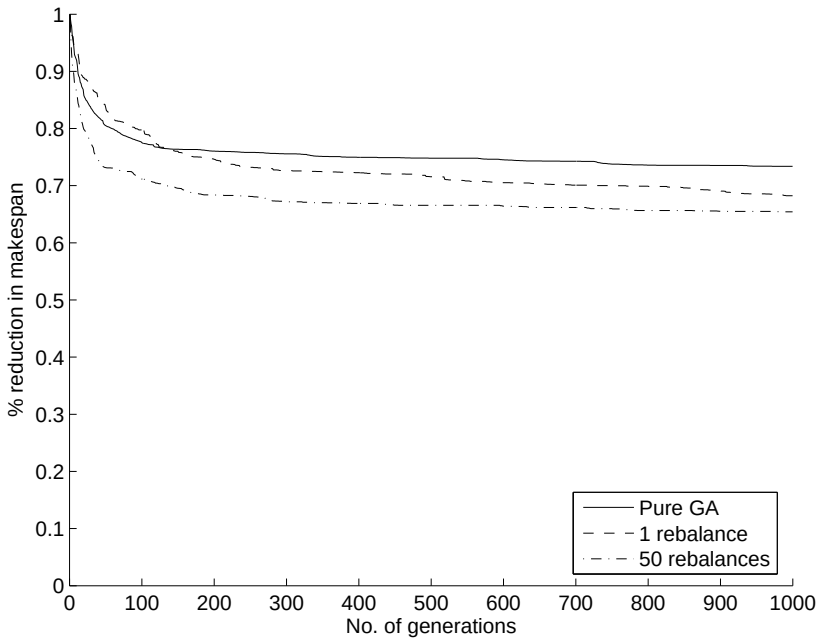

Figure 3.2: Average of 50 simulations of the reduction in makespan after each generation of the GA, where the initial makespan is 1. A set of 10,000 normally distributed tasks was used.

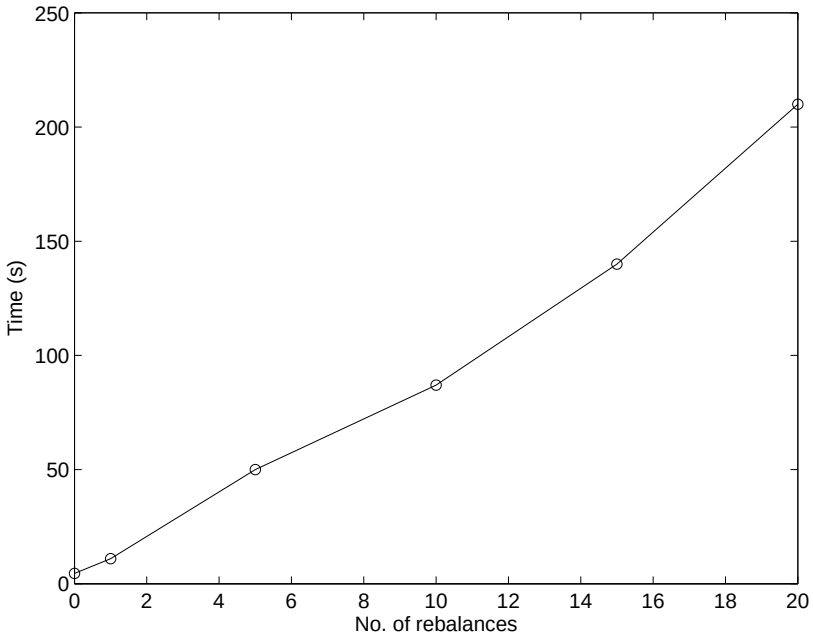

Figure 3.3: Time taken to schedule 10,000 tasks with varying numbers of rebalances in every generation of the GA.

occur in the first 100 generations, after that the reductions begin to level out, requiring larger numbers of generations, with little improvement. The rebalancing heuristic minimizes the makespan further than a pure GA, with 50 rebalances per chromosome in the population per generation, resulting in the makespan being reduced to only 65% of its original value after 1000 generations. A single rebalance reduces the makespan to approximately 70% whilst no rebalancing (pure GA) reduces to 75%.

These rebalances do have an associated additional cost in terms of time. Fig. 3.3 shows the time taken to run a GA for 1000 generations with varying numbers of rebalances, when run on a 1.4 MHz P4. Once again we randomly generate sets of tasks 10,000 tasks, normally distributed with a mean of 10,000 MFLOP and a batch size of 500. It increases the time taken by approximately a constant factor. We have decided to only perform a single rebalancing at each generation to enable the algorithm to run quickly, but this combination of heuristic and evolutionary techniques gives rise to a more efficient scheduler.

3.3.4 Normal distribution of tasks

Fig. 3.4 shows the efficiency of the seven different scheduling algorithms when the task sizes are normally distributed, with varying communication overheads. We used a batch size of 200 with 1000 tasks to be scheduled which were randomly generated at the beginning of each scheduling simulation with each point on the graph consisting of an average of 20 complete schedules. Fig. 3.4 consists of the efficiency of 2000 complete schedules with varying

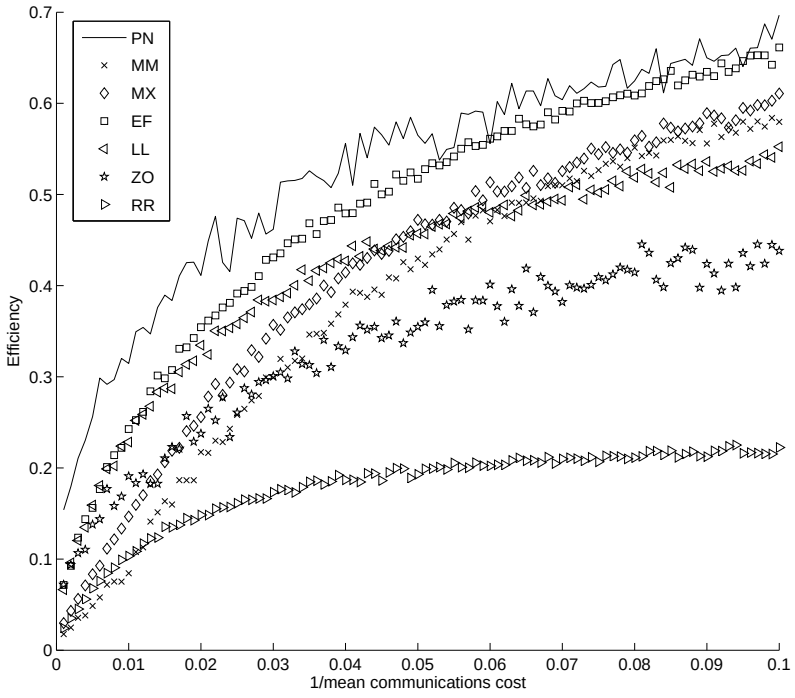

Figure 3.4: Efficiency of schedulers with a normal distribution of task sizes and varying communication costs, where PN is the scheduler presented in this chapter. An efficiency of 1 indicates 100% utilization of processing resources.

communications costs, and all other variables kept fixed. The task sizes were generated with a standard deviation of 1000 MFLOP and a variance of 9×10^5 MFLOP. The horizontal axis in Figs. 3.4 is the mean communication cost for all communication links between all clients and the scheduler. Its is an inverse of the mean communication cost for all links. Each communications link has its own randomly generated mean cost, which is normally distributed. Beginning with a very high communication overhead, all schedulers perform poorly, which is to be expect. As the communication overhead reduces, the ability of the different schedulers to efficiently manage the communication costs becomes apparent. The simple RR algorithm performs very poorly, because it makes no effort to balance the load on different processors, or to consider the communication links. The ZO algorithm is best suited to homogeneous sets of processors. The list scheduling heuristics, EF and LL, have the next best efficiencies, but since they only consider a single task at a time when scheduling, they cannot look far enough forward to better utilize the processing resources. The batch scheduling heuristics and PN perform the best overall. They can take a set of tasks, and have the versatility to choose schedule any task from that set, resulting in a more efficient schedule. Fig. 3.4 shows that our scheduler (PN) gives the best processor efficiency overall. It is best able to manage the communication overhead. Table 3.2 contains the makespan for the algorithm, with a varying batch size and shows that PN out performs all the other schedulers in terms of total execution time with a makespan of 4083 seconds.

	Poisson		Uniform		Normal
Schedulers	10	100	[10:100]	[10:10000]	1000
EF	6145	10993	2316	5607	4946
LL	7282	10820	2588	8194	6049
RR	9497	12773	2990	9760	9812
ZO	6152	9361	1869	6679	7189
PN	3784	6884	1565	4318	4083
MM	5275	7425	1576	5451	5102
MX	7123	7826	1795	4875	4906

Table 3.2: Makespan when task sizes have: 1.) a Poisson distribution with a mean of 10 and 100 MFLOP, 2.) a uniform distribution of [10:100] and [10:10000] and 3.) a normal distribution with a standard deviation of 1000 MFLOP and a variance of 9×10^5 MFLOP.

3.3.5 Uniform distribution of tasks

Fig. 3.5 shows the efficiency of the seven different schedulers with varying communication costs. The task sizes were uniformly distributed between 10 and 1000 MFLOP. The two meta-heuristic schedulers (PN and ZO) clearly provide more efficient schedules compared to the more simple heuristic schedulers. This occurs because the meta-heuristic schedulers have the ability to explore a wider search space than the other heuristic schedulers. We have also varied the range of task sizes noting the makespan in each case. When the task sizes vary from 10 to 100 MFLOP/s (as shown in Table 3.2) many of the schedulers provide similarly efficient schedules. This is because the ratio of the smallest to the largest task is only 1:10. As the set of tasks becomes more equal, the efficiency of most of the schedulers should improve. We see that the task sizes are distributed over a wider rang of 10 to 10000 MFLOP/s, the differences between the various schedulers become more accentuated, as

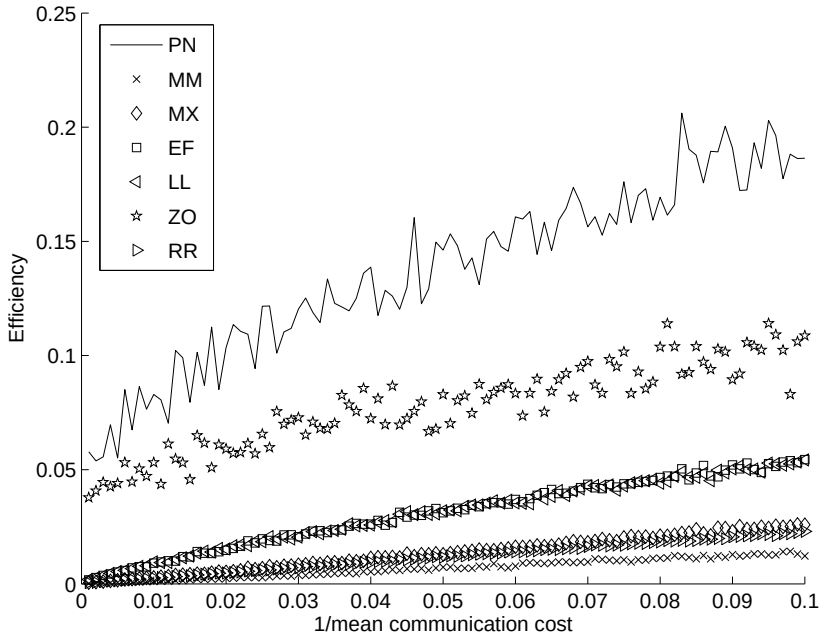

Figure 3.5: Efficiency of schedulers where task sizes are uniformly distributed between 10 and 1000 MFLOP, and varying communication costs, where PN is the scheduler presented in this chapter. An efficiency of 1 indicates 100% utilization of processing resources.

can be seen in Table 3.2.

3.3.6 Poisson distribution of tasks

We have randomly generated sets of tasks using a Poisson distribution and varied the mean. In Table 3.2 where the mean is 10 MFLOP, we can see that PN performs the best followed by MM, whilst MX performs quite badly, when the mean is small. When the mean is increased to 100 MFLOP (see Table 3.2) the batch schedulers all perform well, whilst the immediate mode schedulers

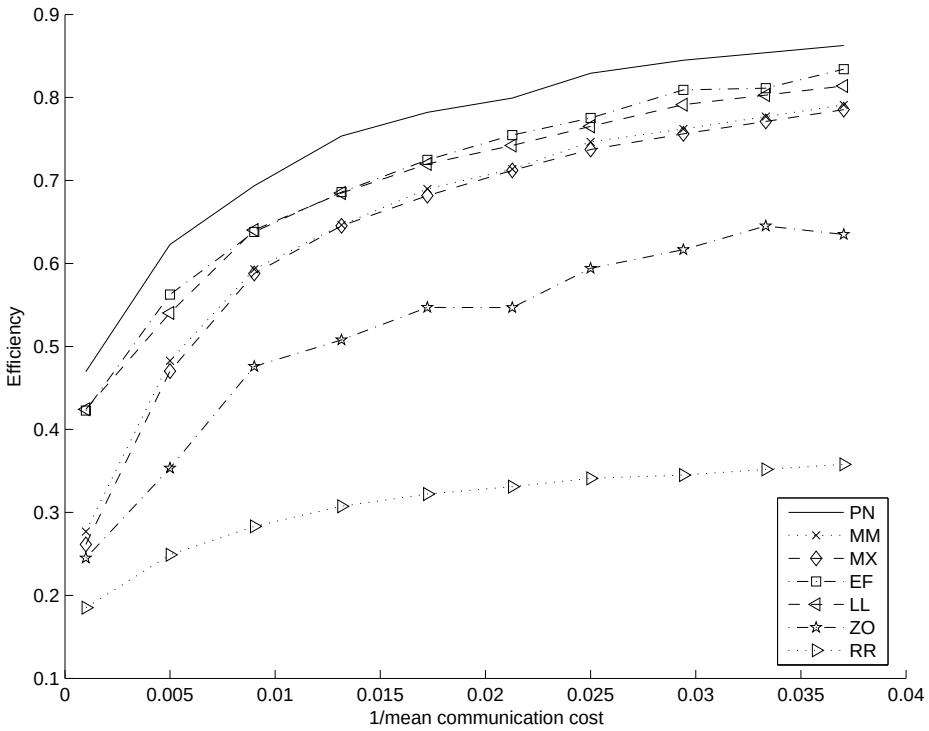

Figure 3.6: Efficiency of schedulers with a Poisson distribution of task sizes and varying communication costs, where PN is the scheduler presented in this chapter. An efficiency of 1 indicates 100% utilization of processing resources.

do not perform as well. Fig. 3.6 shows the efficiency of all of the schedulers. The PN scheduler most efficiently schedules tasks for different communication overheads, with EF, LL, MM, and MX performing reasonably efficiently. This is due to the more homogeneous nature of the task size distribution, in comparison to the other distributions, e.g. the Poisson distribution has a high initial peak and a long tail.

3.3.7 Analysis

The scheduler works very well with simulated sets of tasks with the 3 most common distributions. It consistently produces low makespans and high levels of efficiency, compared to other schedulers. The simplest scheduler RR, which is very commonly used in practice, performs the worst in all cases, because it makes no attempt to utilize any available information about the state of the system or the size of the tasks, when mapping tasks to processors. The batch schedulers perform reasonably well in some cases, due to their ability to choose from a set of of tasks when making a scheduling decision. The list scheduling heuristics have variable performance, which entirely depends on the distribution of the task sizes, but can perform well if in certain cases. The ability of the ZO algorithm to adapt to different task size distributions leads to a more consistent and predictable level of performance, and thus is more reliable for an unknown task size distribution than the simple heuristics.

Simulations do not contain the complexities found in real-world systems. Thus, the performance of schedulers in sterile simulations does not directly translate to real-world systems. We address this deficit by using a real-world distributed system for performance evaluation.

3.4 Experiments

For the experiments described in this section, we primarily used the 3 experimental setups in Table 3.3, run on a heterogeneous Java distributed sys-

	Heterogeneity	No. Proc	MFLOP/s	RAM (MB)	O/S	Processor Type
A	High	91	28-31	256	Linux	P3 600MHz
		50	190-229	512	Linux	P4 2.4GHz
		4	15	192	Linux	P2 266MHz
		1	154	1024	Windows	Centrino 1.4GHz
		1	25	512	Linux	P3 500 MHz
		1	37	256	Linux	P3 1GHz
		1	72	256	Linux	P4 1.7GHz
		1	91	1024	FreeBSD	AMD 2400+XP
B	Low	45	28-31	256	Linux	P3 600MHz
		45	180-200	1024	Linux	P4 D820
C	Homogeneous	45	180-200	1024	Linux	P4 D820

Table 3.3: Client resources of different experimental setups.

tem [46]. The first and simplest setup is a homogeneous set of processors, which we use as a base case for our experiments. This allows schedulers which favour a homogeneous set of processors to excel. The next setup is a set of processors with 2 homogeneous sets of processors. Both of theses setups used a 100 Mbps network. Finally, we used a set of processor with high heterogeneity and with a heterogeneous network which was spread over 3 different LANs and ranged from 10-100 Mbps. We had non-dedicated usage of these processors, and the actual available processing and network resources varied stochastically over time. All experiments were performed at off-peak times to minimize the effect of these variations. All the clients connected to a dedicated server running Linux (Fedora Core 4) on a 3 GHz P4 with 1 GB of RAM.

3.4.1 GA experiments

Parameters used within a GA, such as the number of generations, mutation rate and chromosome length, can effect the running time and quality of results generated by the GA. We will investigate the effect varying these can have on the scheduling algorithm.

The execution time of the scheduler increases approximately linearly with an increase in the number of chromosomes. This can be seen in Fig. 3.7, where we varied the chromosome length and measured the execution time of the GA. We fixed the number of generations at 500 and ignored all other stopping conditions. The execution time of a given chromosome length varies, due to the stochastic nature of overheads in a real-world distributed system, but the majority of times fall into a tight linear range. The tasks used in the experiment are described in Table 2.1. The scheduler produces schedules for large numbers of tasks and processors quickly, for example, the GA scheduler can schedule a batch of 170 tasks in under 1 second.

We then repeated the experiment allowing variable numbers of generations and a variable mutation rate, where the stopping conditions dictate the number of generations to be run. Fig. 3.8 shows that when we vary the chromosome length, the execution time does not increase linearly. The mean chromosome length is 98.42, with a fixed number of processors and a variable number of tasks, and the standard deviation is 9.6. The mean running time, is 18 seconds with a standard deviation of 11 seconds. The total execution time with a fixed mutation rate and fixed number of generations was 7208 seconds versus 7053 seconds for a variable mutation rate and variable

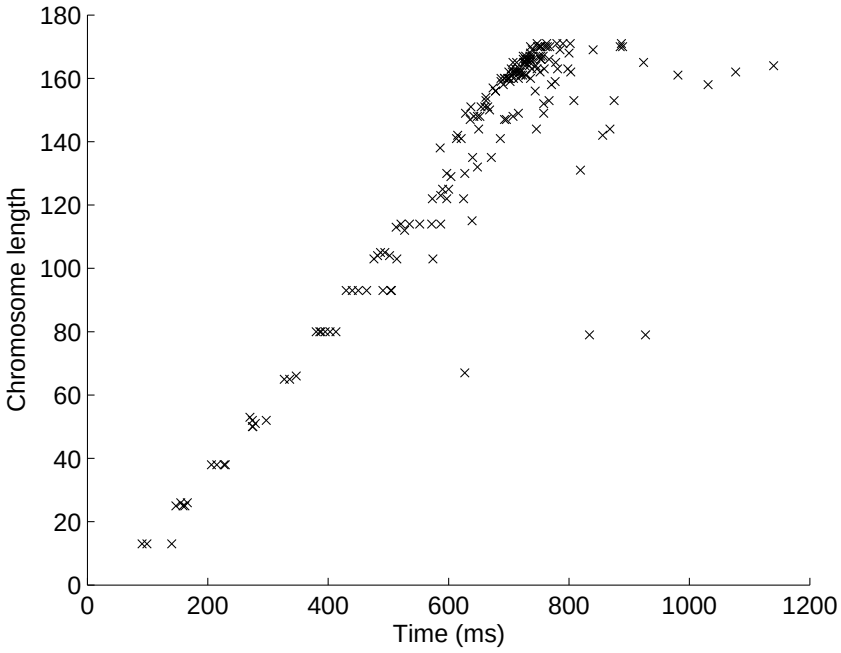

Figure 3.7: Execution time (ms) of PN scheduler with a fixed number of generations and a fixed mutation rate. The chromosome length corresponds to the number of tasks to be scheduled.

a dynamic number of generations.

However, the scheduler uses a variable number of generations, depending on whether the stopping conditions are met. If there in no improvement after 10 generations, the algorithm stops. The histogram in Fig. 3.9 shows the number of generations performed before this stopping condition halts evolution. It forms a Poisson distribution, which indicates that the scheduler finds either a local minimum or the global minimum makespan within a relatively low number of generations.

When the quality of the solution produced is considered, we found that

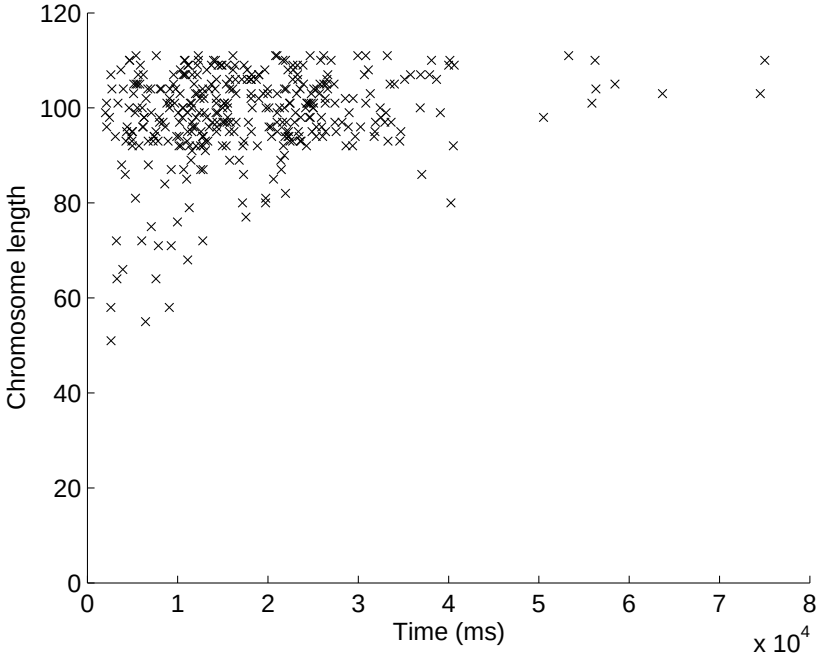

Figure 3.8: Execution time (ms) of PN scheduler with a dynamic number of generations and a variable mutation rate

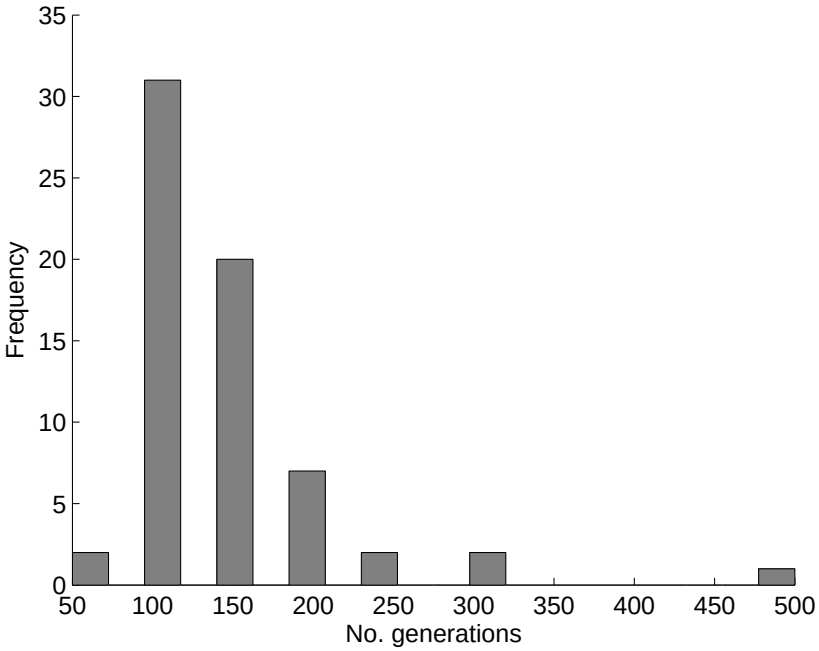

Figure 3.9: Number of generations run before stopping conditions terminate evolution of the GA

the greatest average reduction in makespan occurs within the first 200 generations. Fig. 3.10 shows this with a large reduction in makespan at the beginning, but the returns diminish quickly. Since the execution time of a generation is a constant factor, reducing the number of generations allows for a lower execution time of the scheduler. In a real-time system a client might be lying idle whilst waiting for a schedule to be produced, nullifying the effects of a more efficient schedule, thus a lower scheduler execution time is desirable.

We then looked at the effect the population size on the makespan achieved when scheduling on a real-world distributed system with 124 processors (see

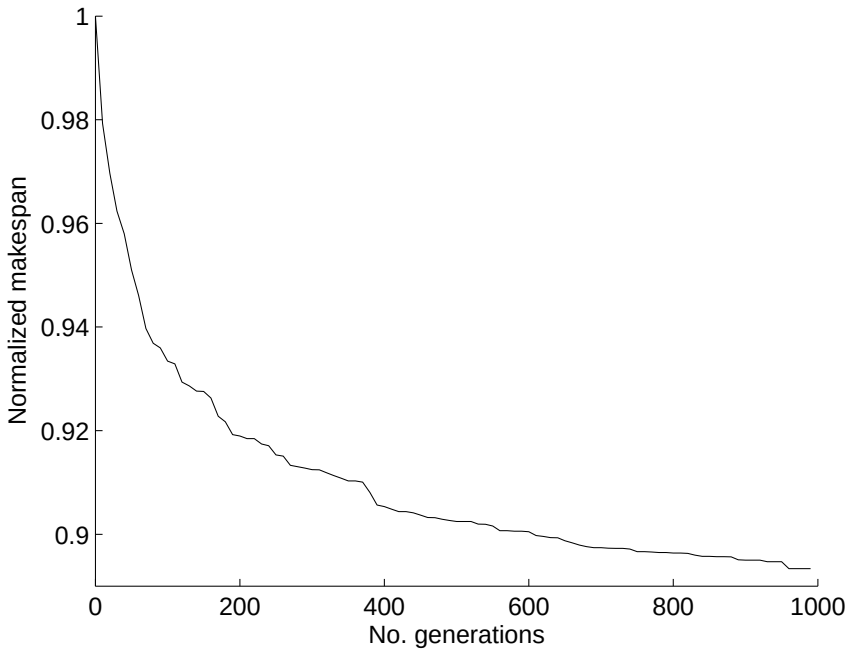

Figure 3.10: Average makespan achieved with varying numbers of generations in the GA scheduler

No. Processors	MFLOP/s	RAM (MB)	Network link (Mb/s)	Processor Type
47	180-200	1024	100	P4 D820
45	190-229	512	100	P4 2.4 GHz
32	28-31	256	10	P3 600 MHz

Table 3.4: Client resources used in the distributed system for the experiment shown in Table. 3.5. The operating system on all clients was Linux.

Table 3.4). Table 3.5 shows that when a larger population size is used, the effect on the overall makespan is negligible compared to using a small population size. This is due to the stopping condition which halts evolution if there is no improvement in makespan after 10 generations. The greater diversity in a large population allows for a minimum to be found in less generations, which offsets the longer execution time for a single generation. A smaller population requires more generations to achieve the same effect, however the execution time for each generation is less. The only difference between using a small and large population size is the spacial requirement. Thus to reduce the overall memory consumption of the algorithm we use a small population size (a micro GA [12]).

3.4.2 Multiple heuristics performance

We wish to show that using multiple heuristics to generate schedules for the initial population of the GA provides more efficient schedules than using each

Population size	Makespan (s)	Scheduling Time(s)	Mean Scheduling Time (s)	% Efficiency	% Communication Costs
10	4653	24.0	0.31	80.4	0.407
20	4720	23.1	0.30	78.1	0.405
30	4701	26.3	0.30	86.4	0.444
40	4672	22.6	0.28	86.9	0.436
50	4649	29.7	0.33	83.3	0.436
60	4846	32.1	0.34	84.3	0.437
100	4686	25.7	0.31	80.9	0.463
1000	4855	32.3	0.34	86.3	0.541
5000	4720	24.8	0.31	83.1	0.418
10000	4711	21.5	0.28	82.7	0.415

Table 3.5: Varying population size of the scheduling algorithm where the GA terminates if there is no improvement in makespan after 10 generations.

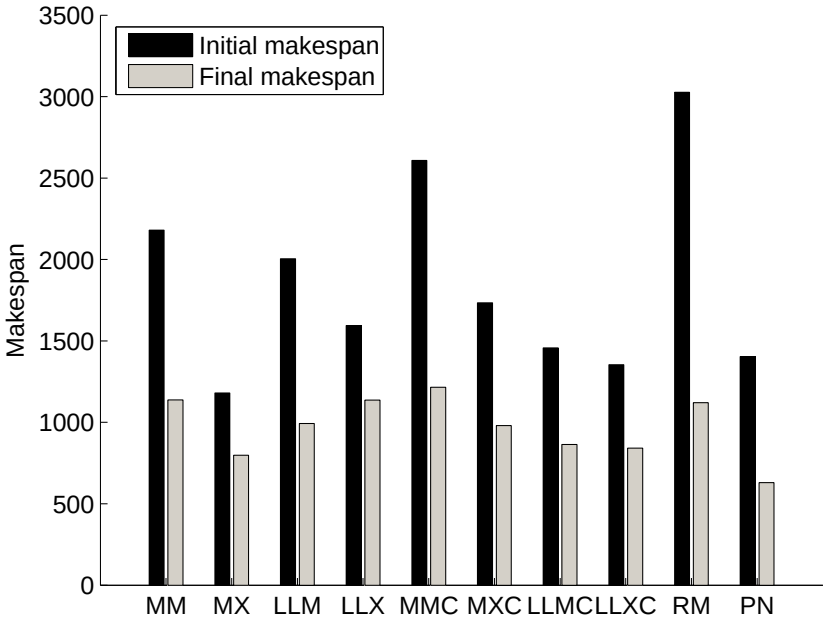

Figure 3.11: Performance of each heuristic when used on its own to initialize the GA

individual heuristic on its own, or using a purely random initial population. In Fig. 3.11 we use each heuristic individually to initialize the population of the GA. Each bar is an average 10 simulations, and we scheduled 600 tasks with normally distributed execution times on 30 heterogeneous processors.

The black bar shows the average initial makespan produced by the heuristic, and the gray bar corresponds to the average final makespan produced by the GA from that initial population. The population consists of only one heuristic and random variations of the schedule produced by the heuristic. A randomly chosen initial population (RM) presented for comparison purposes. The algorithm presented in this chapter (PN) utilizes all of the heuristics to

generate an initial population. The initial makespan for PN is an average of the best solutions generated by the heuristics. As can be seen in Fig. 3.11 using multiple heuristics provides, on average, a lower makespan.

In Fig. 3.12 we compared each heuristics initial solution to the final evolved solution (PN), with PN utilizing all of the heuristics. A set of 6 real-world problems (see Chapter. 2.3) where used for this experiment, processed by 25 non-dedicated heterogeneous processors (see Table 3.6). Fig. 3.12 shows the average initial solutions (normalized makespan) found by each heuristic after scheduling 60 different batches of tasks. The final evolved solution provides more efficient solutions on average than the solutions produced initially by the individual heuristics. The errorbars also show that the schedules produced by PN vary over a smaller range than the schedules produced by the other heuristics.

3.4.3 Performance evaluation

Each scheduler was presented with the same set of problems (see Sect. 2.3) and the same set of processors (see Table 3.3). The makespan is measured as the time from when the first task is requested from the distributed system, to the time when the final task is returned to the system. Table 3.7 shows that there is a huge difference in makespan (lower is better) with PN processing all tasks much faster than the next best scheduler when using a highly heterogeneous set of processors and networking resources. The variation in makespans can be accounted for by inefficient mappings of tasks to processors, such as slow processors being given computationally intensive tasks or

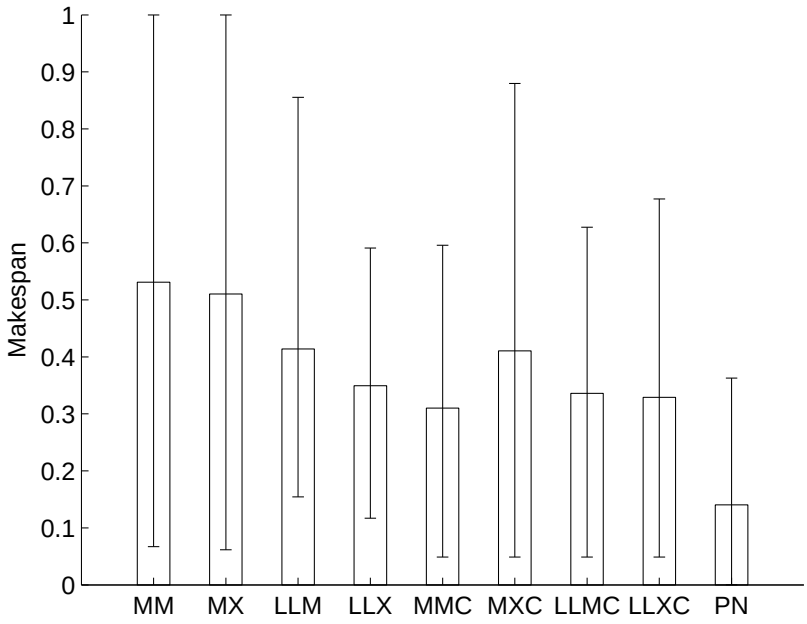

Figure 3.12: Performance of heuristics compared to our algorithm (PN) with real problems on a real heterogeneous distributed system, with normalized makespan.

No. Proc	MFLOP/s	RAM available (MB)	O/S	Network link (Mb/s)
9	214	257-296	Windows	10
7	244	100	Windows	100
3	255	261-265	Windows	10
2	223	257-267	Windows	10
2	255	100	Windows	100
1	32	100	Windows	10
1	221	64	Linux	100

Table 3.6: Client resources used in the distributed system for the experiment shown in Fig. 3.12.

processors with high communication overheads being given tasks with a low P-to-C ratio. The experiment was repeated with with a set of resources that displayed low heterogeneity (see Table 3.8.B). With less heterogeneity the difference in makespan is only 13% between the best (PN) and the worst (SA). With high heterogeneity this difference was 132%, with PN generating the lowest makespan.

When the experiment is repeated on a homogeneous set of processors the differences in makespan between the schedulers becomes negliable (see Table 3.9) with most schedulers utilizing the processing resources efficiently with up to 97% efficiency. PN, ZO and TA generate schedules which are within 1% of each other in this case and can adapt well to this homogeneous resource environment, which is to be expected. The simple heuristic sched-

Scheduler	Makespan (s)	Scheduling Time(s)	Mean Scheduling Time (s)	% Efficiency	% Communications	%Inefficiency
PN	9144	138.2	2.239	53.01	2.84	0
ZO	14278	276.1	1.904	33.94	2.24	56
TA	16378	322.8	4.818	33.66	2.34	79
SA	21260	4605.4	47.478	30.07	5.95	132
MX	15486	0.5	0.006	34.94	1.84	69
MM	18321	0.4	0.005	32.21	2.30	100
LL	19645	0.1	0.001	25.05	1.82	114
EF	14492	0.4	0.004	46.88	10.96	58
RR	20314	0.1	0.001	31.90	9.10	122

Table 3.7: Comparison of schedulers with a set of highly heterogeneous processors and a heterogeneous set of networking resources.

ulers generate solutions which have makespans which are 20-38% longer than the evolutionary algorithms.

Fig. 3.13 shows the number of idle clients while the set of problems is being processed using the PN scheduler in a highly heterogeneous resource environment. The initial assignment of tasks to processors does not happen instantaneously because the client machines only contact the server at set intervals (1 minute in this case). Near the end when the steep slope shows that all of the clients stop processing tasks within a short interval. If this was a shallow slope it would indicate processing resources are idle and underutilized.

Scheduler	Makespan (s)	Scheduling Time(s)	Mean Scheduling Time (s)	% Efficiency	% Communications	%Inefficiency
PN	8437	60.2	0.506	92.5	1.1	0
ZO	8593	39.4	0.210	90.6	0.9	2
TA	8767	39.5	0.376	88.4	1.4	4
SA	9564	3938.3	17.27	84.1	7.4	13
MX	9065	0.091	0.0008	87.3	1.1	7
MM	8860	0.166	0.0013	87.0	1.3	5
LL	9053	0.021	0.0002	87.0	0.9	7
EF	8602	0.089	0.0007	90.9	1.1	2
RR	8812	0.096	0.0006	88.4	0.9	4

Table 3.8: Comparison of schedulers with a set of 2 types of homogeneous processors and a heterogeneous set of networking resources.

Scheduler	Makespan (s)	Scheduling Time(s)	Mean Scheduling Time (s)	% Efficiency	% Communications	%Inefficiency
PN	10408	50.4	0.49	96.9	1.3	1
ZO	9969	21.7	0.17	97.6	1.2	0
TA	10126	22.3	0.23	97.5	1.3	1
SA	10351	1530.5	12.24	95.2	3.2	1
MX	12034	0.05	0.01	81.8	1.1	20
MM	13788	0.04	0.01	69.9	0.8	38
LL	13841	0.01	0.01	69.9	0.8	38
EF	13836	0.03	0.01	69.7	0.8	38

Table 3.9: Comparison of schedulers with a homogeneous set of processors.

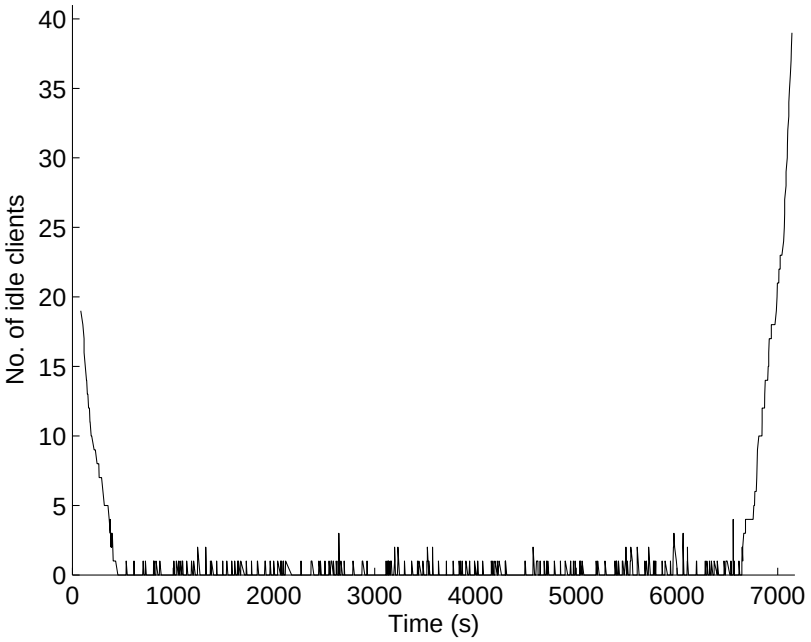

Figure 3.13: The number of idle clients in the system while the set of problems is being processed with the authors scheduling algorithm (PN)

3.5 Conclusion

A scheduler was developed for the task allocation problem in a dynamic heterogeneous distributed system. It is a multi-heuristic evolutionary algorithm, which utilizes a GA, to allocate tasks to processors in polynomial time. The use of eight heuristics to initialize the GA allowed for more efficient schedules to be created than would have been with a purely random initial population. If at any stage a processor becomes idle the scheduler returns the current best solution, which will always be at least as efficient as the best heuristic solution. The GA was implemented in Java and incorporated into a distributed system. A set of real-world problems from bioinformatics, bio-medical engineering and cryptography was used to test the scheduler. Experiments were performed up to 150 heterogeneous processors, and show that the scheduler presented in this chapter outperforms the most commonly used heterogeneous distributed computing scheduling heuristics. The more heterogeneous the resources of a system become, the harder it is to generate an efficient mapping of tasks to processors. We have presented an algorithm which achieves better efficiency than other schedulers as the resources become more heterogeneous.

For future work, the next logical step would be to distribute the scheduling algorithm to take full advantage of the available computational resources [4].

The GA based scheduler works well in many situations, however it does have a number of disadvantages, notably with predictability and verifiability. If the algorithm is given the same set of inputs multiple times, the same output solution is not guaranteed, due to the stochastic nature of this evolutionary algorithm. The running time needed to achieve a solution equal to,

or better than, a given makepsan, can also vary. This makes it unsuitable for situations with deadlines. Since GAs cannot be verified they are unsuitable for some applications, such as medical applications, due to the difficultly in verifying the output, or even understanding why a particular solution has been reached.

A simpler solution has been developed which addresses these disadvantages, whilst not making any additional restrictive assumptions. The uncertainty in the properties of the system is utilized to produce efficient schedules. It has been shown to produce solutions which are nearly as good as more complex evolutionary schedulers. It has a constant running time, so is suitable for applications with deadlines. There is no randomness, thus the same solution will be given for the same set of inputs. Its predictable and repeatable properties mean it can be verified, thus allowing for it to be used in situations not suited to an evolutionary algorithm.

Chapter 4

Task allocation using estimation error

This chapter is based on work presented in [80, 75].

In real-world dynamic heterogeneous distributed systems, allocating tasks to processors can be an inefficient process, due to the dynamic nature of the resources and the tasks to be processed. The information about these tasks and resources is not known a priori, and thus must be estimated online. In this chapter we utilize the accuracy of these estimates, and when combined with different objectives, such as minimizing makespan and evenly distributing load, naturally gives rise to a family of four different scheduling algorithms. The algorithms have been implemented in the a distributed system, evaluated with the set of problems described in Chapter 2. We have found that considering estimation error when allocating tasks to processors can provide more efficient solutions, than when estimation error is not considered.

We have found that using a simple heuristic, combined with estimation error, can in some cases provide solutions approaching the efficiency of complicated well-known evolutionary algorithms.

4.1 Introduction

The problem of estimating both the execution time of the tasks to be processed and the resources of the system has been tackled in a number of ways [5]. Some distributed systems, such as SETI@home [52], ignore the resources of the system [52, 54, 91], or treat their heterogeneous resources as a homogeneous set [6, 22, 52, 54, 74, 91, 101] by ignoring variation in the available computational resources of the processors. Some distributed systems restrict themselves to homogeneous tasks [52, 54, 91] which reduces the complexity of the scheduling problem.

Other algorithms require the user to define the length of time that a problem is expected to take [7, 21, 92, 96, 98]. Many require a directed acyclic graph with task and communication information, and precedence constraints given in advance [7, 21, 55, 57, 56, 92, 96, 98]. Communication costs are also not properly considered by many algorithms, for example all communication links are assumed to be homogeneous [102], it is assumed communication and computation can take place simultaneously [21], or it is assumed that there is instantaneous message passing [107]. The restrictive assumptions placed on the type of tasks that can be processed, and the processing and communication resources of the system simplifies the scheduling problem, but reduces the generality and usefulness of the solutions.

Some research has been done to address these restrictive assumptions. Sinnen *et al.* [92] look at a processor's involvement in communication and show that considering this involvement, when scheduling, leads to more efficient resource utilization in real-world distributed systems. Cohen *et al.* [13] focus on scheduling the communication between processors, to minimize the communication overhead in a distributed system. Theys *et al.* [98] generate and store many scheduling solutions before run-time, then select the most suitable schedules during run-time, which allows the scheduler to adapt to a variable task and resource environment. The dynamic level scheduling algorithm proposed by Dogan and Ozguner [23] addresses the variability of network and processor resources caused by failures, and attempts to minimize the probability of these failures adversely effecting the overall operation of the distributed system. Ali *et al.* [5] create a generalized robustness metric for unreliable parallel and distributed systems where the system resources may vary or the estimated task execution times may be erroneous.

In this chapter we present a scheduling algorithm which addresses these restrictive assumptions. In contrast to the techniques of the previous paragraph, the scheduler assumes that no knowledge is available a priori about the tasks to be processed, or the communication and computational resources of the distributed system. This information is dynamically estimated online.

We utilize the error in these estimates, seeking to schedule the tasks with the minimum estimated error earliest, or schedule the tasks with the most estimated error earliest. When this is combined with different objectives, such as minimizing makespan (total execution time) and evenly distributing

load, it naturally gives rise to a family of four different scheduling algorithms.

4.2 Task Scheduling

In a real-world online distributed system, the dynamic nature of the underlying resources of the system can limit the ability of traditional scheduling algorithms to function efficiently. We have created four scheduling algorithms, out of a possible family of 8 algorithms, which incorporate multiple different objectives and consider the error in the estimation of system resources and error in the estimation of task execution time (2 maximizing the error and 2 minimizing the error). These objectives are 1.) minimizing makespan, 2.) minimizing load imbalance and 3.) managing uncertainty.

For the remainder of this chapter we will define the percentage efficiency as

$$\text{efficiency} = (\sum_{j=1}^{M} \text{time processor } j \text{ has spent processing})/(\gamma \times M), \quad (4.1)$$

where M is the number of processors, and γ is the number of seconds since the scheduler instantiation.

In this section we will introduce a number of scheduling algorithms based on these objectives.

4.2.1 Estimation error

We must estimate the execution time of a task because it is undecidable to calculate exactly [93]. Problems consisting of homogeneous tasks will have

the least estimation error, while problems with complicated task execution time distributions will have a greater estimation error due to the increased complexity involved in modeling complicated distributions [11]. The percentage error in the ETC of task i on processor j is

$$\mathrm{ET}(i,j) = \left| \frac{\mathrm{ETC}(i,j) - t_i^j}{t_i^j} \right|, \tag{4.2}$$

where t_i^j is the actual time to compute task i on processor j.

The estimated computation to communication ratio (CCR) is defined as,

$$\mathrm{CCR}(i,j) = \frac{\mathrm{ETC}(i,j)}{\mathrm{C}(i,j)}, \tag{4.3}$$

where $\mathrm{C}(i,j)$ is calculated from Alg. 2.1 by substituting t_i for c_i.

A combined error weight value (EW) for a given task-processor mapping is defined as

$$\mathrm{EW}(i,j) = \frac{\mathrm{ET}(i,j)}{\mathrm{CCR}(i,j)}. \tag{4.4}$$

Eq. (4.4) produces a small value when the task error (ET) is small and the CCR is large. A small value is preferable to a large value in this instance. This allows for processors with the least communication costs, and least error to be differentiated from processors with less desirable properties. A large value of EW indicates a mapping which is possibly more erroneous.

We use two different strategies when handling the estimated task execution times and the predicted error. See Fig. 4.1 for an example gannt chart. The first is to ignore the predicted estimation error in the estimated task

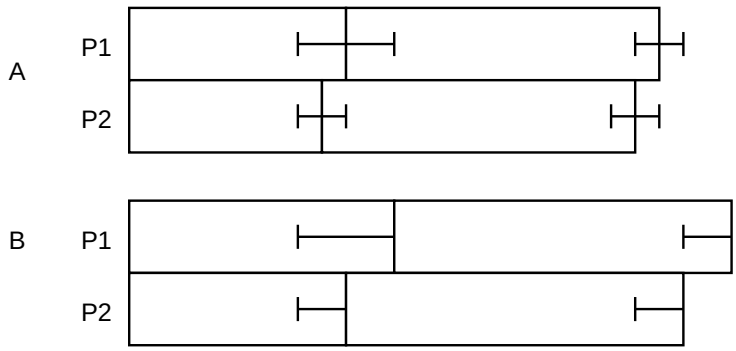

Figure 4.1: A gannt chart showing the best case (A) and worst case (B) estimated task execution times, with the error bars indicating predicted error.

execution time. We call this the best case scenario. The next strategy is to assume the worst case and to apply the maximum amount of predicted estimation error to the estimated task execution time.

4.2.2 Algorithm structure

Each of the algorithms described in this section can be described using the scheduling algorithm structure in Alg. 4.1, with different functions X affecting the different scheduling algorithms. The input to Alg. 4.1 is a set of tasks to be processed, the system's communication and processing resources, and a function X which is used to decide the task-processor mappings. This algorithm uses a greedy strategy and at each iteration selects the task-processor allocation which minimizes X.

```
Input: Set of unscheduled tasks, set of processors, load on each
       processor, an objective function X
Output: Mapping from tasks to processors, Q
Mapping is initialized to ∅;
while unscheduled tasks remaining do
    foreach Unscheduled task i do
        minval := MAX_INT;
        foreach Processor j do
            currval := X(i, j, load);
            if currval ≤ minval then
                a := i;
                b := j;
                minval := currval;
    Add (a, b) to the mapping Q;
    Update current load on Processor b;
    Remove Task a from list of unscheduled tasks;
```

Algorithm 4.1: The scheduling algorithm template, parametrized by objective function X

4.2.3 Minimizing Makespan

In this section we will describe three algorithms which seek to minimize the makespan, two of which use estimation error.

We wish to allocate tasks to processors, whilst minimizing the overall total execution time. The estimated makespan of a task i, allocated to a processor j, is

$$\mathrm{FA}(i,j) = \mathrm{ETC}(i,j) + \mathrm{C}(i,j) + \mathrm{ST}(Q_j), \tag{4.5}$$

where $\mathrm{ST}(Q_j)$ is the start time of processor j in seconds since the arrival of the first task for processing, defined as

$$\mathrm{ST}(j) = \sum \mathrm{ETC}(Q_j, j), \tag{4.6}$$

and Q_j contains all tasks that have been mapped to, or currently being processed by, processor j. FA is a cost function based on the Max-Min heuristic [40]. When used as the objective function, in place of X in Alg. 4.1, it will allocate a task to the processor which will finish processing it earliest.

Eq. (4.5) is combined with the estimation error value from Eq. (4.4) to produce two scheduling algorithms. The first is characterized by the cost function

$$\mathrm{FE}(i,j) = \mathrm{FA}(i,j)\mathrm{EW}(i,j)^{\beta}, \tag{4.7}$$

where β controls the exponential multiplier of EW and $\beta > 0$.

The FE heuristic aims to locally minimize the amount of estimation error

and maximize the CCR. By delaying the processing of more error-prone tasks, it gives the scheduler time to gather more observations about past estimations and can possibly be used to generate less error-prone estimates, which is then used to pre-emptively reschedule previously allocated tasks. Thus the tasks that are processed earliest are the tasks with the most accurate estimates. The effect of EW is controlled by β, with a proportionally large β reducing the effect of FA on the value of FE.

Fig. 4.2 is a simple example of the algorithm in operation with 2 processors and 4 tasks. At time 0, T3 and T4 are scheduled last due to the large estimation error, and FE attempts to minimize the overall makespan. At time 2 more information has become available, which improves the estimation of the task execution times for T3 and T4. The tasks are pre-emptively rescheduled and T3 is reassigned to P2 and T4 is reassigned to P1. The reduction in task execution time estimation error and the pre-emptive rescheduling leads to an overall reduced makespan at time 2, compared to the initial schedule.

The FZ cost function is defined as

$$\mathrm{FZ}(i,j) = \frac{\mathrm{FA}(i,j)}{\mathrm{EW}(i,j)^{\beta}}, \tag{4.8}$$

and it schedules tasks with the most estimation error earliest. It is not efficient at the beginning, but by processing the most error-prone tasks first, it allows for the tasks with the least amount of error to be scheduled last. This allows for greater confidence in the accuracy of the predicted makespan as the computation progresses, allowing for a more efficient global minimization of the total makespan.

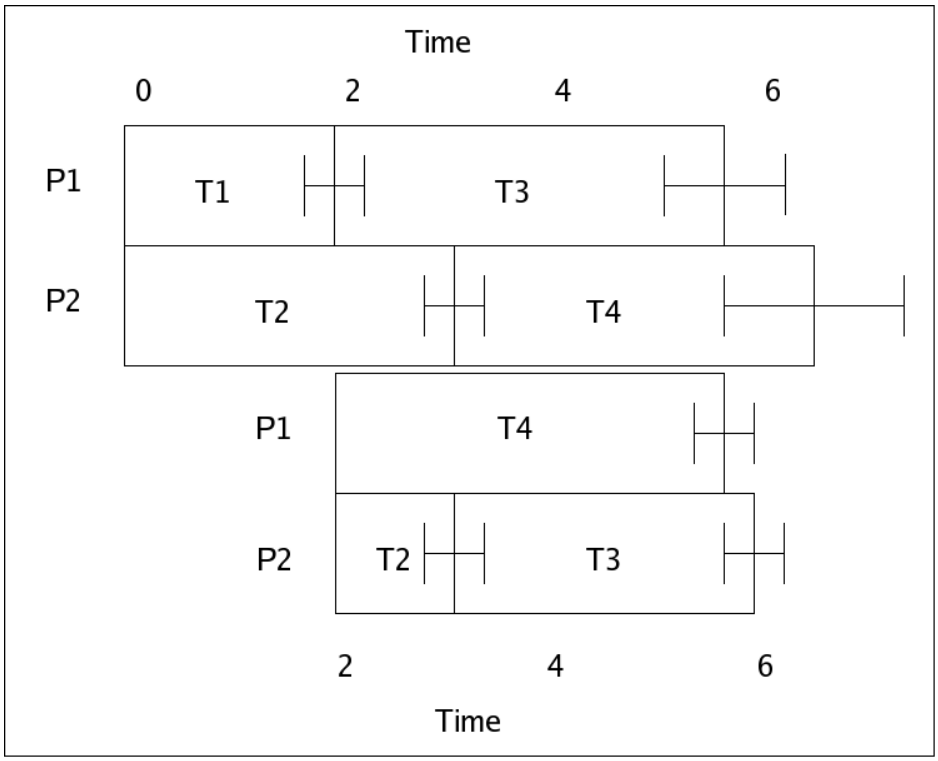

Figure 4.2: An example of the FE algorithm at time 0 and 2, with the error bars illustrating the bounds of the estimation error of the task execution time.

	EW $\rightarrow 0$	EW $\rightarrow \infty$
FA $\rightarrow 0$	$\underline{0}$	∞
FA $\rightarrow \infty$	∞	∞

Table 4.1: The FE scheduler favors a low execution time and low error. The limit of FE is shown where EW and FA tend towards 0 and ∞.

If β is static a problem can emerge where EW(i,j) drowns out FE and FZ. For example, if the makespan reduces over time, but the estimation error stays the same, the EW value increases its influence over the final scheduling decision over time. Thus in a system with very little difference in processor makespans, the scheduling decisions will be primarily influenced by the estimation error in the system, resulting in an inefficient solution. This can be rectified by controlling the influence exerted by EW by setting,

$$\mathrm{FE}_d(i,j) = \mathrm{FA}(i,j)^{\mathrm{EW}(i,j)}\mathrm{EW}(i,j)^{\mathrm{FA}(i,j)}, \tag{4.9}$$

thus as the variation in makespans on processors decreases, so does the influence exerted by EW over the final total makespan. Similarly a dynamic version of FZ is defined as follows,

$$\mathrm{FZ}_d(i,j) = \frac{\mathrm{FA}(i,j)^{\mathrm{EW}(i,j)}}{\mathrm{EW}(i,j)^{\mathrm{FA}(i,j)}}. \tag{4.10}$$

We tried each combination of EW and FA, where the resulting value is minimized. Tables 4.1-4.4 show the characteristics favored in each case (underlined), when the values tend towards zero and ∞. FE and FZ aim for low execution times. The two other combinations were unfeasible because they favour high execution times.

	EW $\rightarrow 0$	EW $\rightarrow \infty$
FA $\rightarrow 0$	1	$\underline{0}$
FA $\rightarrow \infty$	∞	1

Table 4.2: The FZ scheduler favors a low execution time and high error. The limit of FZ is shown where EW and FA tend towards 0 and ∞.

	EW $\rightarrow 0$	EW $\rightarrow \infty$
FA $\rightarrow 0$	1	∞
FA $\rightarrow \infty$	$\underline{0}$	1

Table 4.3: A scheduler with EW/FA favors a high execution time and low error making it unfeasible. The limit is shown where EW and FA tend towards 0 and ∞.

	EW $\rightarrow 0$	EW $\rightarrow \infty$
FA $\rightarrow 0$	∞	0
FA $\rightarrow \infty$	0	$\underline{0}$

Table 4.4: The 1/(FA*EW) scheduler favors a high execution time and high error making it unfeasible. . The limit is shown where EW and FA tend towards 0 and ∞.

4.2.4 Load-balancing

Evenly distributing the load on each processor in the distributed system is a common goal in a real-world distributed system. This aims to maximize the utilization (or efficiency) of the processing resources. A load-balancing weighting,

$$\begin{aligned} \mathrm{LA}(i,j) = \; & \textstyle\sum_{y=1}^{M} \big[(\max\big\{\max_{x=1}^{M}[\mathrm{ST}(x)], \mathrm{FA}(i,j)\big\} \\ & -\mathrm{ST}(y)) - \{\mathrm{ETC}(i,j) + \mathrm{C}(i,j)\}] \\ & \times \Big[\textstyle\sum_{y=1}^{M} \max_{x=1}^{M}\{\mathrm{ST}(x) - \mathrm{ST}(y)\}\Big]^{-1}, \end{aligned} \tag{4.11}$$

considers the current inefficiency of the resource utilization, and calculates the effect allocating task i to processor j will have on the overall efficiency of the system. A low value of LA corresponds to a well balanced system, whereas a high value indicates an inefficient utilization of resources.

LA is combined with EW to create two scheduling algorithms,

$$\mathrm{LE}(i,j) = \mathrm{LA}(i,j)\mathrm{EW}(i,j)^{\beta} \tag{4.12}$$

and

$$\mathrm{LZ}(i,j) = \frac{\mathrm{LA}(i,j)}{\mathrm{EW}(i,j)^{\beta}} \tag{4.13}$$

which consider the estimation error along with the load of the system. In the LE algorithm the task-processor mappings which reduce the load imbalance the most, and have the lowest estimation error, are allocated first. The most

error-prone tasks will have the least effect on the overall load of the system.

The LZ scheduler aims to schedule tasks with the most estimation error earliest whilst also seeking to minimizing the load imbalance (see Eq. (4.13)). Over a number of batches of tasks LZ allocate the most error-prone tasks first, allowing for the least error-prone tasks to be load balanced at the end. LE and LZ reduce to each other (as with FE and FZ), and they also both reduce to LA.

4.2.5 Suitability matching

Matching attempts to match the computational requirements of a task to an appropriate processor, whilst also considering the communication resources available. The suitability of a task-processor mapping is generated by considering all processing and communication resources in the system, all previously processed tasks, and all un-mapped tasks. In a dynamic system the resource environment will vary over time, giving rise to error in our estimates of system resources, thus leading to inefficient task-processor mappings. If all tasks are suitably matched to processors, the amount of error due to inefficient matching can be minimized. For example, the most suitable processor for tasks with large amounts of data, would be a processor with a high bandwidth network link to the scheduler. A task which requires a large amount of computation would be most suited to a powerful processor. By matching tasks to a more suitable processor, it leaves the possibility of a previously unseen future task to be processed on a more suitably matched processor.

The suitability,

$$\mathrm{PS}(i,j) = 1 - \left| \frac{(T_i - \min(T))}{\sum_{x=1}^{N}(T_x - \min(T))} \times \frac{\sum_{y=1}^{M}(P_y - \min(P))}{(P_j - \min(P))} \right|, \qquad (4.14)$$

of mapping a task i to a processor j. All tasks T, measured in MFLOP, and all processors P, measured in MFLOP/s, are considered when generating the suitability value for a mapping. Only the processing resources are considered in this equation. The suitability of the communication resources is considered by,

$$\mathrm{CS}(i,j) = 1 - \left| \frac{(A_i - \min(A))}{\sum_{x=1}^{N}(A_x - \min(A))} \times \frac{\sum_{y=1}^{M}(B_y - \min(B))}{(B_j - \min(B))} \right|. \qquad (4.15)$$

A_i gives the size of task i in bytes and B_j gives the bandwidth of the communication resources from the scheduler to processor j in bytes per second. Eq. (4.14) and Eq. (4.15) are brought together to produce an overall suitability measure,

$$\mathrm{SB}(i,j) = \mathrm{PS}(i,j) + \frac{\mathrm{CS}(i,j)}{\mathrm{CCR}(i,j)}, \qquad (4.16)$$

for a given task-processor mapping which weights the importance of the computation and communication resources using the estimated CCR from Eq. (4.3).

A heuristic,

$$SE(i,j) = SB(i,j)EW(i,j)^{\beta}, \qquad (4.17)$$

which minimizes error and considers the suitability of task-processor mappings has been developed. This allows for the most accurate matching of

tasks to suitable processors. By matching suitable tasks to processors and communication resources, it is hoped that overall when more unseen tasks are presented for scheduling, that adequate suitable resources will be available to process the tasks. Modifying Eq. (4.17) gives

$$SZ(i,j) = \frac{SB(i,j)}{EW(i,j)^{\beta}}, \tag{4.18}$$

which maximize the error whilst attempting to match tasks to suitable processing and communication resources.

The use of SB on its own is not sufficient to generate an efficient mapping of tasks on processors. The task requirement space would need to map perfectly on to the system resource space for an efficient solution to be found, which is an unrealistic restrictive assumption. To more efficiently utilize SB to we need to combine it with another objective, such as FA or LA. A taxonomy of all the schedulers presented in this section is given in Table. 4.5.

4.3 Experiments

For the experiments described in this section we used two distributed system configurations, one with 90 PCs (Table. 4.6.A) and one with 74 PCs (Table. 4.6.B). The processor speeds varied by up to 10%, due to slightly differing hardware and software configurations. All experiments were carried out on system A with the exception of the experiments in Section 4.3.2 which were carried out on system B. All resources were non-dedicated, running Linux, and were connected by a 100 Mb/s network. The clients were

Symbol	β	Description	Equation
FA(i,j)	0	Makespan	Eq. (4.5)
FE(i,j)	+	Min makespan, Min error	Eq. (4.7)
FZ(i,j)	-	Min makespan, Max error	Eq. (4.8)
FE_d(i,j)	+	Min makespan, Min error	Eq. (4.9)
FZ_d(i,j)	-	Min makespan, Max error	Eq. (4.10)
LA(i,j)	0	Min load-imbalance	Eq. (4.11)
LE(i,j)	+	Min load-imbalance, min error	Eq. (4.12)
LZ(i,j)	-	Min load-imbalance, max error	Eq. (4.13)
SB(i,j)	0	Min unsuitability	Eq. (4.16)
SE(i,j)	+	Min unsuitability, Min error	Eq. (4.17)
SZ(i,j)	-	Min unsuitability, Max error	Eq. (4.18)

Table 4.5: Taxonomy of schedulers, where + indicates any positive number, and - indicates any negative number.

System	No. Proc	MFLOP/s	RAM (MB)	Processor
A	45	28-31	256	P3 600MHz
	45	180-200	1024	P4 D820
B	38	28-31	256	P3 600MHz
	36	180-200	1024	P4 D820

Table 4.6: Client resources of two heterogeneous distributed systems (A and B)

connected to a dedicated server running Linux on a 3 GHz P4 with 1 GB of RAM. We used a single core on the P4 D820 processors running a 32-bit version of Linux. The set of tasks to test the schedulers are described in Chapter 2.3, from the fields of bioinformatics, biomedical engineering and cryptography.

Scheduler	Makespan (s)	Sched. time (s)	% Efficiency	% Comms
FA	3505	28	51	0.6
FE	6202	547	33	0.8
FZ	2408	108	66	1.6
LA	4114	344	55	0.7
LE	17013	1850	21	0.3
LZ	12138	426	33	0.3
SB	24188	1229	14	0.2
SE	12553	1956	22	0.3
SZ	13349	1741	23	0.3

Table 4.7: Comparison of schedulers.

4.3.1 Scheduler Performance

Two different metrics are used to evaluate the performance of the schedulers given in this chapter: 1.) makespan, which is the total execution time, and 2.) efficiency, which is defined as the percentage of time the processing resources are in use. Each set of algorithms has been grouped together based on their on their common objective, comparing each algorithm with estimation error and without estimation error. A trace of the efficiency is given over time in Figs. 4.3-4.5, comparing each family of algorithm.

The load-based schedulers (LA, LE, LZ) provide approximately 21-55% efficiency overall, with LA providing the most efficient solution, as shown in Fig. 4.3. Using estimation error along with load provides poor efficiency and makespan when a static β is used. The makespan of LE is more than 4 times greater than LA, so using estimation error with the load-balancing objective results in very large total execution times.

The suitability-based schedulers (SB, SE, SZ) provide the worst overall efficiency. This is because the suitability objective does not consider any

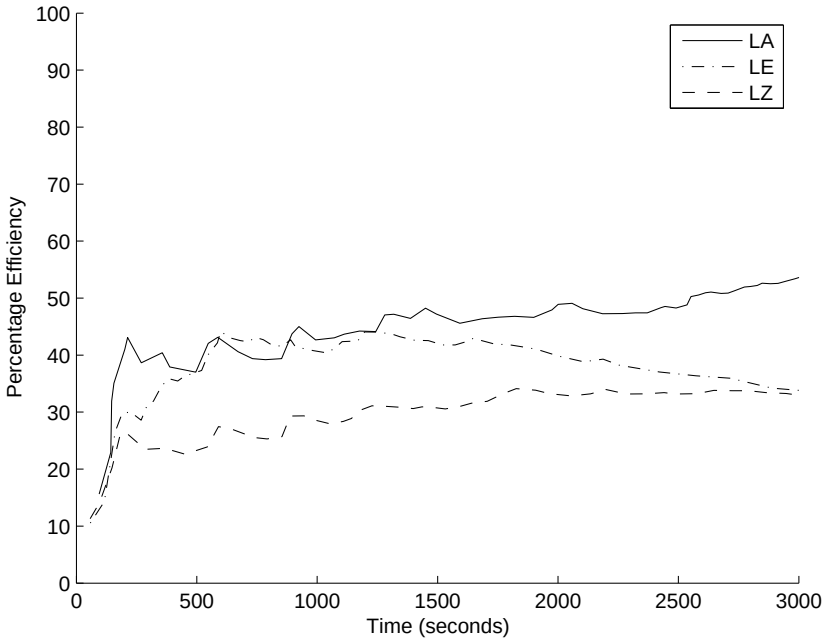

Figure 4.3: The efficiency of 3 load-based schedulers over time

property related to minimizing efficiency or makespan. While SB more efficiently utilizes the processing resources (see Fig. 4.4), this does not translate into a low total execution time. For example, if there is a low powered processor in the system, a lower makespan may be achieveable by leaving the processor idle. This however would negatively impact on the efficiency of the resource utilization. The makespan of SE and SZ is nearly half that of SB (see Table 4.7). The difference is greatest with these schedulers because the suitability objective relies solely on the accuracy of the estimation of the task execution times, and the processor and communication resources. Thus utilizing the estimation error has greatly improved the accuracy of this scheduling objective in a real-world distributed system.

The makespan-based schedulers (FA, FE, FZ) provide the best overall efficiency achieving above 80% at some points, as shown in Fig. 4.5. It is interesting to note that FA provides the most efficient utilization of resources but the makespan of FZ is the lowest of the schedulers described in this chapter at 2408 seconds. So although FA utilizes the processing resources for a higher % of time, the heterogeneous nature of these resources means that the makespan does not follow suit. Overall FA only achieves an efficiency of 51% compared to FZ which achieves an overall efficiency of 66% (see Table 4.7). The makespan of FA is also 45% higher than that of FZ. FZ achieves this reduced makespan by utilising estimation error. FA is a simple heuristic, and with the addition of estimation error, this heuristic can provide a low makespan, without the added complexity of other algorithms which achieve similar results.

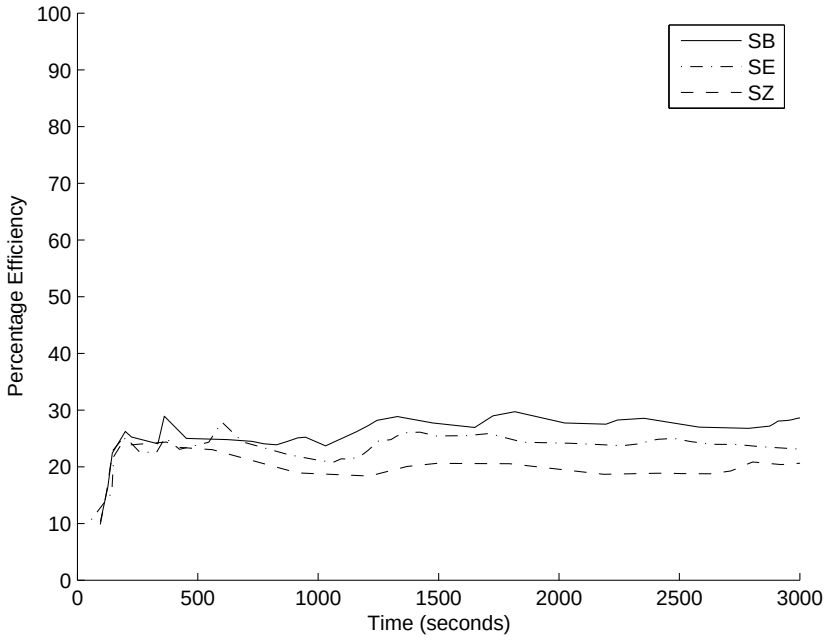

Figure 4.4: The efficiency of 3 suitability-based schedulers over time

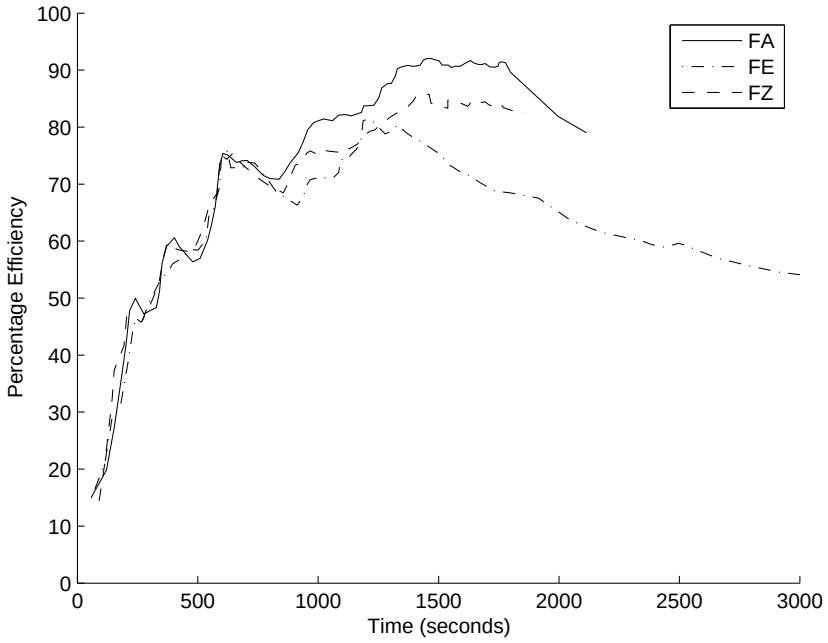

Figure 4.5: The efficiency of 3 makespan-based schedulers over time

Scheduler	Makespan (s)	Sched. time (s)	% Efficiency	% Comms
FZ	2408	108.0	66	1.6
TA	2351	11.1	76	1.1
SA	9252	836.1	35	1.7
LL	3066	0.1	62	0.7
EF	3096	0.1	55	0.7
RR	6176	0.1	38	0.4

Table 4.8: Comparison of common schedulers which do not use estimation error to FZ, the best performing scheduler, which uses estimation error.

In nearly all cases (the exception being the load-based schedulers) the error based schedulers provide better efficiency than their non-error based counterparts. Thus the addition of estimation error can improve upon simple heuristics.

Only the schedulers which try to minimize makespan provide a high level of efficiency. We have compared the most efficient algorithm FZ to a number of commonly used algorithms (see Table 4.8). Tabu search optimization (TA) is an evolutionary based scheduler based on OpenTS [73].

A simulated annealing (SA) based scheduler was created using the Jannealer API [43]. These are complicated meta-heuristic algorithms, which use evolutionary techniques to generate solutions. With the Tabu and simulated annealing algorithms, the value of the parameters can have a huge impact on the end result. We fine tuned the implementations of TA and SA to the data, to ensure a good comparison was available. With TA, we recursively broke down the problem to be solved into a tree like structure of depth $logN$ and optimized each piece. This resulted in a fast convergence to a solution, but is less useful for a generalized data-set. The parameters of the SA scheduling

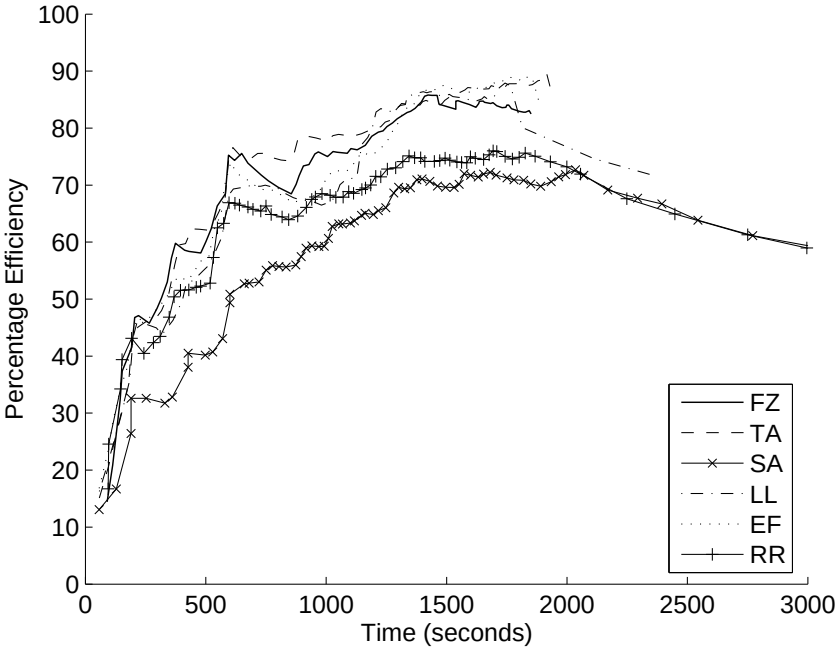

Figure 4.6: The efficiency of FZ over time compared to evolutionary and heuristic schedulers which do not use estimation error.

algorithm were calculated using another SA instance.

Two immediate mode schedulers have been implemented, lightest-loaded (LL) which assigns tasks to the lightest loaded processors, and earliest first [58] (EF) which assigns tasks to the processors which will finish processing them earliest. Round robin (RR), is the simplest and one of the most commonly used schedulers. All of these schedulers used the same input parameters such as, estimated task execution times, estimated processor speeds and estimated communication resources.

It is interesting to note that although FZ does not achieve the best efficiency overall, it does achieve one of the lowest makespans. This is because in a heterogeneous distributed system, maximizing resource utilization does not correspond to minimizing makespan.

As can be seen in Table 4.8, FZ has a makespan of 2408 seconds, which is between 27% and 284% better than the other schedulers with the exception of TA, which has a makespan of 2351 seconds. FZ is based on a very simple heuristic combined with estimation error, whereas TA is a complicated stochastic evolutionary algorithm, which has been fine tuned to suit the dataset. By considering estimation error, a simple heuristic can achieve nearly the same makespan as a state-of-the-art evolutionary technique.

4.3.2 Varying the Error Weight

We varied β to change the effect EW has in FE and FZ. We used 1, 0.5, and 0.1 as well as a dynamically (d) varying β. Each experiment was performed using 74 heterogeneous processors as described in Table 4.6.B. Table 4.9

	Strategy	β	Avg. Makespan (s)	Avg. Scheduling time (s)	% Efficiency	% Communications
FA	b	-	3203	89	65	1.1
	w	-	3130	117	66	1.2
FE	b	1.0	9279	587	26	0.5
	w	1.0	8142	354	36	0.5
	b	0.5	5498	276	48	0.7
	w	0.5	6029	354	43	0.8
	b	0.1	7094	240	42	0.5
	w	0.1	6889	257	40	0.6
	b	d	5602	442	43	1.5
	w	d	9136	167	34	0.4
FZ	b	1.0	3100	2.2	63	1.5
	w	1.0	3270	1.2	59	0.8
	b	0.5	2968	85	65	1.1
	w	0.5	2793	1.1	73	1.0
	b	0.1	2510	1.3	83	1.2
	w	0.1	2762	35	64	0.8
	b	d	3026	0.5	67	0.8
	w	d	2814	2.2	72	1.7

Table 4.9: Experiments with 74 processors (see Table 4.6.B) varying the value of β, where d is a dynamic value of β (see Eq. (4.9)). b and w are the best and worst case strategies respectively.

describes the results of these experiments. Each experiment was repeated twice, and the average is given.

FA which does not consider estimation error is used as a benchmark. We also investigated using best and worst case values for task execution times. With the best case (b) we take the mean estimated task execution times as given by the k-NN algorithm, when scheduling. With the worst case (w), we add the maximum amount of estimated error to the estimated task execution times. The algorithm is however quite robust to using both b and w.

Overall FZ performed best providing a makespan which was 20% lower than FA, when using a best case task execution time and $\beta = 0.1$. FE did not perform well when compared to FZ or FA, consistently producing schedules with large makespans. The makespan produced by using the worst case task execution times is variable, whereas the makespan produced when using the best case is more stable and predictable.

Fig. 4.7 shows the efficiency of using FZ with best case task execution times, whilst varying the value of β. With $\beta = 0.1$ the efficiency is better than the other values of β. While they all begin with similar efficiency, as time goes on, the effect of the estimation error on the overall efficiency becomes apparent, with each becoming clearly delineated.

Fig. 4.8 shows the efficiency when using FZ with worst case task execution times, whilst varying the value of β. Once again the efficiency of each begins very similarly, but becomes more delineated with time. There is very little difference between a β value of 0.1 or 0.5 in terms of makespan, but it is interesting to note the difference in efficiency. A β value of 0.1 achieves

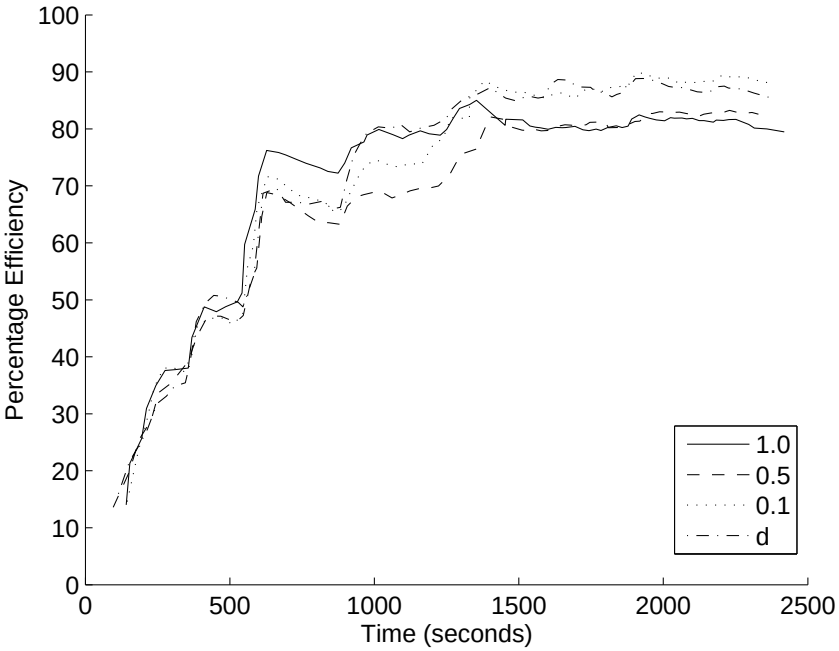

Figure 4.7: Efficiency of using FZ with varying values of β with best case task execution times.

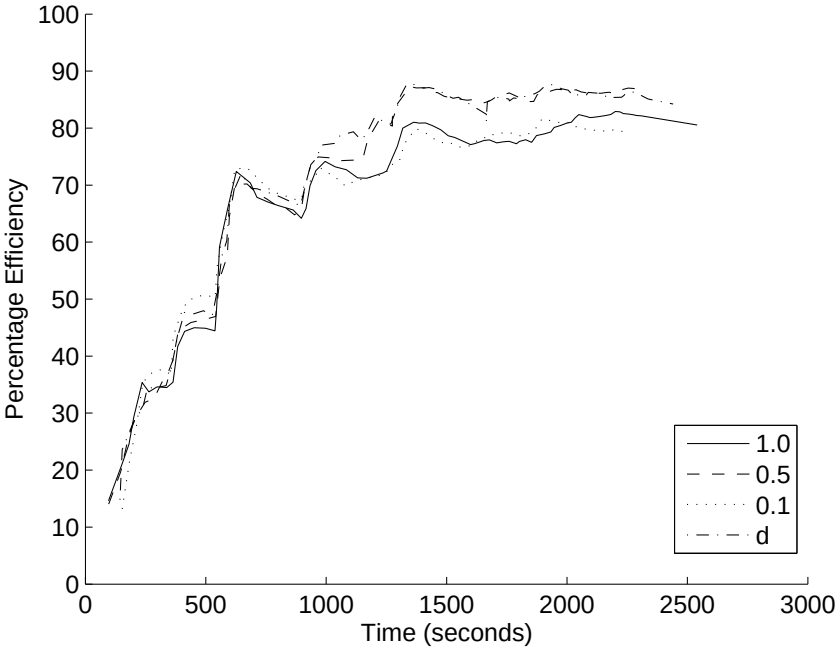

Figure 4.8: Efficiency of using FZ with varying values of β with worst case task execution times.

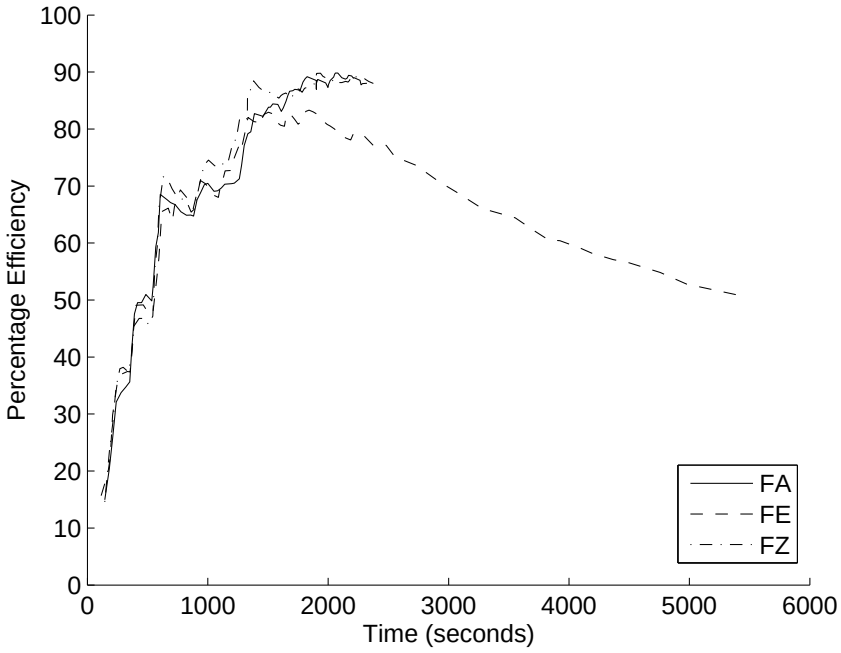

Figure 4.9: Efficiency of multiple schedulers with best case task execution times.

an efficiency of only 65% whilst a β value of 0.5 achieves an efficiency of 73%. This huge difference is due to the types of processors in the distributed system, where the slowest processor has only approximately 15% of the computational resources of the fastest processor. Thus a schedule which utilizes the faster processors more of the time over the slower processors can have a lower makespan but also a lower overall efficiency.

Figs. 4.9 and 4.10 compare the efficiency of FA, FE, and FZ using best and worst case task execution times respectively. The best performing β value for FE and FZ is used in each Fig (see Table 4.9). FE is clearly far less efficient than FA or FZ. This is consistent in all experiments (see Table 4.9), where

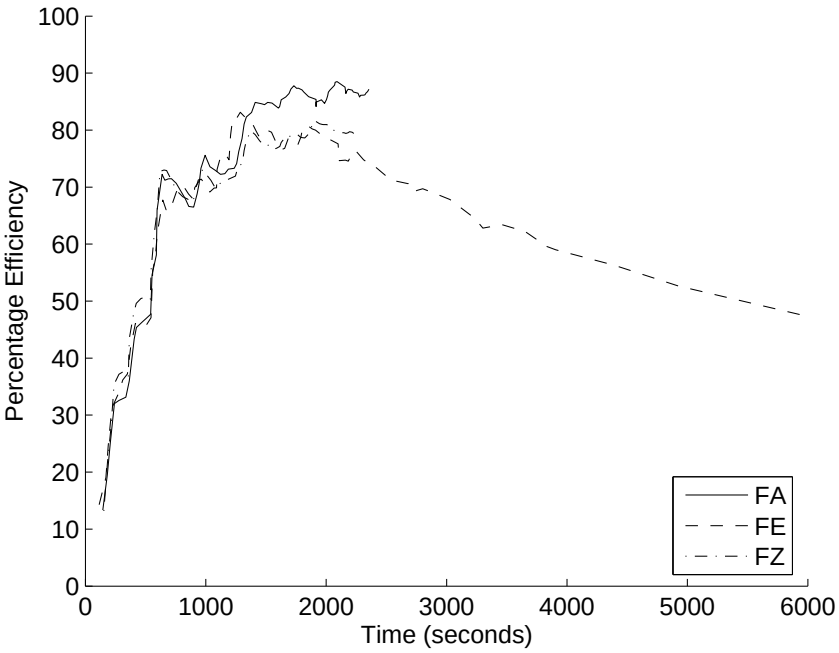

Figure 4.10: Efficiency of multiple schedulers with worst case task execution times.

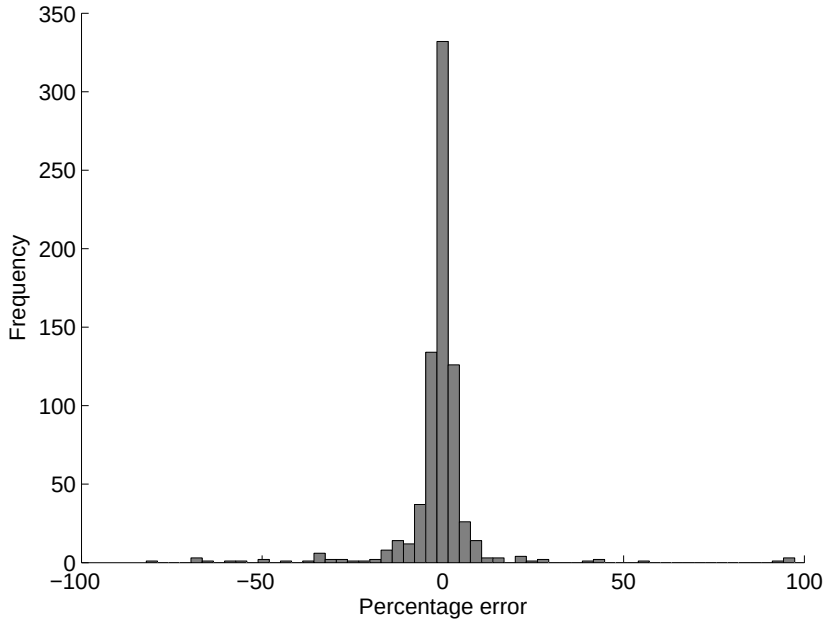

Figure 4.11: Histogram of the error between the estimated and the actual task execution times, with outliers outside [-100,100] removed.

FE schedules tasks on the slowest processors in the system near the end of the overall schedule, resulting in a large makespan. FZ in both cases trails off, which indicates that computationally large tasks have been allocated to slow processors.

The error between the actual and the estimated task execution times forms a normal distribution. The outliers (representing 1% of the total number of tasks) have been removed, as have results with no estimated execution time, generated at the beginning of the experiement. The best case task execution time strategy performs well because the mean is centered around 0.

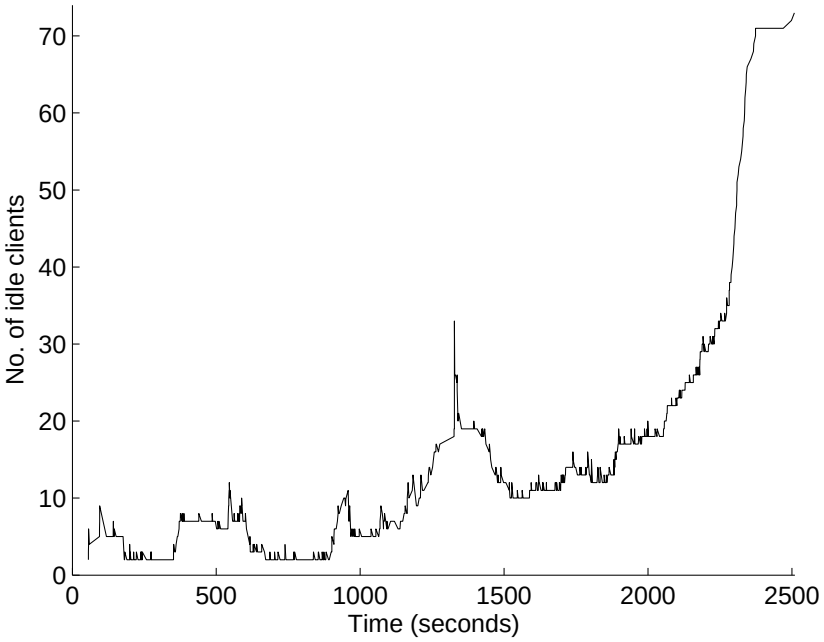

Figure 4.12: Number of idle processors over time when using FZ with $\beta = 0.1$ and best case task execution times.

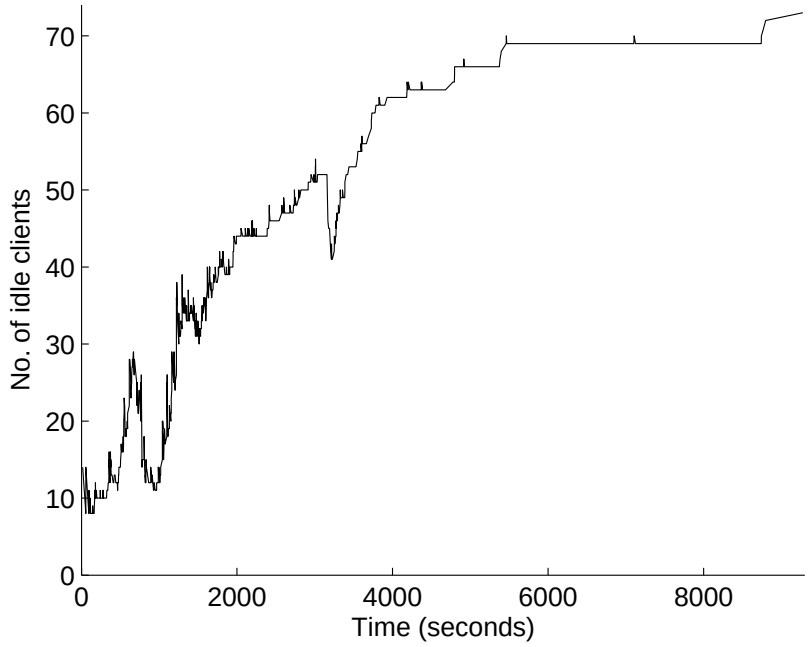

Figure 4.13: Number of idle processors over time when using FE with $\beta = 1.0$ and best case task execution times.

Fig. 4.12 shows the number of idle processors at a given point in time for FZ, using $\beta = 0.1$ and a best case task execution time. The number of idle processors increase throughout the computation as the scheduler decides not to schedule tasks on the slowest processors which have a lower computational capacity. The steep slope at the end of Fig. 4.12 indicates all processors finished processing within a short time frame. The large increase in the number of idle processors at time 1400 is due to the staged nature of some of the problems in the problem set (described in Chap. 2.3).

Compare this to the worst performing schedule in Fig. 4.13, using FE with $\beta = 1.0$, using best case task execution times. The slope at the end is slowly

increasing, indicating that tasks where allocated to the slowest processor in the system, leading to a high number of idle processors.

4.4 Conclusion

Processing problems efficiently and quickly, using a distributed system which utilizes the spare clock cycles of donated PCs, is very problematic. The available processing and network resources can vary without warning, greatly impacting on the makespan of problems being processed. The problems themselves can contain vastly different task distributions, adding more complexity to the scheduling problem. Assumptions generally used about the resources, and the tasks to be processed, restrict the usefulness of many schedulers, to the point where they can only perform well in simulated sterile setups, and are less useful for real-world distributed systems. These real-world complexities have been successfully addressed with the use of complicated evolutionary scheduling heuristics.

We have shown that it is possible to manage these real-world complexities with a simple scheduler, and achieve nearly the same makespan and efficiency as a complex evolutionary scheduler. We focused on managing the uncertainty of the state of the system and of the estimation of the tasks computational requirements, to reduce total execution time and to improve the efficiency of resource utilization. Less erroneous property estimation is essential to producing an accurate schedule. Otherwise, the actual execution time will overrun the planned processing time. By accepting that errors will be inherent in these estimations, we can factor this into scheduling algorithms,

thus leading to a more accurate, and lower, total execution times.

The FZ algorithm is shown to be robust to a variety of different conditions and input parameters, and consistently produces schedules which have a low makespan. With both the best and worst task execution times, the FZ scheduler is consistent in the low makespans produced. It performs nearly as well as a complex evolutionary heuristic, which has been finely tuned to suit the input data.

A preliminary investigation has been done on combining the different objectives listed in this chapter to further improve the overall efficiency of the scheduler. Multi-objective optimization algorithms, such as AbYSS [71], NSGA-II [18] and SPEA2 [104], have shown that this is a fruitful research path. Early results show that this strategy holds much promise as the next step in this research.

Chapter 5

Low memory distributed reconstruction of large digital holograms

Parts of this chapter have also been published in the following articles [3, 76]. This chapter describes a distributed application, which provides the impetus for the scheduling research, presented in this thesis, by providing computationally intensive real-world problems which need to be processed as quickly as possible.

We present a parallel implementation of the Fresnel transform suitable for reconstructing large digital holograms. Our method has a small memory footprint and utilizes the spare resources of a distributed set of desktop PCs connected by a network. We show how we parallelize the Fresnel transform and discuss how it is constrained by computer and communication resources.

Finally, we demonstrate how a 4.3 gigapixel digital hologram can be reconstructed and how the efficiency of the method changes for different memory and processor configurations.

5.1 Introduction

In digital holography, optically captured digital holograms are usually reconstructed by algorithmic means using a computer. However, with continuing advances in CCD technology, novel methods are required to keep pace with the growing volume of data [69], and the associated increased computational requirements. In processor technology, the number of computations a single processor can perform per second has stagnated and recent developments have focused on adding multiple cores to a CPU. Thus, using a single powerful processor may no longer keep pace with the increased computational requirements for digital holography. In this chapter we address this problem by implementing a parallel algorithm for digital hologram reconstruction suitable for distributed computing systems as well as multi-core processors.

While some effort has gone into accelerating computer generated holograms using algorithmic means [67], graphics processing units [2, 64, 86] and specialized hardware [41, 63], digital reconstruction of optically captured holograms has received little attention. A discussion on algorithmic optimization to Fresnel-like transforms has been presented [36], however due to the fact that the Fresnel transform is very efficiently implemented using the fast Fourier transform (FFT) there has been little need for special-purpose methods. It can be expected that the computational cost and memory re-

quirements for hologram reconstruction will increase at a higher rate than computer technology improvements in the future. To address this we have constructed a Fresnel method that uses the spare processing and memory resources of a distributed set of desktop PCs to reconstruct large holograms.

To our knowledge this is the first time the Fresnel transform has been parallelized on a distributed system. While parallel versions of the FFT have been proposed [44, 84], none of them have been used to implement hologram reconstruction in a distributed system.

5.2 Methods for hologram reconstruction

Given a hologram distribution, U, we can reconstruct the object image in a plane parallel to the hologram plane and at distance, d, by modeling the light propagation. The operation is depicted in Fig. 5.1. Light propagation between parallel planes can be mathematically describe by the Fresnel-Kirchhoff integral

$$W(u,v) = \frac{i}{\lambda} \iint_{-\infty}^{\infty} U(x,y) \exp\left[\frac{-2\pi i}{\lambda} r(x,y,u,v)\right] dxdy, \qquad (5.1)$$

where $r(x,y,u,v) = [(x-u)^2 + (y-v)^2 + d^2]^{\frac{1}{2}}$ and λ is the wavelength of the light source.

If the reconstruction distance, d, is large compared to the hologram size, the Fresnel or paraxial, where r is substituted by the linear and quadratic

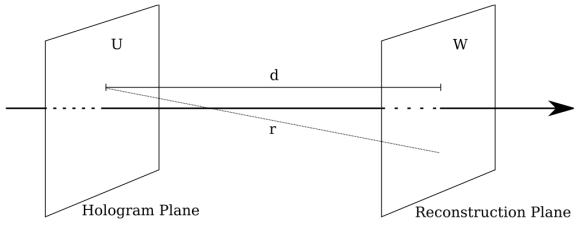

Figure 5.1: Hologram and Reconstruction planes at a distance d. The distance r is measured between each data point pair of U and W.

terms of its Taylor expansion, is valid [31, 87]. This leads to the expression

$$W(u,v) = \frac{i}{\lambda}\exp\left(\frac{-2\pi i}{\lambda}d\right)\exp\left[\frac{-i\pi}{\lambda d}\left(u^2+v^2\right)\right] \times \iint_{-\infty}^{\infty} U(x,y)\exp\left[\frac{-i\pi}{\lambda d}\left(x^2+y^2\right)\right] \times \exp\left[\frac{2\pi i}{\lambda d}\left(xu+yv\right)\right]dxdy. \tag{5.2}$$

$W(u,v)$ is called the Fresnel Transform of $U(x,y)$

In digital holography the hologram, $U(x,y)$, will be represented as a real or complex valued array of size $N \times M$ elements. Discrete numerical reconstruction methods could be based on Eq. (5.1) or Eq. (5.2). However, these would yield $\mathcal{O}(n^2)$ complexity for a full image reconstruction of n samples.

Instead it has been shown in literature that both equations can be expressed using the Fourier Transform [53, 87]. This allows for implementation using the Fast Fourier Transform (FFT) with complexity $\mathcal{O}(n \log n)$, resulting in better numerical efficiency.

We are concerned with numerically calculating Eq. (5.2) for some known $U(x, y)$. It has been shown that Eq. (5.2) can be rewritten in terms of the continuous Fourier transform [31]. By employing a discrete Fourier transform operator (which can be implemented using the FFT), denoted as $\mathcal{F}$ below, one may easily derive the two algorithms below [36, 53, 87].

The convolution approach is described by the following equation

$$W_d(m,n) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{U\right\}\exp\left\{\frac{2\pi i d}{\lambda}\right\}\right. \\ \left.\times\exp\left\{-\pi i\lambda d\left[\left(\frac{m}{M\delta_m}\right)^2+\left(\frac{n}{N\delta_n}\right)^2\right]\right\}\right\}, \tag{5.3}$$

where $m \in [-\frac{M}{2}, \frac{M}{2})$ and $n \in [-\frac{N}{2}, \frac{N}{2})$ are discrete coordinates and $\mathcal{F}$ denotes the discrete Fourier transform. The convolution method is based on the observation that the original problem can be formulated as a convolution between the hologram function and a phase function. Thus the convolution theorem may be applied to express the operation as a multiplication in frequency space.

The direct method is derived from Eq. (5.2) by rewriting it as a Fourier

transform of the hologram times a phase factor. It is expressed by

$$W_d(m,n) = \mathcal{F}\left\{U \exp\left\{\frac{\pi i}{\lambda d}\left[(m\delta_m)^2 + (n\delta_n)^2\right]\right\}\right\} \times \exp\left\{\frac{2\pi i d}{\lambda} - \pi i \lambda d\left[\left(\frac{m}{M\delta_m}\right)^2 + \left(\frac{n}{N\delta_n}\right)^2\right]\right\}. \tag{5.4}$$

While the direct method requires only one Fourier transform and thus is less computationally expensive than the convolution approach, it effectively changes the size of the reconstructed image. In other words, the output pixel size is linearly proportional to the distance parameter, d. The larger d, the larger the pixel size of the reconstruction and therefore the larger the spatial area of the the reconstruction. This property is sometimes undesirable. The convolution method on the other hand will keep the size of each sample constant and thus is more suited for hologram analysis approaches where object sizes must be comparable. We have decided to focus on parallelizing the convolution method because we wish to keep the resolution constant and generate more accurate results for comparison purposes.

A hologram encodes both phase and amplitude and thus represents the full light-field at the sensor. This allows us to reconstruct different views of the captured scene. By only considering a sub-area of the total hologram, we are effectively creating a camera with a smaller aperture and a consequent decrease in the resolution of the reconstruction. However, not only will this procedure lead to an increased depth of field, it will effectively reconstruct an image based on light coming from only certain directions of the scene. Thus, the location of the aperture relative to the optical axis will dictate the view

imaged through it. Therefore, by choosing size and location in the hologram plane different perspectives can be reconstructed.

The aperture procedure can be written as

$$U_A = T_{k,l}\left\{UA_{k,l}^{s,t}\right\}\exp\left\{\frac{2\pi i}{\lambda d}\left(km\delta_m^2 + ln\delta_m^2\right)\right\}, \tag{5.5}$$

where $A_{k,l}^{s,t}$ is a binary valued box aperture function of dimensions $s \times t$, with its center located at the discrete coordinates (k, l) in the hologram plane. It is defined as follows:

$$A_{k,l}^{s,t}(m,n) = \begin{cases} 1 & \text{if } (m-k, n-l) \in [\ \frac{-s}{2}, \frac{s}{2}\ [\ ,\ )\ \frac{-t}{2}, \frac{t}{2}\ ) \\ 0 & \text{else.} \end{cases} \tag{5.6}$$

$T_{k,l}$ is an operator that translates the origin of the box aperture by $(-k, -l)$ to the optical axis of the hologram. This yields a common image center in the reconstructions. This operation will however introduce a phase shift in the holographic data, which in turn will act as a translation of the reconstructed object. This is counteracted by the exponential in Eq. (5.5).

5.2.1 Parallelized Fresnel transform

The 2D Fourier transform is linearly separable into two orthogonal 1D Fourier transforms. As depicted in Fig. 5.2, the algorithm consists of three stages. Each stage must be fully completed before the next stage can proceed. In stage 1, a 1D FFT is performed on each row of the hologram. In stage 2, a 1D FFT is performed on each column, the quadratic phase factor of Eq. (5.3)

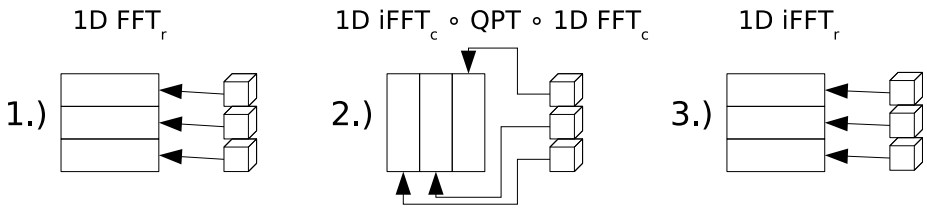

Figure 5.2: Three stage parallelized reconstruction algorithm based on Eq. (5.3). The text shows the computations performed, where QPT denotes multiplication by the quadratic phase term of Eq. (5.3). The cubes represent processors operating on rows or columns individually.

is applied, and a 1D inverse FFT is performed on each column. Finally, in stage 3, an inverse 1D FFT on each row reveals the reconstructed hologram.

5.3 Limits on holographic parallization

In this section, we discuss how our parallel algorithm is affected by the available computer resources. An introduction and general discussion on parallel computing can be found in [8]. In practical terms, parallelizing Eq. (5.3) is constrained by a number of factors which limit the efficiency of the parallization achievable speedup. These limits are: 1) available memory, 2) available processing resources, and 3) a finite communications channel.

The limits on the applicability of parallization of this algorithm are: the size of the reconstruction, the memory requirement of the reconstruction, the number of processors, the granularity of parallization and the rate of transmission of data. We will define these limits, to allow for a more efficient implementation and execution of the algorithm. We will only consider sets of homogeneous processors in this analysis.

5.3.1 Granularity of parallization

The coarsest granularity of parallization is achieved by using a hologram row or column as an atomic unit and grouping these together to achieve the desired granularity. In the following section we assume a reconstruction with dimensions $N \times N$ pixels, and we have a parallel computing system consisting of P processors. The maximum degree of parallization is N, thus the maximum possible speedup of the parallelized reconstruction will be achieved by using $P = N$ processors. Ignoring communication time (assuming instantaneous communication), there would be no idle time, resulting in 100% processor efficiency.

From Fig. 5.2, the amount of data to be transmitted is $6N^2$ pixels where there is N^2 pixels transmitted in each direction at each of the three stages of the computation. When $P < N$ the optimal number of rows to group together is $\lceil N/P \rceil$. This, however, may not be realistically achievable due to memory limitations, necessitating smaller row groups.

Breaking down the algorithm further where each hologram row is parallelized requires breaking up the FFT computation. This results in increased communication costs of $6N^2 \log_2 N$ pixels where there is an additional $\log_2 N$ transmission overhead from additional intermediate calculations. Most parts of the parallelized FFT are dependent on previously calculated data and the maximum degree of parallization is $N^2/2$. If there is instantaneous communication, only N processors would be processing 100% of the time, with all other processors lying idle for significant periods of time. The efficiency at

the maximum degree of parallelisation ($P = N^2/2$), is calculated as

$$\xi_{max} = \log_2 N/(N-1)\,. \tag{5.7}$$

There is no reduction in execution time when $P > N^2/2$. For example, if $N = 32$ and $P = 512$, then the processor efficiency is 16% using Eq. (5.7).

These upper bounds on speedup are not realistically achievable because communication costs are never instantaneous and it is rarely feasible to use large amounts of computing resources inefficiently. Thus we recommend that, firstly, whole rows and whole columns are used as the smallest granularity of parallization to minimize communication costs and increase processor efficiency, and secondly, no more than $P = N$ processors be employed (achieving simultaneously high speedup and high efficiency).

5.3.2 Reconstruction size

For a given set of processing and communication resources, we find the bound on hologram size for which a reconstruction takes the same length of time on a parallel system as it does on a single machine. We assume a dedicated client-server architecture with a single shared communications channel, with 100% processor efficiency and the granularity of parallelization is at the row level.

A holographic pixel can be represented in many different formats, so we abstract away from implementation specific details, viewing a pixel as a single unit. The number of holographic pixels to transmit is $6N^2$. The speed of the communications channel, B, is defined in terms of the number of holo-

gram pixels transmitted per second, which is independent of implementation specific issues, such as compression. Thus, the total transmission time is $6N^2/B$.

The algorithm requires $mN^2 + fN^2 \log_2 N + N$ calculations, where m is the number of calculations to generate the quadratic phase term of Eq. (5.3) and f is a constant factor based on the implementation of the FFT. The point at which a parallel system takes the same length of time as a single machine is thus when $mN^2 + fN^2 \log_2 N + N = \{(mN^2 + fN^2 \log_2 N + N)/P\} + 6N^2/B$ is satisfied, where P is the number of processors used. Ignoring constants, this relationship has the simplified interpretation of

$$N = 2^{1/P + 1/B}, \tag{5.8}$$

which describes the dependency of the minimum reconstruction size required to benefit from parallization. As B or P increases, N asymptotically decreases towards a lower bound. This equation can be reworked to calculate other parameters such as the minimum transmission rate and the minimum number of processors. Speedup, S, can be calculated by setting $S = (mN^2 + 8fN^2 \log_2 N)/[(mN^2 + 8fN^2 \log_2 N)/P + 6N^2/B]$, which is an upper bound on the speedup possible for a given setup and problem instance. As the size of the reconstruction required increases, so does the efficiency of the resource utilization.

5.4 Experimental results

We have implemented the reconstruction algorithm on a Java based distributed system, which uses the spare clock cycles of idle PCs in a university teaching laboratory [50]. Based on this implementation we have evaluated the parallelism of our method as well as the efficiency of memory constraints as described below.

In order to test the performance for large holograms we reconstructed a $2^{16} \times 2^{16}$ (4.3 gigapixel) digital hologram. This is, to our knowledge, the largest ever digital hologram reconstructed. Since no real digital holograms of this size have been captured, we padded out a 2032×2048 hologram in the hologram plane. We used a computer with a single 2.2 GHz Xeon processor and 1 GB of memory, running GNU/Linux. The total reconstruction time on our system was 30 hours. In Fig. 5.3(a) a zoomed in view of the centre of the reconstruction plane can be seen. Figure 5.3(b) shows the relative size of the object within the full field. Next we will show the benefit of using multiple processors to reduce the total execution time.

5.4.1 Distributed reconstruction time

Using multiple processors can decrease the total reconstruction time of a large digital hologram reconstruction. Figure 5.4 shows the reduction in reconstruction time achieved when using multiple processors. We used a homogeneous set of 26 desktop PCs running GNU/Linux, each with 2.0 GHz Intel Processors, 1 GB of memory and connected by a non-dedicated 100 Mb/s Ethernet network. The total processing time is reduced by utilizing more

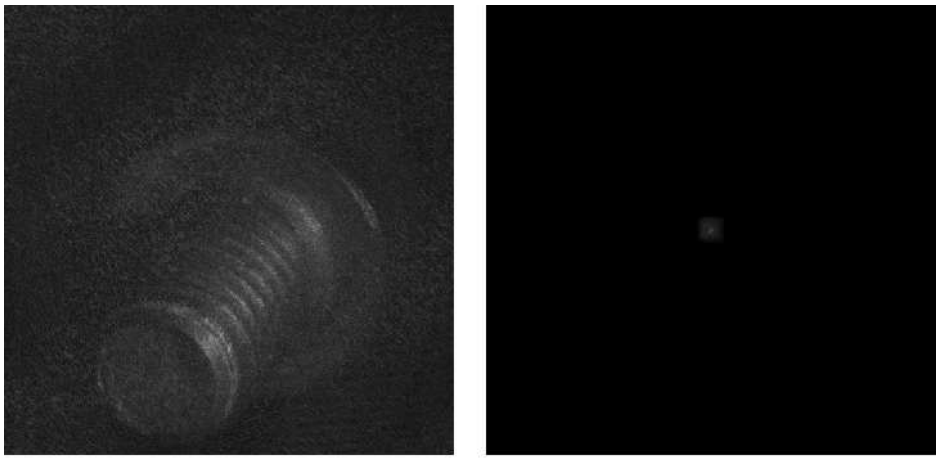

Figure 5.3: Animation of the $2^{16} \times 2^{16}$ reconstruction, from (a) zoomed-in view to (b) full field (MPEG2 - doi:10.1364/OE.16.001990).

processors, however only to a certain point, which varies depending on the computation rate of the processors, the speedup of the network and the size of the reconstruction. In Fig. 5.4 we see large reductions in the reconstruction time when up to 8 processors are added. After that there is no benefit gained by using more processors (as explained in Sect. 5.3), because the network connection is being fully utilized by the system.

5.4.2 Low memory reconstruction

Reconstruction computations have a large space requirement. As the size of a reconstruction increases, it quickly becomes infeasible to process the whole reconstruction in memory on a standard PC. We utilize the high capacity of low cost commodity hard disks in lieu of increased memory. We only keep the portion of the reconstruction which is currently being processed in memory,

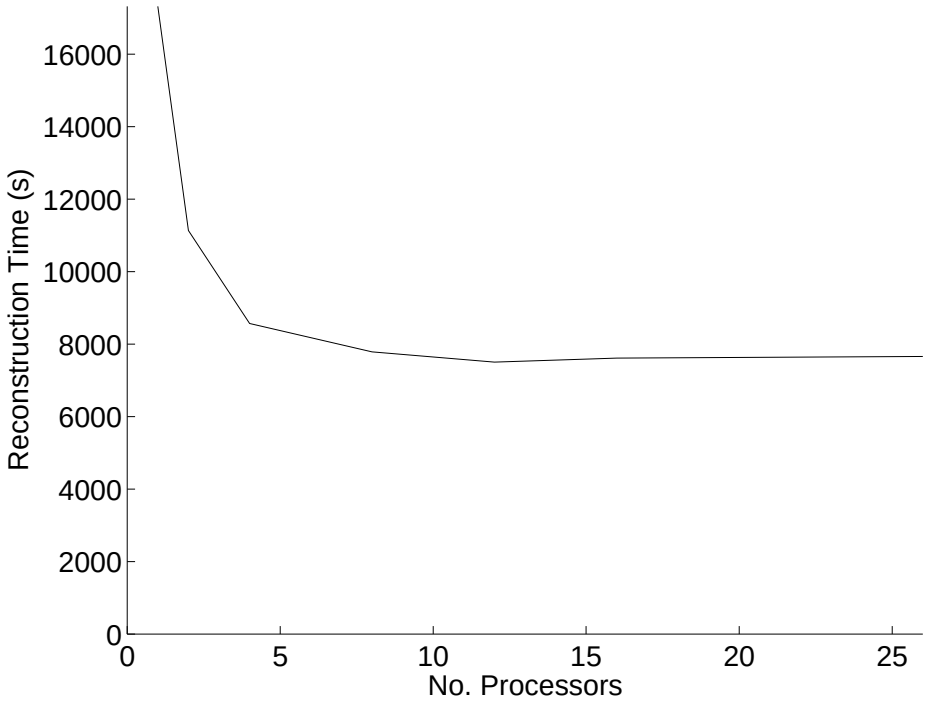

Figure 5.4: Reconstruction time for $2^{14} \times 2^{14}$ digital hologram using varying numbers of processors.

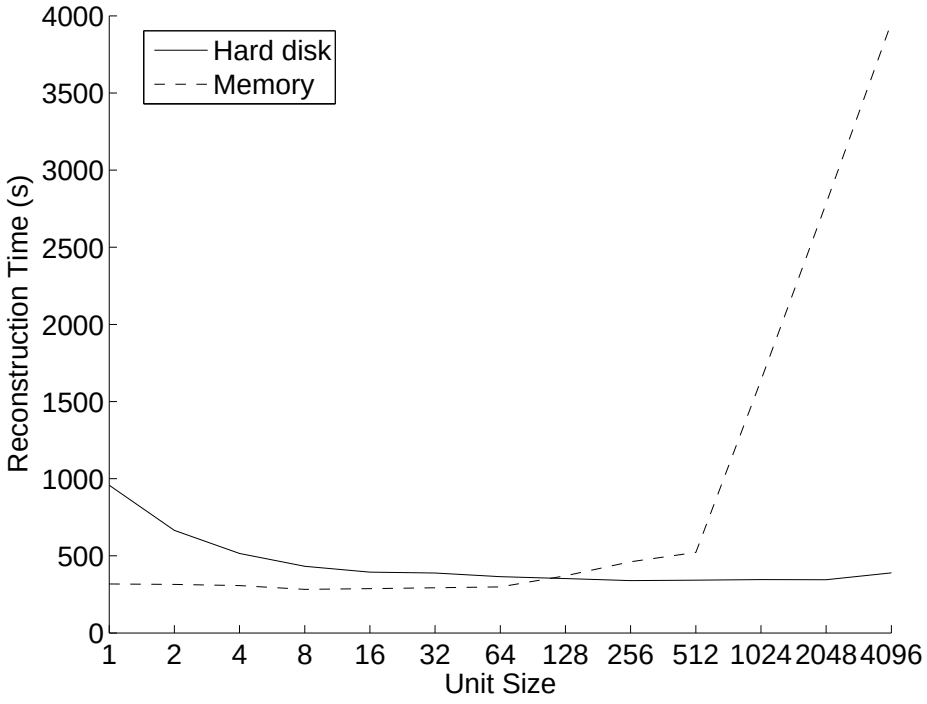

Figure 5.5: Execution time with varying sized units for a $2^{12} \times 2^{12}$ reconstruction, using the hard disk as intermediate storage versus keeping the whole computation in memory.

with the rest of the data stored to hard disk. There is an additional overhead to read and write data on a hard disk, but since we have advanced knowledge of how the data needs to be accessed, we can minimize this factor. The overhead varies depending on the hardware and the amount of data stored in memory at any one time.

If more memory is available, it is more efficient to read in and process multiple rows at once, with this grouping of rows referred to as a unit of work. Figure 5.5 shows the variation in processing times with different unit sizes when reconstructing a $2^{12} \times 2^{12}$ hologram, using a single 2.2 GHz Xeon processor with 2 GB of memory. We performed the reconstructions with and without (i.e. using memory only) using the hard disk as intermediate storage. Using the hard disk, we found that increasing the unit size reduces the reconstruction time by more efficiently reading and writing to the hard disk. Keeping the whole computation in memory provides a consistant reconstruction time, which is faster than using the hard disk as intermediate storage, but only up to a point. When the available physical memory is expended, non-optimized hard disk based swap space is used by the operating system, increasing the reconstruction time eight-fold.

With a unit size of one row, the hard disk based reconstruction algorithm requires 16 MB of memory compared to 800 MB when storing the whole hologram in memory. This memory limitation prevents the memory-only based algorithm from working at all for large holograms. However, with the hard disk based reconstruction algorithm, simply choosing a unit size which falls within the available memory of the machine makes it possible to reconstruct

very large holograms on standard commodity hardware.

5.4.3 Comparison to other implementations

We have compared the algorithm presented in this chapter to a number of other reconstruction methods A graphics processing unit (GPU) based reconstruction implementation has been developed [3]. It exploits the fast and highly parallel architecture of the GPU to reconstruct digital hologram views at video frame rate speeds. Two different GPU methods were developed, one which processes the whole 2D reconstruction in one step, and another which breaks up the problem into 1D FFTs as shown in Fig. 5.2, which has a lower memory requirement. Two other implementations which perform reconstructions using the CPU were written for comparison purposes. The first was written for Matlab, a widely used maths interpreter, which uses highly optimized algorithms for numerical processing. The second was written in C++ and utilizes the fast FFTW library [28] (v. 3.1.2, single threaded).

Table 5.1 show a comparison table of average reconstruction times for both the direct and convolution approaches and hologram sizes up to 4096 $\times$ 4096. The results for the GPU implementation were run on a Linux PC with 2 Gigabytes system RAM, AMD Athlon Dual Core 64 bit processor and a GeForce 8800GTX graphics card with 768 Mbytes on board RAM. The Matlab implementation was executed on a dedicated server equipped with a Dual Core Xeon 1.6GHz CPU and 4 Gigabytes of RAM. The native C++ based method was executed on an AMD Athlon 64 X2, 2.3 GHz equipped with 2 GB of RAM. The algorithm described in this chapter was run on

	Reconstruction Time (ms)					
	GPU		CPU		Java	
Resolution	2D	1D	C++	Matlab	RAM	Hard Disk
512×512	2	3	251	502	6219	21046
1024×1204	7	19	1060	1845	25360	77856
2048×2048	47	106	4550	7438	104462	311276
4096×4096	204	598	23530	29731	367838	1179478

Table 5.1: Reconstruction times in milliseconds for the convolution method using 6 different methods. The GPU and CPU times were averaged over 1000 runs [3].

the same dedicated server, using 1 CPU. One approach used RAM as the intermediate storage mechanism, and the other approach used the hard disk as the intermediate storage mechanism. The 2D GPU method out performs the C++, Matlab and Java implementations by between 125 and 10523 times for a 512×512 reconstruction. For a reconstruction of size 4096×4096 it out performs them by between 115 and 5781 times. All of the methods, with the exception of the Java hard disk based implementation, quickly reach the limits of the available RAM, making larger reconstructions impossible.

The Java based method is unsuitable for small reconstructions, but by using the hard disk as intermediate storage, it can reconstruct very large holograms, which is not possible with the other reconstruction methods. A significant limiting factor is that the speed of data transfer with hard disks is very slow compared to RAM.

5.5 Conclusions

We created a parallel Fresnel hologram reconstruction method that can reconstruct large holograms on standard desktop PCs. We have shown how it is possible to reduce the reconstruction time for large holograms by using a distributed system. The method also has a small memory footprint, allowing for the possibility of performing holographic reconstructions on resource constrained devices. This could open up possibilities for shared distributed computing on, for example, mobile devices in the future.

In our future work we will look at the effect of using a heterogeneous web computing system for reconstructions, as well as robust parallel reconstructions with quality of service guarantees for holographic video. Other interesting possibilities include implementing other computationally expensive methods in digital holography, for example advanced speckle reduction and hologram image processing techniques.

Chapter 6

Conclusion

In this thesis we presented two scheduling methods for allocating tasks to processors in an unreliable heterogeneous distributed system. No restrictive assumptions are made about: the heterogeneity of the processing or communication resources, the availability of the resources, the architecture of the communications network, the task execution time distributions of the problems to be processed, or the communication distribution requirements of the problems to be processed. They begin with zero initial knowledge about the state of the system or the problems to be processed and generate all required information on-line. It dynamically adapts to any set of processing and communication resources, whilst utilizing these resources as efficiently as possible. We showed that they can efficiently allocate tasks to processors on a real-world heterogeneous distributed system. Next we will summarizes the work presented in this thesis.

6.1 Dynamically estimating properties

Given an unreliable, non-dedicated set of processing and communication resources, a scheduler is required to allocate tasks to processors. No information about the state of the system, which can vary over time, or the tasks to be processed is known in advance, and thus must be estimated dynamically. A property estimation method is presented in Chapter 2, which utilizes a kNN algorithm, a smoothed average and an analytical benchmark. These estimated properties are then used by two different scheduling techniques, which make less restrictive assumptions than the current state-of-the-art methods.

For future work, it is envisaged that the next step will be to investigate different methods for weighting the different inputs to the kNN or possibly to replace the whole estimation algorithm with a more advanced learning technique such as a neural network.

6.2 Task allocation using GAs

A multi-heuristic evolutionary scheduling algorithm is presented in Chapter 3. Multiple simple heuristics are combined to enhance a genetic algorithm to schedule tasks in a fast and efficient manner. The evolutionary nature of a GA allows for near optimal solutions to be found quickly, even in a dynamic distributed system with constantly changing resources. Real-world experiments using up to 150 heterogeneous processors have shown that the algorithm achieves better efficiency than other state-of-the-art heuristic algorithms.

The novelty in this work is with the lack of restrictive assumptions, that other schedulers make. This scheduler can be used with any set of processing or communication resources, to process any problems. It requires no inputs from the user, and can be treated as a black box, yet it will model the state of the system and the task execution time distributions online and generate efficient scheduling solutions. The evolutionary nature of the hybrid GA scheduler allows it to adapt to any set of input parameters allowing for very efficient solutions to be found within a short space of time. This makes it more generalizable than the current state-of-the-art schedulers for distributed computing.

The next step in this work will be to parallelize the algorithm across the distributed system, to make best use of any idle resources. This chapter sets out a framework which can easily be used with other evolutionary algorithms, and scheduling heuristics.

6.3 Task allocation using estimation error

A deterministic method utilizes the error inherent in estimating the properties of the system and the execution time of tasks to allocate tasks to processors in Chapter 4. It is predictable and can be verified, due to the absense of randomness. This makes it more suitable for certain applications where predictability is required, such as real-time applications or medical applications. It uses the error inherent in estimating properties as inputs to a scheduler. A family of schedulers are presented which use this method, each seeking to maximize or minimize the uncertainty of the values of the prop-

erties (such as task execution time) when making task allocations. These are then combined with a number of objectives, minimizing makespan, load balancing, and matching the properties of tasks to suitable processors. It has been evaluated on a number of non-dedicated real-world heterogeneous distributed system configurations. The efficiency achieved is nearly the same performance as more complicated evolutionary scheduler, but without the complexity of an evolutionary algorithm.

The main contributions that this work makes is with the lack of restrictive assumptions it makes, and its ability to adapt to the variability of a real-world distributed system. This work is the first to utilize the inherent error in estimating the properties of a distributed system and to use it to enhance the allocation of tasks to processors. It can achieve nearly the same performance as a more complicated meta-heuristic. It has been shown to work very effectively on a real-world distributed system.

A preliminary investigation has been done on combining the multiple different objectives, from this chapter, to further improve the overall efficiency of the scheduler. Early results indicate this strategy holds much promise as the next step in this research.

6.4 Distributed Applications

The scheduling research presented in this thesis is intended to more efficiently utilizing the resources of a heterogeneous distributed system to reduce the overall execution time of computationally intensive problems. A computationally intensive problem has been parallelized and is described in Chap-

ter 5. It is a low memory distributed reconstruction application for large digital holograms [3, 76].

A parallel implementation of the Fresnel transform, suitable for reconstructing large digital holograms, has been developed. It has a small memory footprint and utilizes the spare resources of a heterogeneous distributed system to reduce the overall execution time. We demonstrated how a 4.3 gigapixel digital hologram can be reconstructed, which is 16 times larger than previously reconstructed holograms, in the literature. We also show the effect of different memory and processor configurations on the efficiency of the algorithm.

The digital holography research presented in Chapter 5 feeds into a longer term goal of 3-dimensional television. We focused on the open problems with holographic view reconstruction: size of reconstruction, speed of reconstruction, and memory footprint required. This work has since fed into 3 different on-going research topics: 1.) reconstruction using commodity graphics cards [3], 2.) reconstruction on resource constrained mobile devices, and 3.) distributed reconstruction of streaming video [77].

6.5 Final words

The initial impetus for this research was to allow for a non-technical user (such as a biologist) to setup a distributed system with any available processing resources at their disposal, so that they could efficiently process their computationally intensive problems, whilst treating the internal workings of the distributed system as a black box. We have addressed this problem by

creating two methods complementary scheduling methods, and shown their effectiveness in a real-world distributed system, efficiently processing computationally intensive problems from a number of different fields. This is still an open problem, but the methods presented in this thesis go some way to advancing the area.

Appendix A

Taxonomy

Paper	Static scheduling	Homogeneous resources	Non pre-emptive	Task sizes known	Comms costs known	Platform dependent	Dedicated resources	Year
[6]	-		-	x	x			2003
[22]	-		-	x	x			-
[101]	-		-	x	x			2003
[52]		x	x	x	x			2001
[54]	-	x	x	x	x			2002
[91]	-	x	-	x	x			2003
[74]			x					2003

Table A.1: Taxonomy of scheduling within web computing platforms. Dash (-) indicates unknown or inapplicable.

Paper	Static scheduling	Homogeneous resources	Non pre-emptive	Task sizes known	Comms costs known	Platform dependent	Dedicated resources	Year
[38]	x	x	x	x	x		x	1994
[90]	x		x	x	x		x	1996
[32]	x		x	x	x		x	1999
[96]	x	x	x	x	x			2006
[16]	x	x	x	x	x		x	1999
[59]	x		x	x	x	x	x	2008
[98]	x		x	x	x		x	2001
[21]	x		x	x	x		x	2002
[1]	x		x	x	x		x	2001
[60]	x		x	x	x			2002
[62]	x		x	x	x			2003
[92]	x		x	x	x			2006
[10]	x		x	x	x		x	2001
[102]	x		x	x	x		x	1997
[108]		x	x	x	x		x	2001
[107]		x	x	x	x		x	2001
[105]		x	x	x			x	1998
[33]			x	x	x		x	2001
[81]			x	x				2004
Chap. 3			x					2008

Table A.2: Taxonomy of evolutionary schedulers. Dash (-) indicates unknown or inapplicable.

Paper	Static scheduling	Homogeneous resources	Non pre-emptive	Task sizes known	Comms costs known	Platform dependent	Dedicated resources	Year
[55]	x	x	x	x	x	x	x	1996
[7]	x		x	x	x		x	2004
[24]	x		x	x	x		x	2002
[34]		x	x	x	x	x	x	2000
[66]		x	x		x	x	x	1997
[45]		x	x	x	x	x	x	1984
[61]			x	x	x		x	1999
[68]						x		1999
[26]						x		1997
[80]			x					2008

Table A.3: Taxonomy of non-evolutionary schedulers. Dash (-) indicates unknown or inapplicable.

Symbol	Description	Location
X_i	Task input parameters	Sect. 2.1
x_2^{i}	A single input parameter	Sect. 2.1
q	No. input params	Sect. 2.1
t_i	Actual processing time of task	Sect. 2.1
j	Processor index	
i	Task index	
ϵ	Task processing time not estimated	Sect. 2.1
O	Set of observations	Eq. (2.2)
n	No. prev task execution times	Sect. 2.1
k	k nearest neighbours	Sect. 2.1.1
U_j	Set of task result input parameters	
L	% of obervationss deselected	Alg. 2.1 & Fig. 2.1
ETC	Estimated time to compute	Eq. (2.6) & Eq. (2.8)
d	Euclidean distance	Eq. (2.4)
y	No. of obervations used in kNN	Eq. (2.3)
f	Index variable	
w	Weighting value in kNN	Eq. (2.5)
P_j	Computational rate in MFLOP/s	Sect. 2.1.2
Γ	Smoothing function	Sect. 2.1.2
T_i	Estimated task MFLOPs	Sect. 2.1.2
M	No. of processors	Sect. 4.2
N	Total No. of Tasks to schedule	
TIME	Start of scheduling time	Sect. 4.2
ET(i,j)	Task error value	Eq. (4.2)
C(i,j)	Estimated communications cost	Sect. 4.2.1
CCR(i,j)	Computation to Communication Ratio	Eq. (4.3)
EW(i,j)	Error weight	Eq. (4.4)
ST()	Next idle time of processor	Eq. (4.6)
Q_j	Queue of scheduled tasks	Alg. 4.1

Table A.4: Taxonomy of variables used

Symbol	Description	Location
c_i	communications overhead	Sect. 2.1
α, β	Control variables	Sect. 4.2
PS(i,j)	Task processor suitability	Eq. (4.14)
CS(i,j)	Communications suitability	Eq. (4.15)
A_i	Size of task in bytes	Sect. 4.2.5
B_j	Bandwidth to processor j	Sect. 4.2.5
ξ_{max}	Efficiency at maximum degree of parallization	Eq. (5.7)

Table A.5: Taxonomy of variables used

Appendix B

Task Allocation Problem

We wish to map tasks to processors in a dynamic heterogeneous distributed system. The TA problem is NP-complete in the general case [100]. However, the general TA problem does not accurately model a real-world dynamic distributed system [95]. We will show that if we introduce a small amount of unknown dynamism, then the problem is not contained in NP. Thus we cannot linearly transform efficient solutions from problems contained in NP and must create new heuristics.

B.1 Task allocation problem

In this section we will explore the TA problem and its complexity. We begin with the standard TA problem, which we know is in NP [100]. We then introduce a dynamic element to the problem. We will show that if the amount of dynamism is known, the problem is in NP, however, if the amount of dynamism is unknown, the problem is not contained in NP. If the problem

is in NP, solutions from existing efficient algorithms, such as TSP [15] and 3SAT, can be linearly transformed to solve the TA problem. However this is not possible if the problem is not contained in NP, assuming NP $\neq$ PSPACE.

B.1.1 TA problem is in NP

The problem is defined as follows: given a set of task identifiers $T = \{0, 1, \ldots, N-1\}$, a set of processor identifiers $P = \{0, 1, \ldots, M-1\}$, a matrix $A : N \times M \mapsto \mathbb{N}$ which contains the execution time in seconds of each task on each processor, and a value $k \in \mathbb{N}$, is there a mapping of tasks to processors which has a makespan (total execution time) of less than or equal to k seconds? Note this definition allows for heterogeneous tasks and processors.

We will first prove that the TA problem is in NP. We do this by proving that each instance of the problem can be verified in polynomial time. Given an instance of the problem, and the solution "yes", we define an algorithm that verifies this solution in all cases. The witness (or certificate) we choose is the mapping $S : T \mapsto P$ from tasks to processors. Our algorithm is shown in Alg. B.1.

Correctness analysis

The cost of each task is added to an accumulator for each processor. After all tasks have been added each accumulator is checked to see that its value is not greater than k.

```
   Input: T, P, A, k, S, N, M
   Output: Yes or No
 1 P : M ↦ ℝ;
 2 P = 0;
 3 foreach Task i = 0..(N − 1) do
 4 |  P[S[i]]+ = A[S[i]][i];
 5
 6 foreach Processor j = 0..(M − 1) do
 7 |  if P[j] > k then
 8 |  |  return No;
 9 |
10 return Yes;
```

Algorithm B.1: Algorithm to verify a given solution in polynomial time.

Complexity analysis

The first `FOR` loop requires N iterations, each of which consists of one addition operation. The second `FOR` loop requires at most M iterations, each of which requires one comparison. The total complexity is $\theta(N + M)$, which means it is a polynomial algorithm. Therefore if the TA problem can be verified in polynomial time then it is in NP.

B.1.2 TA problem with dynamism

We will show in this subsection that if a known amount of dynamism is introduced to the TA problem a solution can be verified in polynomial time, but when the amount of dynamism is unknown the solution cannot be verified in polynomial time. At time step t, δ changes occur in matrix A. These changed elements could change to any $\mathbb{R}$ but for simplicity we limit them to a binary set, they either change or they stay the same.

The problem inputs are as follows: a constant value δ, a value $k \in \mathbb{N}$, and the inputs defined in Sect. B.1.1. Is there a mapping of tasks to processors which has a makespan of less than or equal to k seconds after time step t?

The problem definition has two additions. We will now assume that at some point in time the matrix A changes to matrix A', but we restrict the values to simplify the problem. δ corresponds to the number of elements that change at time step t, controlling the dynamism. At time step t all elements in A are copied to A'. During this copy, δ elements of A' are doubled in value. Simplification by limiting the numbers allows for the use of a binary mask, however this could easily be substituted for a mask of different numbers, thus retaining the generality of the method.

In the verification algorithm, to generate matrix A' we create a binary mask $Z \in \{1, 2\}^{N \times M}$, and multiply every element of A by the corresponding mask element to generate the A'. Thus some tasks will take twice as long to process while others will remain unchanged. Of course, as we don't know which Z is correct we must try each possibility to verify the solution to this problem. Our algorithm is presented in Alg. B.2.

Correctness analysis

First we generate all possible combinations of A and the mask to produce 2^N matrices of A' using a binary shifting algorithm. As in Alg. B.1 the cost of each task is added to an accumulator for each processor, with a separate set of accumulators for each instance of A'. Before time t all task execution times are calculated using A, and after time t all task execution times are

```
   Input: T, P, A, t, k, S, N, M
   Output: Yes or No
 1 P : M ↦ ℝ;
 2 A′ : M × N ↦ ℝ;
 3 dispMask ↦ ℕ;
 4 X = 0, 1;
 5 P, A′, X, dispMask = 0;
 6 foreach r = 0..(2^δ − 1) do
 7     foreach f = 0..(N − 1) do
 8         d = r ;
 9         if (d&dispMask) == dispMask then
10             A′[S[f]][f] = 2 × A[S[f]][f];
11         else
12             A′[S[f]][f] = A[S[f]][f];
13         d ≪= 1;
14     foreach Task i = 0..N − 1 do
15         if P[S[i]] < t then
16             if P[S[i]] + A[S[i]][i] > t then
17                 v = (t − P[S[i]])/A[S[i]][i];
18                 P[S[i]]+ = v × A[S[i]][i];
19                 P[S[i]]+ = (1 − v) × A′[S[i]][i];
20             else
21                 P[S[i]]+ = A[S[i]][i];
22
23         else
24             P[S[i]]+ = A′[S[i]][i];
25
26     X = 0;
27     foreach Processor j = 0..(M − 1) do
28         if P[j] > k then
29             X = 1;
30
31     if X == 0 then
32         return Yes;
33
34
35 return No;
```

Algorithm B.2: Algorithm to check a schedule where the task execution times change at a given time t and A' is dynamically created.

calculated using each instance of A'. After all tasks have been added, each of the accumulators are checked to see that its value is not greater than k.

Complexity analysis

The first `FOR` loop requires 2^{δ} iterations.. The second `FOR` loop requires N iterations, each of which consists of four operations. The third `FOR` loop requires at N iterations. For tasks that are executing at time t, three operations are required, and this occurs at most $M \times 2^N$ times. The fourth `FOR` loop requires at M iterations with one comparison.

The total complexity is $O((N + M)2^{\delta})$, so when δ is a constant value the solution can be verified in polynomial time, but when δ is unknown it is $\delta = N$. Thus with a complexity of $O((N + M)2^N)$ the problem is not polynomial time verifiable and is thus not contained in NP (in this instance it is in PSPACE, assuming NP $\neq$ PSPACE).

Discussion

Different masks can give radically different makespans. Every possible permutation of Z must be checked to see if it gives rise to a makespan of less than or equal to k. When dynamism introduces a known number of changes, the problem is in NP, but when the number of changes is unknown, the problem is not contained in NP. If δ is unknown, it will always require all possible masks to be checked to verify a solution, thus it is not contained in NP. It is thus not possible to linearly transform any NP-complete solvers to our dynamic scheduling problem. The DTA problem tackled in this thesis

contains multiple dynamic elements. Adapting Alg. B.2 to verify a problem with these elements would involve additional masks, which still place the problem outside NP if an unknown number of changes occur. Subramani has has also shown this to be the case in [95].

The problem which we tackle is defined as follows: given a set of task identifiers $T = \{0, 1, \ldots, N-1\}$, a set of processor identifiers $P = \{0, 1, \ldots, M-1\}$, a set of communication link identifiers $C = \{0, 1, \ldots, M-1\}$, a matrix $A : N \times M \mapsto \mathbb{R}$ which contains the execution time in seconds of each task on each processor, a matrix $D : N \times M \mapsto \mathbb{R}$ which contains the communication time in seconds of transmitting a task to a processor using a communication link, and a value $k \in \mathbb{N}$. Is there a mapping of tasks to processors which has a makespan of less than or equal to k seconds? The matrix A transforms to a new matrix using a mask Z at each time step, and the values of the mask are $\in \mathbb{R}$.

Bibliography

[1] I. Ahmad, Y.-K. Kwok, I. Ahmad, and M. Dhodhi. Scheduling parallel programs using genetic algorithms. In A. Y. Zomaya, F. Ercal, and S. Olariu, editors, *Solutions to Parallel and Distributed Computing Problems*, chapter 9, pages 231–254. John Wiley and Sons, New York, USA, 2001.

[2] L. Ahrenberg, P. Benzie, M. Magnor, and J. Watson. Computer generated holography using parallel commodity graphics hardware. *Opt. Express*, 14:7636–7641, 2006.

[3] L. Ahrenberg, A. J. Page, B. Hennelly, J. McDonald, and T. J. Naughton. Using commodity graphics hardware for real-time digital hologram view reconstruction. *IEEE/OSA Journal of Display Technology*, In Press 2008.

[4] E. Alba, A. J. Nebro, and J. M. Troya. Heterogeneous computer and parallel genetic algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362–1385, September 2002.

[5] S. Ali, A. A. Maciejewski, H. J. Siegel, and J.-K. Kim. Measuring the robustness of a resource allocation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):630–641, July 2004.

[6] D. Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, pages 17–19, Madrid, Spain, November 2003.

[7] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, February 2004.

[8] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[9] H. Bodlaender, M. Fellows, and T. Warnow. *Two strikes against perfect phylogeny*, volume 623, pages 273–283. Springer-Verlag, NY, USA, 1992.

[10] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel Distribed Computing*, 61(6):810–837, 2001.

[11] R. J. Carroll, D. Ruppert, and L. A. Stefanski. *Measurement error in nonlinear models*. Chapman & Hall, Boca Raton, 1998.

[12] A. Chipperfield and P. Flemming. Parallel genetic algorithms. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1118–1143. McGraw-Hill, New York, USA, first edition, 1996.

[13] J. Cohen, E. Jeannot, N. Padoy, and F. Wagner. Messages scheduling for parallel data redistribution between clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1163–1175, Oct 2006.

[14] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the First European Conference on Artificial Life*, pages 134–142, Paris, France, 1992. Elsevier.

[15] CONCORDE TSP Solver. http://www.tsp.gatech.edu/concorde, 2008.

[16] R. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, August 1999.

[17] B. V. Dasarathy. *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6:182–197, 2002.

[19] F. Dehne, editor. *Special issue on Coarse Grained Parallel Algorithms for Scientific Applications*, volume 45. Springer, New York, July 2006.

[20] L. P. Devroye. The uniform convergence of nearest neighbour regression function estimators and their application in optimization. *IEEE Transactions on Information Theory*, 24:142–151, March 1978.

[21] M. K. Dhodhi, I. Ahmad, A. Yatama, and I. Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 62:1338–1361, September 2002.

[22] Distributed.net. http://www.distributed.net.

[23] A. Dogan and F. Ozguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):308–323, 2002.

[24] A. Dogan and F. Ozguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):308–323, March 2002.

[25] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, USA, 1979.

[26] X. Du and X. Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1997.

[27] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[28] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[29] H. Gautama and A. van Gemund. Low-cost static performance prediction of parallel stochastic task compositions. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):78–91, Jan 2006.

[30] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, May 1986.

[31] J. W. Goodman. *Introduction to Fourier Optics*. Roberts and Company, 3rd edition, 2005.

[32] M. Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 280–285, New Orleans, Louisiana, USA, 1999. ACM Press.

[33] W. A. Greene. Dynamic load-balancing via a genetic algorithm. In *13th IEEE International Conference on Tools with Artificial Intelligence*, pages 121–129, Dallas, Texas, USA, November 2001.

[34] B. Hamidzadeh, L. Y. Kit, and D. Lilja. Dynamic task scheduling using online optimization. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1151–1163, November 2000.

[35] W. Härdle. *Applied Nonparametric regression.* Cambridge University Press, 1990.

[36] B. Hennelly and J. Sheridan. Fast numerical algorithm for the linear canonical transform. *Journal of the Opt. Soc. of Am.*, 22:928–937, 2005.

[37] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, USA, 1975.

[38] E. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, February 1994.

[39] HPC Challenge. http://icl.cs.utk.edu/hpcc, 2005.

[40] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[41] T. Ito, N. Masuda, K. Yoshimura, A. Shiraki, T. Shimobaba, and T. Sugie. Special-purpose computer HORN-5 for a real-time electroholography. *Opt. Express*, 13:1923–1932, 2005.

[42] M. A. Iverson, F. Ozguner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling

in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, December 1999.

[43] Jannealer. http://jannealer.sourceforge.net, 2006.

[44] T. Janse, V. von Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier volume rendering on the GPU using a split-stream-FFT. In *Proc. of the VMV'04, Stanford, CA*, pages 395–403. IOS Press BV, 16-18 November 2004.

[45] H. Kasahara and S. Narita. Practical multiprocessing scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11):1023–1029, November 1984.

[46] T. Keane, R. Allen, T. J. Naughton, J. McInerney, and J. Waldron. Distributed Java platform with programmable MIMD capabilities. In N. Guelfi, E. Astesiano, and G. Reggio, editors, *Scientific Engineering for Distributed Java Applications*, volume 2604, pages 122–131. Springer Lecture Notes in Computer Science, February 2003.

[47] T. M. Keane and T. J. Naughton. DSEARCH: sensitive database searching using distributed computing. *Bioinformatics*, page bti163, 2004.

[48] T. M. Keane, T. J. Naughton, S. A. A. Travers, J. O. McInerney, and G. P. McCormack. DPRml: distributed phylogeny reconstruction by maximum likelihood. *Bioinformatics*, page bti100, 2004.

[49] T. M. Keane, A. J. Page, J. O. McInerney, and T. J. Naughton. A high-throughput bioinformatics distributed computing platform. In *Bioinformatics and its Medical Applications Special Track, The 18th IEEE International Symposium on Computer-Based Medical Systems*, pages 377–382, Dublin, Ireland, June 2005.

[50] T. M. Keane, A. J. Page, T. J. Naughton, S. A. Travers, and J. O. McInerney. Building large phylogenetic trees on coarse-grained parallel machines. *Algorithmica, Special issue on Coarse Grained Parallel Algorithms for Scientific Applications*, 45(3):285–300, July 2006.

[51] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[52] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@HOME massively distributed computing for SETI. *Comput. Sci. Eng.*, 3(1):78–83, 2001.

[53] T. Kreis. *Handbook of Holographic Interferometry*. Wiley-VCH, 2005.

[54] E. Krieger and G. Vriend. Models@Home: distributed computing in bioinformatics using a screensaver based approach. *Bioinformatics*, 18(2):315–318, February 2002.

[55] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[56] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999.

[57] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[58] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7(3):141–147, 1988.

[59] Y.-C. Lee and A. Zomaya. A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1215–1223, Sept 2008.

[60] D. Liu, Y. Li, and M. Yu. A genetic algorithm for task scheduling in network computing environment. In *Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 126–129, Beijing, China, October 2002.

[61] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.

[62] V. D. Martino. Sub optimal scheduling in a grid using genetic algorithms. In *International Parallel and Distributed Processing Symposium*, pages 148–155, Nice, France, April 2003.

[63] N. Masuda, T. Ito, K. Kayama, H. Kono, S. Satake, T. Kunugi, and K. Sato. Special purpose computer for digital holographic particle tracking velocimetry. *Opt. Express*, 14:587–592, 2006.

[64] N. Masuda, T. Ito, T. Tanaka, A. Shiraki, and T. Sugie. Computer generated holography using a graphics processing unit. *Opt. Express*, 14:603–608, 2006.

[65] MD5. http://tools.ietf.org/rfc/rfc1321.txt, 1992.

[66] P. Mohapatra. Dynamic real-time tasks scheduling on hypercubes. *Journal of Parallel and Distributed Computing*, 46(1):91–100, 1997.

[67] B. Munjuluri, M. Huebschman, and H.R.Garner. Rapid hologram updates for real-time volumetric information displays. *Applied Optics*, 44:5076–5085, 2005.

[68] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. Alternatives to coscheduling a network of workstations. *Journal of Parallel and Distributed Computing*, 59(2):302–327, November 1999.

[69] T. J. Naughton, Y. Frauel, B. Javidi, and E. Tajahuerce. Compression of digital holograms for three-dimensional object reconstruction and recognition. *Applied Optics*, 41:4124–4132, 2002.

[70] A. J. Nebro, E. Alba, and F. Luna. Multi-objective optimization using grid computing. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 11(6):531–540, 2007.

[71] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham. Abyss: Adapting scatter search to multiobjective optimization. *Evolutionary Computation, IEEE Transactions on*, 2007.

[72] I. M. Oliver, D. J. Smith, and J. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 224–230. Lawrence Erlbaum Associates, Inc., 1987.

[73] OpenTS - Java Tabu Search. http://www.coin-or.org/OpenTS, 2006.

[74] A. Page, T. Keane, R. Allen, T. J. Naughton, and J. Waldron. Multitiered distributed computing platform. In *2nd International Conference on the Principles and Practice of Programming in Java*, pages 191–194, Kilkenny City, Ireland, June 2003.

[75] A. Page, T. Keane, and T. J. Naughton. Adaptive scheduling across a distributed computation platform. In J. P. Morrisson, editor, *Third International Symposium on Parallel and Distributed Computing*, pages 141–149, Cork, Ireland, July 2004. IEEE Computer Society.

[76] A. J. Page, L. Ahrenberg, and T. J. Naughton. Low memory distributed reconstruction of large digital holograms. *Opt. Express*, 16(3):1990–1995, 2008.

[77] A. J. Page, L. Ahrenberg, and T. J. Naughton. Robust distributed digital hologram view reconstruction. In preparation, 2008.

[78] A. J. Page, S. Coyle, T. M. Keane, T. J. Naughton, C. Markham, and T. Ward. Distributed monte carlo simulation of light transportation in tissue. In *proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, pages 1–4, Rhodes, Greece, April 2006. IEEE Computer Society.

[79] A. J. Page, T. M. Keane, and T. J. Naughton. Bioinformatics on a heterogeneous java distributed system. In *Proceedings of the 19th IEEE/ACM International Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 2005. IEEE Computer Society.

[80] A. J. Page, T. M. Keane, and T. J. Naughton. Scheduling in a dynamic heterogeneous distributed system using estimation error. *Journal of Parallel and Distributed Computing*, page doi:10.1016/j.jpdc.2008.07.004, 2008.

[81] A. J. Page and T. J. Naughton. Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. In *15th Artificial Intelligence and Cognitive Science Conference*, pages 137–146, Castlebar, Ireland, September 2004.

[82] A. J. Page and T. J. Naughton. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 189.1, Washington, DC, USA, 2005. IEEE Computer Society.

[83] A. J. Page and T. J. Naughton. Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. *Artificial Intelligence Review*, 24(3-4):415–429, Nov 2005.

[84] M. C. Pease. An adaptation of the fast fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968.

[85] J. M. Pollard. Monte carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[86] M. Reicherter, S. Zwick, T. Haist, C.Kohler, H. Tiziani, and W. Osten. Fast digital hologram generation and adaptive force measurement in liquid-crystal-display-based holographic tweezers. *Applied Optics*, 45:888–896, 2006.

[87] U. Schnars and W. P. Juptner. Direct recording of holograms by a ccd target and numerical reconstruction. *Applied Optics*, 33:179–181, Jan 1994.

[88] B. Schneier. *Applied Cryptography*. John Wiley and Sons, New York, NY, 2nd edition, 1994.

[89] SHA1. http://tools.ietf.org/rfc/rfc3174.txt, 2001.

[90] H. J. Siegel, L. Wang, V. Roychowdhury, and M. Tan. Computing with heterogeneous parallel machines: advantages and challenges. In *Proceedings on Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 368–374, Beijing, China, June 1996.

[91] T. Silvestre, E. Nugues, G. Perrière, M. Gouy, and L. Duret. Phylojava : a generic client-server tool for phylogenetic tree reconstruction - application to grid computing. In M.-F. Sagot and H.-P. Lenhof, editors, *European Conference on Computational Biology*, Paris, France, September 2003.

[92] O. Sinnen, L. Sousa, and F. Sandnes. Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):263–275, March 2006.

[93] M. Sipser. *Introduction to the Theory of Computation*. Thomson, Boston, second edition, 2006.

[94] A. Stuart and J. Ord. *Kendall's advanced theory of statistics,Vol. 1: Distribution theory*. Edward Arnold, London, sixth edition, 1994.

[95] K. Subramani. A Comprehensive Framework for Specifying Clairvoyance, Constraints and Periodicity in Real-Time Scheduling. *The Computer Journal*, 48(3):259–272, 2005.

[96] A. Swiecicka, F. Seredynski, and A. Zomaya. Multiprocessor scheduling and rescheduling with use of cellular automata and artificial immune

system support. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):253–262, March 2006.

[97] E. Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70:809–825, 2001.

[98] M. D. Theys, T. D. Braun, H. J. Siegal, A. A. Maciejewski, and Y.-K. Kwok. *Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach*, chapter 6, pages 135–178. John Wiley and Sons, New York, USA, 2001.

[99] Top 500 Super Computers. http://www.top500.org, 2005.

[100] J. D. Ullman. NP-complete scheduling problems. *J. Computing System Science*, 10:384–393, 1975.

[101] United Devices. *Grid MP Platform Architecture*, 2003. White Paper.

[102] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, November 1997.

[103] M.-Y. Wu and W. Shu. A high-performance mapping algorithm for heterogeneous computing systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, page 6, San Francisco, CA, USA, April 2001.

[104] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Zurich, Switzerland, 2001.

[105] A. Y. Zomaya, M. Clements, and S. Olariu. A framework for reinforcement-based scheduling in parallel processor systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):249–260, March 1998.

[106] A. Y. Zomaya, R. C. Lee, and S. Olariu. An introduction to genetic-based scheduling in parallel processor systems. In A. Y. Zomaya, F. Ercal, and S. Olariu, editors, *Solutions to Parallel and Distributed Computing Problems*, chapter 5, pages 111–133. John Wiley and Sons, New York, USA, 2001.

[107] A. Y. Zomaya and Y.-H. Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):899–911, September 2001.

[108] A. Y. Zomaya, C. Ward, and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795–812, August 1999.