Scheduling in a dynamic heterogeneous distributed system using estimation error

Andrew J. Page [a,*], Thomas M. Keane [b], Thomas J. Naughton [a,c]

[a] *Department of Computer Science, National University of Ireland, Maynooth, Co.Kildare, Ireland*
[b] *Pathogen Sequencing Unit, Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus, Hinxton, CB10 1SA Hinxton, UK*
[c] *University of Oulu, RFMedia Laboratory, Oulu Southern Institute, Vierimaantie 5, 84100 Ylivieska, Finland*

ARTICLE INFO

ABSTRACT

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. 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 on a real-world heterogeneous distributed system with up to 90 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. 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.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

Modern scientific research has ever-increasing computational requirements. Many of the large problems being tackled are ideal candidates for parallelization [6]. 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. Due to the distributed nature of the underlying resources it is necessary to allocate tasks to processors in an efficient manner, otherwise the benefits of using multiple processors is nullified. The multiprocessor scheduling problem is NP-complete in the general case [13,39].

In real-world heterogeneous distributed systems, the allocation of tasks to processors can be an inefficient process. In the absence of an efficient algorithm to find the optimal schedule [13], a heuristic is required to generate a solution in a feasible amount of time. Availability of the underlying non-dedicated processing and communication resources varies continuously during the lifetime of the computation. For example, using the spare clock cycles of open access PCs in a university laboratory as part of a distributed system is highly problematic. PCs can be randomly rebooted, users

can run computationally intensive applications (which leaves few spare clock cycles) and the network can become congested. Information about these resources is thus not known a priori. The tasks to be processed are also not known a priori, because they arrive dynamically. A scheduler is needed which can manage all of these variations to get the most out of an unreliable set of resources.

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 [3,8,10, 28,33,36,37,42]. How to best generate this knowledge is an open problem [37], which is dealt with in different ways. However, 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 accuracy of these estimations becomes particularly important in distributed systems, where users must pay for processing, where demand for computational resources outstrips supply, or where a problem must be processed as quickly as possible. The most common form of estimating task execution times is by benchmarking a task or set of tasks offline in advance [3,8,33,36, 37]. The heterogeneous and non-dedicated nature of the resources in a loosely-coupled distributed system means that this type of estimation can introduce a large amount of error, between the actual execution time and the estimated execution time of a set of tasks.

---

* Corresponding author.
 E-mail addresses: apage@cs.nuim.ie (A.J. Page), tk2@sanger.ac.uk (T.M. Keane), tomn@cs.nuim.ie (T.J. Naughton).

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 [1]. Some distributed systems, such as SETI@home [22], ignore the resources of the system [22,23,32], or treat their heterogeneous resources as a homogeneous set [2,9,22, 23,30,32,40] by ignoring variation in the available computational resources of the processors. Some distributed systems restrict themselves to homogeneous tasks [22,23,32] 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 [3,8,33,36,37]. Many require a directed acyclic graph with task and communication information, and precedence constraints given in advance [3,8,24,26,25,33, 36,37]. Communication costs are also not properly considered by many algorithms, for example all communication links are assumed to be homogeneous [41], it is assumed communication and computation can take place simultaneously [8], or it is assumed that there is instantaneous message passing [43]. 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 some of these restrictive assumptions. Sinnen et al. [33] 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. [5] focus on scheduling the communication between processors, to minimize the communication overhead in a distributed system. Theys et al. [37] 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 [10] 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. [1] 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 paper 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.

The rest of the paper is organized as follows. In Section 2 we present a method for estimating the task execution time. Section 3 contains descriptions of the scheduling algorithms. A real-world distributed system is reviewed in Section 4 and the results of the experiments are presented in Section 5. We conclude in Section 6 and note directions for future research.

2. Problem statement and execution time estimation

The scheduling problem we address in this paper 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 in a star topology. For the remainder of the paper, 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 $X_i = (x^i_1, x^i_2, \ldots, x^i_q)$ and q is the number of parameters. The restriction $X_i \in \mathbb{Z}^q$ is placed on the user in order for the scheduling algorithm to work. 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 scheduler's 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 $t_i = \text{ETC}(X_i, j) + \epsilon$ where $\text{ETC}(X_i, j)$ is the part of the execution time (in seconds) estimated with input vector X_i, j was the processor that task i actually ran on, and ϵ is the error in 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 as a set of sets of observations denoted by O and defined as

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

where c_i is communications overhead. A separate O is maintained for each problem.

Assuming that each previously executed task has a finite running time; past task execution times can be used to predict future execution times [35]. We generate an expected time to compute (ETC) matrix, which gives the expected execution time of each task on each processor, with each row containing the estimated 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 [14], neural networks, support vector machines, and k-nearest neighbors (k-NN) [18]. 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.

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 [4]. 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 [18]. The advantage of k-NN is that it can easily adapt to sparse or dense regions in the distribution.

Two methods are used in this paper to generate estimated task execution times, a k-NN 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. k-nearest neighbors

The estimated task execution times are calculated using selected observations from the set O Eq. (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 $n \longrightarrow \infty$ [7]. We use $k = \lceil n^{4/5} \rceil$ which is shown to perform well in [18]. For example, if $n = 100$ then $k = 40$, so 40% of the observations are selected, whilst if $n = 10\,000$ then $k = 1585$, so 15% of the observations are selected for use in generating an estimated task execution time. To make the algorithm more robust to outliers L-smoothing [15] is used to eliminate a fixed percentage L of the largest and smallest values of t_i from the set of k previously selected observations, which gives

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

observations.

Algorithm 1. Algorithm to estimate the execution time of task i on processor j. Individual steps are explained in the text.

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 - estimated execution time in seconds
Begin
  For each observation $(t_a, X_a) \in U_j$
    Calculate Euclidean distance d_a between X_i and X_a
    (see Eq. (3))
  Sort observations by distance
  Select k observations with smallest distances
  Deselect L% of observations with largest and smallest t_a
  For each selected observation
    Calculate its influence in generating the estimated time ETC
    (see Eq. (4))
  Calculate estimated time ETC from Eq. (5)
  Return ETC
end

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

$$d(X_i, X_a) = \sqrt{\sum_{f=1}^{q} (x_f^i - X_f^i)^2}, \tag{3}$$

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

$$w_a^j(X_i) = \left(\sum_{b=1}^{y} d(X_i, X_b^j) \right) \Big/ d(X_i, X_a^j). \tag{4}$$

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

$$\text{ETC}(X_i, j) = \sum_{a=1}^{y} t_a^j w_a^j(X_i). \tag{5}$$

2.2. 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 [11] and HPCC [16] can provide quite accurate information about system resources, in the context of particular types of computation. Linpack is used in this paper to measure the execution rate of each processor in millions of floating point operations per second (MFLOP/s) [11]. This is a recognized standard used to benchmark systems for inclusion in the list of Top 500 Supercomputers [38].

This 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 [11] 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 $P_j = \Gamma^{Pj}$ using a smoothing function Γ (defined in the next paragraph) that uses the b benchmark results received from processor j up to that point. To avoid privacy issues and proxy problems associated with using IP addresses as unique identifiers, clients remember their own allocated unique identification number. The server then need only keep track of the next unallocated integer. 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 $T_i = \Gamma^{tP}$ 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 $\Gamma_b^a = \Gamma_{b-1}^a + \nu(a_b - \Gamma_{b-1}^a)$, where the smoothness of the sequence of representative values is controlled by $\nu \in [0, 1]$, and where we let $\Gamma_0^a = a_1$. 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

$$\text{ETC}(i, j) = \frac{T_i}{P_j}, \tag{6}$$

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.

3. 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 paper we will define the percentage efficiency as

$$\text{efficiency} = \left(100 \sum_{j=1}^{M} \text{time processor} j \text{ has spent processing} \right) \Big/ (\gamma \times M), \tag{7}$$

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.

3.1. Estimation error

We must estimate the execution time of a task, because it is undecidable to calculate exactly [34]. 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 [4]. The percentage error in the ETC of task i on processor j is

$$\text{ET}(i, j) = \left| \frac{\text{ETC}(i, j) - t_i^j}{t_i^j} \right|, \tag{8}$$

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,

$$\text{CCR}(i, j) = \frac{\text{ETC}(i, j)}{C(i, j)}, \tag{9}$$

where $C(i, j)$ is calculated from Algorithm 1 by substituting t_i for c_i.

A combined error value for a given task-processor mapping is defined as

$$\text{EW}(i, j) = \frac{\text{ET}(i, j)}{\text{CCR}(i, j)}. \tag{10}$$

Eq. (10) produces a small value when the task error (ET) is small and the CCR is large, were a small value is preferable to a large value. 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. The first is to ignore the predicted estimation error in the estimated task 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.

3.2. Algorithm structure

Each of the algorithms described in this section can be described using the scheduling algorithm structure in Algorithm 2, with different functions X affecting the different scheduling algorithms. The input to Algorithm 2 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.

Algorithm 2. The scheduling algorithm template, parametrized by function X

Input: Set of unscheduled tasks, set of processors, load on each processor, a function X
Output: Mapping from tasks to processors
Mapping is initialized to $\emptyset$
while unscheduled tasks remaining
  foreach Unscheduled task i **do**
    minval := MAX_INT
    foreach Processor j **do**
      currval := $X(i, j, \text{load})$
      if currval $\leq$ minval
        $a := i$
        $b := j$
        minval := currval
      endif
    end foreach
  end foreach
  Add (a, b) to the mapping Q
  Update current load on Processor b
  Remove Task a from list of unscheduled tasks
end

3.3. 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

$$\text{FA}(i, j) = \text{ETC}(i, j) + \text{C}(i, j) + \text{ST}(Q_j), \tag{11}$$

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

$$\text{ST}(j) = \sum \text{ETC}(Q_j, j), \tag{12}$$

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 [17]. When used in Algorithm 2 in place of X it will allocate a task to the processor which will finish processing it earliest.

Eq. (11) is combined with the estimation error value from Eq. (10) to produce two scheduling algorithms. The first is characterized by the cost function

$$\text{FE}(i, j) = \text{FA}(i, j)\text{EW}(i, j)^{\beta}, \tag{13}$$

where β controls the proportion of the values of FA and EW receptively which make up FE and $\beta > 0$.
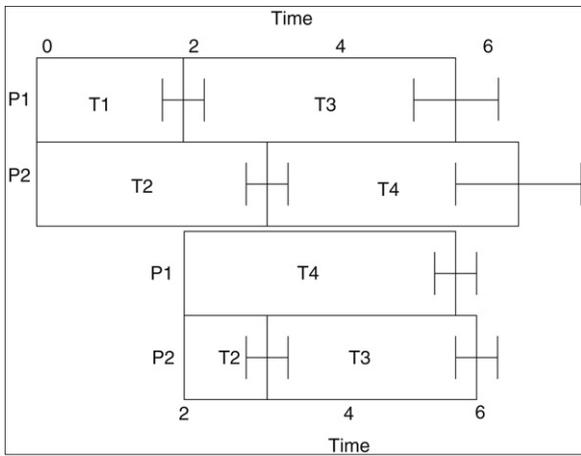
Fig. 1. An example of the FE algorithm at time 0 and 2, with the error bars illustrating the bounds of the estimation error of the task execution time.

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. 1 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

$$FZ(i, j) = \frac{FA(i, j)}{EW(i, j)^{\beta}},\tag{14}$$

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.

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,

$$FE_d(i, j) = FA(i, j)^{EW(i,j)} EW(i, j)^{FA(i,j)},\tag{15}$$

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,

$$FZ_d(i, j) = \frac{FA(i, j)^{EW(i,j)}}{EW(i, j)^{FA(i,j)}}.\tag{16}$$

We tried each combination of EW and FA, where the resulting value is minimized. FE and FZ aim for low execution times. The two other combinations were unfeasible because they favour high execution times.

3.4. 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,

$$LA(i, j) = \sum_{y=1}^{M} \left[\left(\max \left\{ \max_{x=1}^{M}[ST(x)], FA(i, j) \right\} \right. \right.$$
$$\left. - ST(y) \right) - \{ETC(i, j) + C(i, j)\} \right]$$
$$\times \left[\sum_{y=1}^{M} \max_{x=1}^{M} \{ST(x) - ST(y)\} \right]^{-1},\tag{17}$$

is given in Eq. (17). It 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,

$$LE(i, j) = LA(i, j)EW(i, j)^{\beta}\tag{18}$$

and

$$LZ(i, j) = \frac{LA(i, j)}{EW(i, j)^{\beta}}\tag{19}$$

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. (19)). 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. 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 [20]. The system has been successfully deployed on over 350 computers, which were distributed over a number of locations, and has been successfully used to process bioinformatics, biomedical engineering, and cryptography applications.

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

Table 1
Client resources of two heterogeneous distributed systems (A and B)

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

Table 2
Comparison of problem properties

Problem	Ref.	Avg processing (s)	Avg comms (s)	CCR	No. tasks
SLTT	[31]	381.1	3.5	108.8	100
MD5		92.0	3.1	29.3	300
DSEARCH	[21]	731.9	14.0	52.0	612
TSP		215.9	4.5	47.9	121
SHA1		244.1	5.1	47.4	249
ElGamal	[12]	314.5	4.0	77.2	91

5. Experiments

For the experiments described in this section we used two distributed system configurations, one with 90 PCs (Table 1.A) and one with 74 PCs (Table 1.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 5.5 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 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 P4 D820 processors running a 32-bit version of Linux.

5.1. Set of problems

A representative set of heterogeneous tasks for scheduling on a heterogeneous distributed system is an open problem [37]. Estimating task execution times using the KNN approach proves very accurate for standard work load distributions [18], thus we need a more complex set of tasks to properly test the schedulers presented in this paper. To test the scheduler presented in this paper 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 have only a single stage.

The mean CCR is also different for each problem (see Table 2), as is the amount of actual input and output data. Fig. 2 shows the CCR ratio after all tasks have been processed. It forms multiple Gaussian, which is a non-trivial set of tasks to schedule. The processing time of the tasks (see Fig. 3) is heterogeneous, with large outliers, up to over 1000 s. These outliers will cause some algorithms to perform very inefficiently. The overall latency for the system between the server and the processors forms an exponential distribution.

5.2. Estimating system resources

Estimating the system resources and task execution times is a difficult and error-prone. By accurately predicting the error in the estimation of these values, we can use this to help 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.
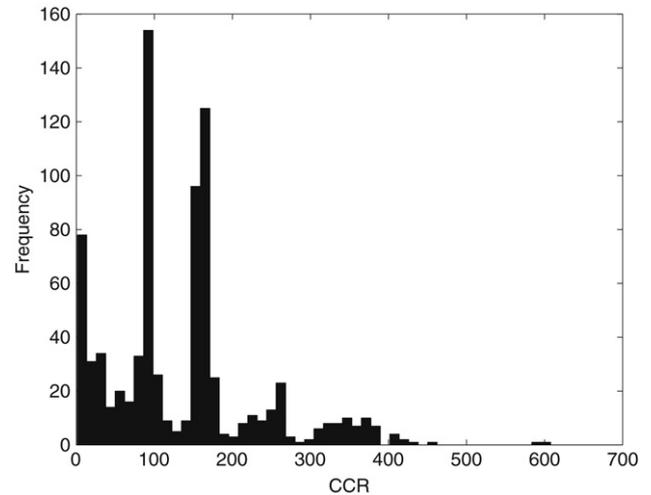
Fig. 2. Histogram of the CCR ratio of the tasks in the test set of problems.
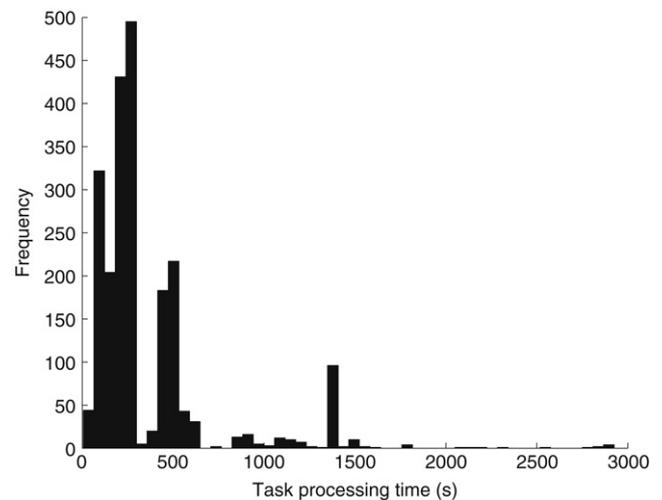
Fig. 3. Histogram of the processing time of tasks in the set of problems.

Fig. 4 shows that the predicted estimation error more closely follows the actual estimation error as time progresses. The actual estimation error of processing resources is low overall. This is measured by periodically running the a 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 consistently low, in the region of 1% due to the homogeneous nature of the communication resources used in this experimental.

5.3. Estimating task execution times

The estimated task execution time using the smoothed estimate and a k-NN are compared to the actual task execution time. Fig. 5 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. 6, which explains the continuous decrease in error from Fig. 5. Taking a simple average of all past task execution times to estimate future task execution times results in a high estimation error.
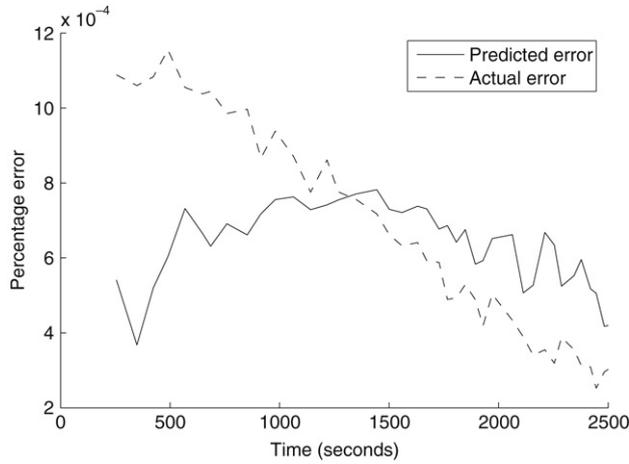
Fig. 4. Predicted computational estimation error and actual computational estimation error over time.

Fig. 5. The absolute percentage error between the actual processing time of a task and the estimated processing time of a task over time with a log scale, using a simple average of past task execution times, a smoothing estimate and a KNN estimate.

Fig. 6. Absolute percentage error between estimated task execution time and k-NN estimate versus the number of observations used to generate the estimate.

In Fig. 7 the predicted and actual estimated task execution time error both approximately linearly decrease over time, with

Fig. 7. Predicted task estimation error and actual task estimation error over time, with absolute values shown.

Fig. 8. The efficiency of 3 load-based schedulers over time.

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.

5.4. Scheduler performance

Two different metrics are used to evaluate the performance of the schedulers given in this paper: (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 (see Fig. 8–9).

The load-based schedulers (LA, LE, LZ) provide approximately 21%–55% efficiency overall, with LA providing the most efficient solution, as shown in Fig. 8. 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.

Table 3
Comparison of schedulers

Scheduler	Makespan (s)	Sched. time (s)	% Efficiency	% Comms
FA	3 505	28	51	0.6
FE	6 202	547	33	0.8
FZ	2 408	108	66	1.6
LA	4 114	344	55	0.7
LE	17 013	1850	21	0.3
LZ	12 138	426	33	0.3
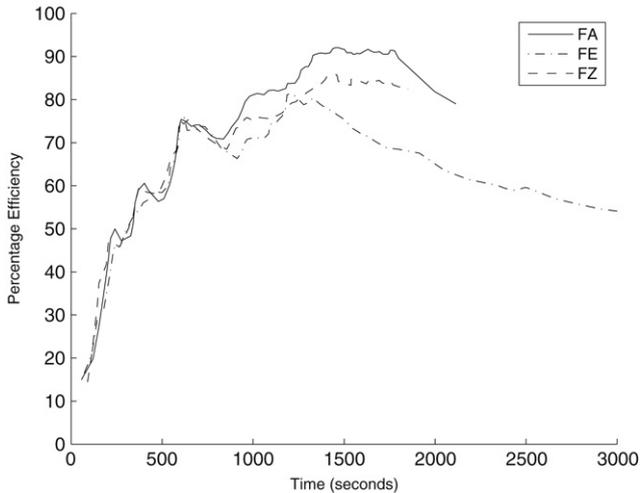
Fig. 9. The efficiency of 3 makespan-based schedulers over time.

Table 4
Comparison of common schedulers which do not use estimation error to FZ, the best performing scheduler, which uses estimation error

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

The makespan-based schedulers (FA, FE, FZ) provide the best overall efficiency achieving above 80% at some points, as shown in Fig. 9. 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 paper at 2408 s. 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 3). 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.

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). Tabu search optimization (TA) is an evolutionary based scheduler based on OpenTS [29].

Table 5
Experiments with 74 processors (see Table 1.B) varying the value of β, where d is a dynamic value of β (see Eq. (15))

Strategy		β	Makespan (s)	Sched. time (s)	% Efficiency	% Comms
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

b and w are the best and worst case strategies respectively.

A simulated annealing (SA) based scheduler was created using Jannealer [19]. 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 that a good comparison was available. With TA, we recursively broke down the problem to be solved, into a tree like structure of depth $\log N$, 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 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 [27] (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, FZ has a makespan of 2408 s, which is between 27% and 284% better than the other schedulers with the exception of TA, which has a makespan of 2351 s. 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 which does not consider estimation error, where both use the same estimated input parameters.

5.5. 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 1.B. Table 5 describes the results of these experiments. Each experiment was repeated twice, and the average is given.
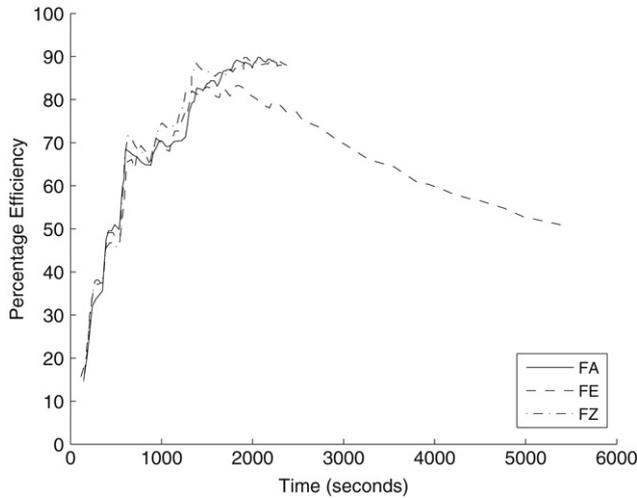
Fig. 10. Efficiency of multiple schedulers with best case task execution times.
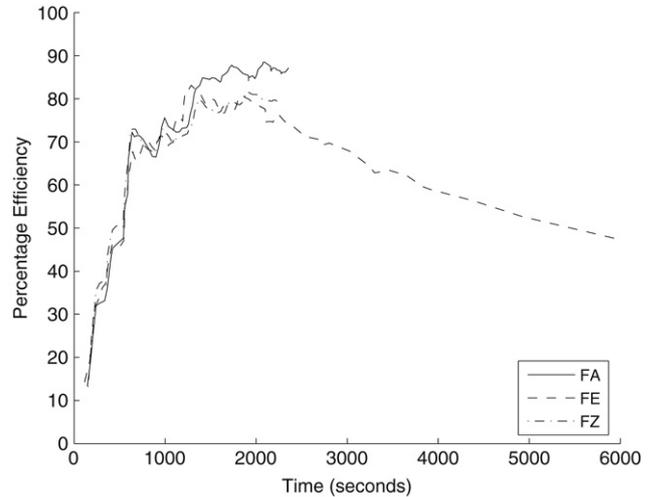
Fig. 11. Efficiency of multiple schedulers with worst case task execution times.

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.

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 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. 10 and 11 compare the efficiency of FA, FE, and FZ using best and worst case task execution times. The best performing β value for FE and FZ is used in each figure (see Table 5). FE is clearly far less efficient than FA or FZ. This is consistent in all experiments (see Table 5), where FE schedules tasks on the slowest processors in the system near the end of the overall schedule, resulting in a large makespan.

Fig. 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. 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.

Compare this to the worst performing schedule in Fig. 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.
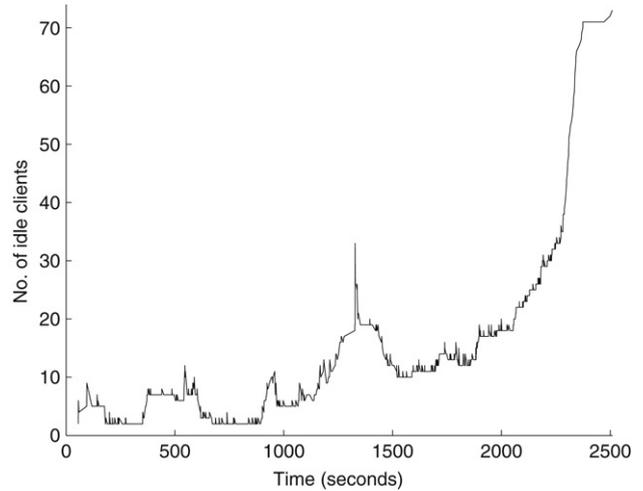
Fig. 12. Number of idle processors over time when using FZ with $\beta = 0.1$ and best case task execution times.
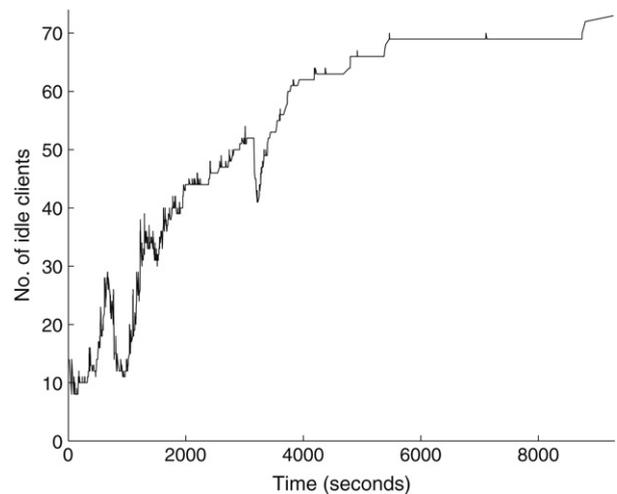
Fig. 13. Number of idle processors over time when using FE with $\beta = 1.0$ and best case task execution times.

6. 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. Accurate 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.

We have developed a simple scheduler, which starts with no advanced knowledge of the system resources, or the problems to be processed. It estimates the system resources and the task computational requirements dynamically at run-time, and adapts to the constantly changing state of the system. Experiments were performed on a real-world heterogeneous distributed system with up to 90 processors, with non-dedicated resources, and processed real-world problems from the areas of computer science, bioinformatics, and biomedical engineering. All information about the tasks and system resources was generated online using analytical benchmarking and a k-NN algorithm.

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.

The distributed system software is freely available under an open source GNU GPL license from the system homepage located at http://distributed.cs.nuim.ie.

Acknowledgments

References

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

[2] D. Anderson, Public computing: Reconnecting people to science, in: Conference on Shared Knowledge and the Web, Madrid, Spain, 2003.

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

[4] R.J. Carroll, D. Ruppert, L.A. Stefanski, Measurement Error in Nonlinear Models, Chapman & Hall, Boca Raton, 1998.

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

[6] F. Dehne (Ed.), Coarse Grained Parallel Algorithms for Scientific Applications, vol. 45, Springer, New York, 2006 (special issue).

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

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

[9] Distributed.net, http://www.distributed.net.

[10] A. Dogan, 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) (2002) 308–323.

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

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

[13] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co, New York, NY, 1979.

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

[15] W. Härdle, Applied Nonparametric Regression, Cambridge University Press, 1990.

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

[17] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, Journal of the ACM 24 (2) (1977) 280–289.

[18] M.A. Iverson, F. Ozguner, L. Potter, Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment, IEEE Transactions on Computers 48 (12) (1999) 1374–1379.

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

[20] T. Keane, R. Allen, T.J. Naughton, J. McInerney, J. Waldron, Distributed Java platform with programmable MIMD capabilities, in: N. Guelfi, E. Astesiano, G. Reggio (Eds.), Scientific Engineering for Distributed Java Applications, in: Springer Lecture Notes in Computer Science, vol. 2604, 2003.

[21] T.M. Keane, T.J. Naughton, DSEARCH: Sensitive database searching using distributed computing, Bioinformatics 21 (8) (2005) 1705–1706.

[22] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky, SETI@HOME massively distributed computing for SETI, Computation Science & Engineering 3 (1) (2001) 78–83.

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

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

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

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

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

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

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

[30] A. Page, T. Keane, R. Allen, T.J. Naughton, J. Waldron, Multi-tiered distributed computing platform, in: 2nd International Conference on the Principles and Practice of Programming in Java, Kilkenny City, Ireland, 2003.

[31] A.J. Page, S. Coyle, T.M. Keane, T.J. Naughton, C. Markham, T. Ward, Distributed monte carlo simulation of light transportation in tissue, in: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society, Rhodes, Greece, 2006.

[32] T. Silvestre, E. Nugues, G. Perrière, M. Gouy, L. Duret, Phylojava : A generic client–server tool for phylogenetic tree reconstruction — application to grid computing, in: M.-F. Sagot, H.-P. Lenhof (Eds.), European Conference on Computational Biology, Paris, France, 2003.

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

[34] M. Sipser, Introduction to the Theory of Computation, 2nd ed., Thomson, Boston, 2006.

[35] A. Stuart, J. Ord, Kendall's Advanced Theory of Statistics, Vol. 1: Distribution Theory, 6th ed., Edward Arnold, London, 1994.

[36] A. Swiecicka, F. Seredynski, 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) (2006) 253–262.

[37] M.D. Theys, T.D. Braun, H.J. Siegal, A.A. Maciejewski, Y.-K. Kwok, Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach, John Wiley and Sons, New York, USA, 2001, pp. 135–178 (Chapter 6).
[38] Top 500 Super Computers. http://www.top500.org, 2005.
[39] J.D. Ullman, NP-complete scheduling problems, Journal of Computing System Science 10 (1975) 384–393.
[40] United Devices, Grid MP Platform Architecture, white Paper, 2003.
[41] L. Wang, H.J. Siegel, V.P. Roychowdhury, 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) (1997) 8–22.
[42] M.-Y. Wu, W. Shu, A high-performance mapping algorithm for heterogeneous computing systems, in: Parallel and Distributed Processing Symposium, Proceedings 15th International, San Francisco, CA, USA, 2001.
[43] A.Y. Zomaya, Y.-H. Teh, Observations on using genetic algorithms for dynamic load-balancing, IEEE Transactions on Parallel and Distributed Systems 12 (9) (2001) 899–911.

Thomas M. Keane received a B.Sc. in Computer Science and Software Engineering in 2002 and an M.Sc. in Computer Science in 2004 from the National University of Ireland, Maynooth. For his Ph.D., Thomas moved to work at the Bioinformatics laboratory at the National University of Ireland, Maynooth, working in the area of phylogenetic methods and high-throughput phylogenomics using distributed computing, completing in 2006. He is currently a Senior Computer Biologist in the Pathogen Genomics group at the Wellcome Trust Sanger Institute. He now works in the area of new sequencing technologies and manages the sequence assembly analysis for the several pathogen genome projects. His research interests include distributed computing and bioinformatics.

Thomas J. Naughton received a B.Sc. degree (double honours) in computer science and experimental physics from the National University of Ireland, Maynooth, Ireland. He has worked at Space Technology (Ireland) Ltd. and has been a visiting researcher at the Department of Radio-electronics, Czech Technical University, Prague, and the Department of Electrical and Computer Engineering, University of Connecticut, Storrs. He is a Senior Lecturer in the Department of Computer Science, National University of Ireland, Maynooth, with a permanent appointment since 2001. Since 2007 he has been a European Commission Marie Curie Fellow at Oulu Southern Institute, University of Oulu, Finland. He leads the EC FP7 three-year eight-partner collaborative project Real 3D. His research interests include optical information processing, computer theory, and distributed computing. He has served as a committee member on 12 international IEEE, ICO, and SPIE conferences. He has co-authored more than 150 publications including 35 journal articles and 20 invited conference papers.

Andrew J. Page received a B.S. degree in Computer Science and Software Engineering from the National University of Ireland, Maynooth in 2003. He is currently working toward his Ph.D. degree in the same university. His research interests include scheduling in distributed systems and digital holography. He is now a postdoctoral researcher at the National College of Ireland, Dublin.