A Dynamic, Decentralised Search Algorithm for Efficient Data Retrieval in a Distributed Tuple Space

Alistair Atkinson

Eastern Institute of Technology
Hawke's Bay, New Zealand
Private Bag 1201, Hawke's Bay Mail Centre, Napier 4142
Email: aatkinson@eit.ac.nz

Abstract

This paper presents an algorithm which may be used to efficiently search for and retrieve tuples in a distributed tuple space. The algorithm, a core part of the Tupleware system, is based on the success or failure of previous tuple requests to remote nodes in the system, and this data is used determine the relative probability of particular remote nodes being able to fulfil subsequent future requests. The logic of this algorithm is distributed and decentralised: each node dynamically calculates its relationship with other nodes at runtime. The behaviour of the algorithm using two applications is analysed, and shows significant improvement in terms of efficiency and performance compared to comparable tuple space implementations.

Keywords: tuple space, data retrieval, locality, distributed computing.

1 Introduction

This paper describes a search algorithm for efficiently retrieving tuples on a cluster-based distributed tuple space. This algorithm forms one of the core part of the Tupleware cluster middleware, a high-level description of which can be found in [3].

The Tupleware system was implemented with the goal of achieving a scalable platform for the implementation and execution of parallel array-based applications, whilst maintaining the simplicity and transparency of the original tuple space paradigm [7].

In order to achieve scalability, it was decided to use a decentralised approach, and to distributed the tuple space across nodes in the cluster. It followed, then, that it would be necessary to store and search for tuples in the most efficient manner possible, and it is for this reason that the search algorithm presented in this paper was developed.

To evaluate the performance and scalability of Tupleware (and its search algorithm) the performance of two non-trivial applications are presented. Ease of programmability is discussed in the previous paper cited above.

The contribution of this research is that it provides an investigation into a concrete implementation of distributed tuple space using non-trivial data-parallel applications. This area of research, while studied previously as we will see in Section 3, often focusses on theoretical models which lack a concrete implementation, and others seek to provide a more general-purpose platform. This research project has focussed on a particular class of applications in order to exploit their common characteristics, which has informed the development of the search techniques being presented.

2 Motivation

Tuple spaces, first introduced by Gelernter's Linda coordination language [7], are recognised as offering many advantages over the more common message-passing model as a distributed computing paradigm. These advantages include: a decoupling of the computations and coordination parts of a parallel program, both temporal and geographic distribution, loosely-coupled interaction between processes, and a higher level of abstraction which unburdens the programmer from needing to explicitly deal with lower-level details of inter-process communication.

However, the adaption of the tuple space model to a distributed environment poses some additional challenges compared to its implementation on multiprocessor computers. Namely, the increase in latency and relatively restricted network data transfer rates cause any operations involving network communication to become relatively expensive, and also distributed systems generally need to be able to scale to large number of nodes. This means that the available network bandwidth must be used as efficiently as possibly, and thus it is necessary to minimise the number of communication events in a given system in order to achieve this.

These limitations were illustrated in [18], which described the scalability of various tuple space implementations under varying loads, and showed the scalability of the included systems to be relatively poor. A study presented in [15] also outlined the issues involved with efficiently implementing a tuple space in a distributed environment, and presented a performance evaluation of an unmodified JavaSpaces system using applications with similar characteristics to those presented in this paper. The results of this evaluation showed the limitations of a centralised tuple space for tightly-coupled applications, and its more commendable performance for loosely-coupled replicated-worker style applications.

These factors are what motivated this particular research, which addresses these issues, and proposes a search algorithm suitable for use in a distributed tuple space which can be utilised for array-based parallel applications.

3 Previous Work

Some of the more notable systems which feature distributed or multiple tuple spaces are briefly described in this section. The systems included can be contrasted in terms of the transparency of their distribution, the logical integration of the tuple space(s), and whether or not the system logic is centralised or decentralised.

3.1 Multiple Tuple Spaces in Linda

Multiple Tuple Space Linda [14] (usually abbreviated to MTS-Linda) was one of the earliest attempts to add multiple tuple spaces to the original Linda model. MTS-Linda incorporates tuple spaces which are treated as first-class objects, and can be manipulated by the programmer to suit an application's requirements. The use of multiple tuple spaces allowed data (represented as passive tuples), and processes (represented as active tuples), to be grouped and manipulated as a whole. As tuple spaces are treated as first class objects, each tuple space is simply conceptualised as a ”local data structure within a process” [14], which goes some way towards raising the level of transparency of the system's distribution.

Tuples which reside in other (non-local) tuple spaces may also be accessed, provided they are within the ”context tuple space” of the process making the access request. That is, multiple tuple spaces may belong to the same context, and processes may opt to retrieve tuples from either its local tuple space, or from a tuple space contained in the same context. In this way, multiple tuple spaces are added in a hierarchical manner, rather than the flat, or disjoint way they have been incorporated in some other systems.

Another attempt at implementing multiple tuple spaces for Linda was by Rowstron & Wood [16], who adapted the Linda model to networks of heterogeneous workstations. This system did not propose a new way of adding multiple tuple spaces to the system, but simply assumed that they existed. The main contribution this system made was the addition of new tuple space access primitives, namely bulk retrieval operations `collect()` and `copy-collect()`. The former operation moves a set of matching tuples from one tuple space to another, and the latter performs a similar function, except matching tuples are copied from one tuple space to another [17]. This implementation classifies tuple space as either local or remote, the main difference being that tuples stored in a local tuple space are not accessible by remote nodes in the system, whereas those stored in a remote tuple space do not have this restriction. Further, local tuples are stored locally, in the local processes address space, whereas remote tuples are stored on remote *tuple space servers*, which generally reside on separate, dedicated nodes on the network. The decision as to whether a given tuple is classified as being local or remote is performed dynamically at runtime by the system kernel.

The bulk operations allowed the movement of multiple tuples using only a single operation, whereas in the original Linda model this would have required multiple invocations of the tuple spaces access operations. This factor, along with the optimisation of the locality of stored tuples, allowed the system to make more efficient use of the network, and to realise some significant performance gains compared to traditional implementations [16].

3.2 SwarmLinda

In terms of algorithmic approaches to tuple search and retrieval in systems with multiple or distributed tuple spaces, there are a small number of proposed systems at the time of writing that have this as their focus. The most notable of these is arguably the SwarmLinda distributed tuple space described in [11].

SwarmLinda employs a tuple storage and retrieval algorithm inspired by the collective intelligence displayed by swarms of ants. SwarmLinda is characterised by agents (in this case, ants) acting individually, but whose individual actions combine to exhibit a collective intelligence. These agents ”act extremely decentralised” and perform their actions ”by making purely local decisions and by taking actions that require only a few computations” [5].

The collectively intelligent behaviour displayed by these ant swarms relates to the locality or tuple storage, and efficient tuple retrieval from the network. In short, when a new tuple is produced, it will be stored by one of the ’ant’ agents on a node which stores tuples with the most similar characteristics to the new tuple. Tuples with a particular characteristic emit a ’scent’, and this scent is used by the agents when they need to retrieve a tuple. The characteristics of the tuple to be retrieved, along with the scent being emitted, is used to guide the search of the agent. It is argued that a SwarmLinda system will dynamically adapt itself to the characteristics of the tuples being stored, so that tuple retrieval operations will tend towards optimal over time. However, at the time of writing, these ideas have not been implemented in a concrete system, and as such no performance data are available to determine their effectiveness.

3.3 Scope

Scope [12] is a formal model for the addition of multiple tuple spaces to Linda-like systems. It aims to address the scalability problem of Linda, and also to increase the expressiveness of Linda-like operations so as to enable operations such as transactions, and prevent semantic limitations such as the multiple-read problem. Most relevant to the research presented in this paper, however, is the generalised way in which Scope handles the issue of multiple tuple spaces, in particular its idea of ”overlapping” tuple spaces.

Multiple spaces have traditionally been added to tuple space systems in one of two way: by nesting spaces hierarchically, as in MTS-Linda, or by simply adding disjoint spaces which have no logical relationship, as in JavaSpaces [12].

Scope presents a generalised approach to the addition of multiple spaces, introducing the idea of overlapping tuples spaces. This allows some parts of each space to be shared, and other parts to be separate. In concrete terms, tuples are able to belong to more than one space at a time. Essentially, each ”portion” of tuple space is represented by a named scope, and these portions can be combined and arranged based on defined scope operations. These operations are based on the set operations union, complement and intersection, and can be used to define tuple membership to one more more scopes. The expressiveness of Scope allows it to implement hierarchical and disjoint tuple spaces in addition to overlapping spaces.

A concrete implementation of Scope is presented in [13]. However, no performance results are available for any Scope-based implementation, and no subsequent research seems to have been done at the time of writing.

3.4 JavaSpaces

JavaSpaces [10] is an implementation of the spaces paradigm from Sun Microsystems. Specifically, it is a service which forms part of the Jini distributed software architecture. It provides a stand-alone object space, called a JavaSpace.

The system may have more than one space, however each space is a separate entity and their respective roles in the system are not coordinated. Each application must contain the logic for utilising the available JavaSpaces infrastructure.

Like most derivative implementations of the tuple space model, JavaSpaces is an effective platform for implementing a range of distributed applications and utilising common design patterns. In particular, it has been shown in [2] to be ideally suited to the Master/Worker style of parallelism, particularly coarse-grained parallel applications. For applications which are more fine-grained or tightly-coupled, JavaSpaces can experience scalability problems due the the increased communication demands inherent in these applications (see, for example [18]).

4 Tupleware Overview

Tupleware is a library and runtime system which provides a distributed tuple space platform for computing clusters. It is aimed specifically at array-based numerical and/or scientific applications, which exhibit common characteristics that may be exploited in order to optimise the communication patterns between cluster nodes.

Tupleware has previously been described in [3], so the entire system will not be covered in great detail again here. Instead, the description will be restricted to a high-level overview of the operation of the main components of the system only, in order to inform the discussion of the performance results. For a more detailed description refer to the previous publication referenced above.

4.1 System Architecture

A complete Tupleware system consists of a collection of nodes, each of which hosts its own local partition of the tuple space. These local partitions, combined, constitute the whole (distributed) tuple space.

The main components of the Tupleware architecture consist of the following: a runtime system, a tuple space service, and tuple space stub objects. These components form a layered architecture upon which an application module can execute. An example Tupleware system consisting of two nodes is illustrated in Figure 1.

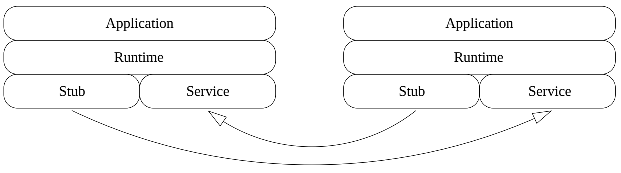

Figure 1: Architecture of a two-node Tupleware system.

4.2 System Components

The main components are briefly described as follows:

4.2.1 Tuples & Templates

The fundamental data object in a tuple space system is a tuple, which are used to encapsulate one or more data objects. A tuple has one or more fields each of which contain a value. Fields should not contain any **null** values, and tuples are treated as immutable objects.

Templates are used to perform content-based associative lookup on tuples. Templates are similar to tuples in that it encapsulates a set of data fields. However, unlike a tuple, some (or all) of these fields may be assigned **null** values, denoting wildcards which may match against any value during associative lookup. Associative lookup involves the use of the `matches()` method, which determines whether a template matches a given tuple. The matching function has the same semantics as the original Linda: a template must be an equivalent length, and its specified values must be equal to a given tuple in order to positively match.

4.2.2 Local Tuple Space

A local tuple space provides the basic functionality required for tuple storage and lookup on a single node. The local space maintains all of the stored tuples on node, and is searchable by a node's service in response to tuple requests from remote nodes.

The local tuple space is thread-safe, and, in most instances,uses the first three fields of a tuple as the key to a hash table which references the tuple data itself. The reasoning behind this was that in almost all applications relevant to those targetted by Tupleware, the first three elements are always those used to identify and index the array, and, in the case of the ocean model, more than one iteration of previous array values will need to be stored (usually two, and sometimes three).

An example of this arrangement is shown in Figure 2.

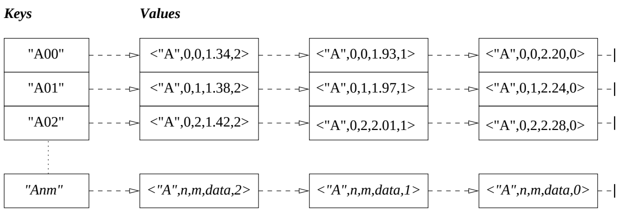

Figure 2: Local tuple storage using a hash table.

4.2.3 Inter-node Communication

Communication between nodes is carried between the stub and service components of the system. All communication instances are initiated by the stub objects, which send requests for tuples across the network to the service running on a remote node. This service provides the means by which remote nodes may search other nodes' local tuple spaces, and always answer queries directly rather than forward unfulfillable requests.

4.2.4 Runtime System

The Tupleware runtime system contains the core system logic, and control the operation of the lower level components such as stub objects and each node's local tuple space. The runtime system initiates and controls the search and retrieval of remote tuples, using the search function presented in the following section. Each nodes' runtime system maintains a collection of stub objects, each of which corresponds to a remote node in the system.

Finally, the runtime system presents an API for use by the top level application layer. It is the only interface into the Tupleware system for an application, and is largely responsible for maintaining the transparency of the tuple space's distribution on the cluster.

4.2.5 Application Processes

At the highest layer, application processes implement the application logic which in turn interfaces with the underlying runtime system. Applications have a reasonably transparent interface to the distributed Tupleware system, and are able to make use of the standard Linda-style operations.

5 Search Algorithm

5.1 Overview

The principle behind the search algorithm is to minimise the number of communication instances required to retrieve a tuple by targetting retrieval requests to those nodes which have the highest probability of being able to satisfy the request, based on the success of previous requests. This technique was adopted due to the nature of the applications at which Tupleware is targetted for use. These are generally array-based applications in which the array is decomposed into individual regions, and each region is processed in parallel.

The characteristics of applications such as these is that any communication between processes is going to tend to occur between those processes which are processing ”neighbouring” regions of the array, whereas nodes processing unrelated regions of an array are going to tend to communicate very rarely, if at all. It is these observed characteristics that we wish to be reflected in the tuple search patterns carried out by the runtime, and the search function provides the platform for achieving this.

5.2 Search Algorithm Operation

An executing Tupleware system consist of individual nodes, each with its own runtime system, each of which will need to communicate with a subset of all nodes in the system almost exclusively, and rarely if at all with all other nodes. These groupings, or clusters, of nodes will emerge quickly during the execution of an application as each individual runtime system dynamically adapts to the patterns of communication instances it is tasked with carrying out.

Underpinning the operation of the search algorithm is a *success factor* which is associated with each tuple space stub object maintained by the runtime system. The success factor is a numerical value between zero and one, and is used to denote the likelihood of the tuple space service associated with a given tuple space stub being able to fulfil a request for a tuple. A higher success factor represents that there is a greater chance of success, and vice versa.

At the beginning of an application’s execution, each stub has a success factor of 0.5 as there are no previous requests from which to calculate another value. A value of 0.5 is meant to represent an intermediate chance of success. Due to all stubs starting with an equal success factor, the initial requests made are random, however the success factor will be recalculated based on the success or failure of these requests, and quite quickly a distinct ordering, or ranking, emerges which can be used to prioritise subsequent requests.

The recalculation of the success factor occurs every time a stub is used to perform a request, and is based on the following equation:

$$S = \begin{cases} S + (1 - S) \times A & \textit{Success} \\ S - S \times A & \textit{Failure} \end{cases}$$

where:

- S is the success factor, and
- A is the adjustment factor.

The adjustment factor is a floating point value between zero and one used to specify by how much the success factor should be adjusted each time it is recalculated. This value will determine how quickly the success factor moves towards either one or zero, or in other words, by how representative a successful request is in terms of the prioritisation of subsequent requests.

This value should be chosen based on the application and the number of processes in a system. If there is a weak relationship between the data being computed on each process, then each process may ultimately end up needing to communicate with a relatively large number of other processes. In cases such as this a small adjustment factor should be used, as one successful request to a remote tuple space does not imply that there is a much greater probability of success for future requests. However, if there is a tight relationship between the data segments being computed by each process, then it follows that these processes will likely communicate very frequently, and that a successful request should have a higher bearing on the probability of success for subsequent requests.

In practice, an adjustment factor of 0.2 was used as it reflects the characteristics of these particular applications. An adjustment factor of greater than 0.5 would reflect a fairly volatile system with a very weak relationship between processes, where a value of 0.1 or 0.2 would represent a more stable relationship.

An an application performs each iteration of its processing, the success factor will be recalculated, and over a relatively small number of iterations, the success factor associated with a node’s neighbouring nodes will be greater than non-neighbouring nodes. An example of this is illustrated in Figure 3.

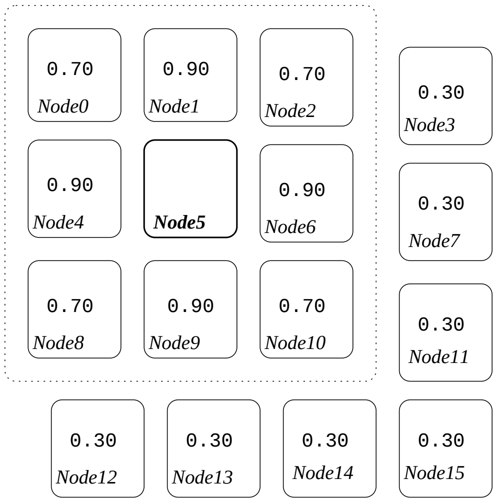

Figure 3: The success factor associated with a node’s neighbouring nodes.

5.3 Benefits of the search algorithm

The search algorithm being presented here can be contrasted with the one used previously, which relied on a most-recently-successful approach to order tuple requests to remote nodes. The problem with this approach was that, while it was more effective than a blind search, it took into account only the success of the immediately previous search.

This was a non-optimal solution for the types of application being targetted by this system, especially in the case of the ocean model, as quite often data retrieval will involve several requests to several neighbouring nodes (one for each boundary being updated). In these cases an null response from a remote node does not necessarily mean that the request cannot be fulfilled by the said node in the future, but rather often it is that the required tuple simply has not yet been produced at the current point in time. However, the result would be, using the most-recently-successful approach, for this particular remote node to be treated as though is should be given a much lesser precedence for subsequent searches, which is not necessarily the desired outcome.

On the other hand, the search algorithm being presented here imposes much less drastic modification to search precedence, as it takes into account all historical retrieval requests (with a greater weight to those carried out more recently). The search precedence given by remote nodes’ associated success factors provides a more accurate reflection of the actual probability of a successful tuple retrieval.

Another benefit implicit to this search algorithm is its decentralised nature: updates to the success factor associated with each remote node is performed by each individual runtime without requiring any sharing of global state information between nodes. Put simply, each node maintains its own unique ”view” of the cluster’s tuple storage, based on its own search history. This eliminates any overhead associated with the transmission of global state information.

Finally, the search algorithm executed dynamically and allows for changes and reconfigurations to tuple storage on the cluster. If the storage characteristics change, the groupings illustrated in Figure 3 will alter to reflect this.

5.4 Summary

In this section we have presented a dynamic, decentralised search algorithm which is used to guide searches for tuples stored on remote nodes of a cluster. The algorithm has been contrasted with the one previously used by the Tupleware system, and its benefits discussed.

In the following sections we present the performance characteristics of the system using two non-trivial applications.

6 Applications

6.1 Overview

Two applications were used to test the behaviour of Tupleware: a 2-D ocean modelling application, and a parallel sorting application based on a modified quicksort. These applications were chosen for their contrasting characteristics, namely their different levels of granularity and communication characteristics. However, both of these applications involve processing segments of an array in parallel, and both are able to benefit from the search algorithm being presented in this paper.

Each application is briefly described below.

6.2 Ocean Model

The ocean model is a two-dimensional simulation of an enclosed body of water. The model calculates the water current velocity and surface elevation based on a given wind velocity and bathymetry.

The body of water is represented by the application as a 2-D grid, and each cell in the grid represents a single grid point. Grid points each individually store descriptive data, including the depth of the water at that point, along with the surface elevation and current velocity. Wind velocity is assumed to be constant across the grid. The variables stored in each grid point are staggered in such as way that the *u* and *v* variables are associated, respectively, with the x-axis and y-axis edges of each grid point. The *eta* variable is representative of the centre point of each grid point.

When executed, the model iterates through a specified number of time-steps; and at each time-step, the surface elevation and current velocity values of each grid point are recalculated based on the values stored at neighbouring grid points. This process continues for the specified number of time-steps, at which point the model should be in a steady-state and thus finished.

The model is parallelised through domain decomposition of the grid, which splits the grid into a number separate *panels*, up to the number of nodes available for processing. Each panel is assigned to a specific node, whose responsibility it is to perform the processing on the panel. As each panel represents only part of the complete grid, at each iteration it is necessary for the boundary values of each panel to be retrieved from neighbouring panels. This process is illustrated by Figure 4, which shows a 9x9 grid which has been decomposed into three panels. Each 9x3 panel has a halo region, represented by the shaded cells, whose values are updated after each iteration of the model. The arrows between neighbouring halo region cells represent the communication instances which are involved in each boundary update.

6.3 Quicksort

Quicksort [9] is a widely used sorting algorithm with an average case execution time of $O(n\log n)$. It is an efficient general-purpose sorting algorithm which rarely exhibits its worst-case execution time.

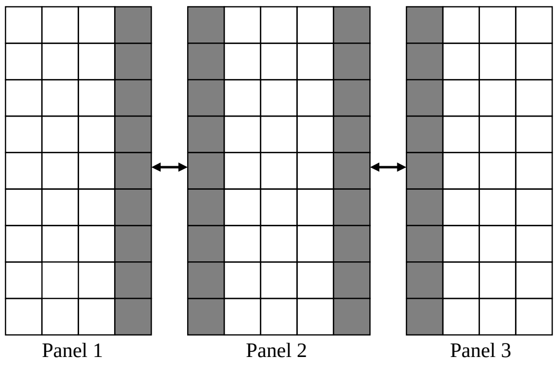

Figure 4: Updating panel boundary values.

There are several characteristics of the quicksort algorithm which led to it being used in the performance evaluation which follows. Firstly, parallelisation of quicksort (and modified and implemented here) is reasonably straightforward, and produces processing tasks which are loosely-coupled and have only moderate data dependencies. Secondly, by modifying the quicksort algorithm so that partitioning ends when an array segment length reaches a certain predefined threshold, it is possible to adjust the granularity of the parallelism exhibited by the sorting algorithm. This feature is useful as it allows us to evaluate the performance of the system with various levels of communication frequency.

The algorithm used to evaluate the system in this paper is a modified version of quicksort. As described above, unsorted arrays are repeatedly partitioned until their length is less than or equal to a predetermined threshold value. At this point, partitioning ends and the remaining unsorted array segment is sorted using some other sequential algorithm; in this case, insertion sort [6].

The ability to adjust the granularity of the application in this manner is useful, as it assists in determining the true scalability of the system and at which point the communication requirements begin to outweigh the benefits of the distribution of the application.

7 Performance Evaluation & Analysis

7.1 Metrics

The metrics used to evaluate the behaviour and performance of Tupleware as presented in this paper are the *runtimes*, which are the wall-clock timings of the execution of various part of the application, and from which we can derive the *speedup* delivered by the Tupleware system. Speedup as used here does not differ from its standard usage in this area, that being the ratio of sequential runtime to parallel runtime [4, p. 74].

From these metrics, we can then make assertions regarding the *scalability* of the system as a whole. There are two aspects of scalability which will be outlined: scalability in terms of the number of processors, and scalability in terms of the problem size. The former is directly related to speedup and Amdahl's Law [1]; if a parallel program tends towards a speedup of N when executed on N processes, then it is said to be scalable. The latter aspect, scalability in terms of problem size, is concerned with how effectively the problem can be split amongst the available processors. That is, if a parallel system can execute a problem of size S in a time of T, then if the size of S doubles we wish the execution time to be no greater than $2.T$. If doubling the problem size results in significantly more than doubling the total runtime, the system would not be scalable in terms of problem size.

7.2 Execution Environment

The performance testing was conducted on a sixteen-node cluster, with each node consisting of a Pentium 4 (3GHz) processor with 1GB of memory, and running Ubuntu Linux 8.04 (kernel 2.6) along with Java 6 update 10. Nodes were connected by a 100Mbps Ethernet network. Performance profiling was carried out using the *Clarkware Profiler* (Clark, 2008), which is able to measure the total and per-iteration runtimes (wall-clock time) between specified points in a program. Each process was executed with the following Java command-line options: `-Xms512M` to set an initial heaps size of 512MB, and `-Xmx2048M` for a maximum heap size of 2GB.

7.3 Ocean Model

The ocean model was tested on a varying number of nodes, from one through to sixteen. When discussing the number of nodes taking part in the system, we are specifying the number of worker nodes. In all Tupleware application these always exists exactly one master process in addition to one or more of the worker processes.

The size of the grid was also varied for experimental purposes, ranging from 1200x1200 through to 2400x2400 in increments of 200x200. This gives a substantial range of grid sizes, keeping in mind that the total number of grid points increases exponentially as the grid grows larger; this is illustrated in Table 1, which also details the amount of raw data stored in each size grid. A 2400x2400 grid was the largest possible for execution before some nodes, particularly the master node, began to use virtual memory, which began to artificially effect the behaviour of the system.

Grid Size	*Grid Points (million)*	*Data (MB)*
1200x1200	1.44	87.9
1400x1400	1.96	119.6
1600x1600	2.56	156.3
1800x1800	3.24	197.8
2000x2000	4.00	244.1
2200x2200	4.84	295.4
2400x2400	5.76	351.6

Table 1: Total grid points and data size.

The number of timesteps completed by the model remained constant at fifty; this gave the system sufficient parallel execution in order for a rigorous performance evaluation to be performed.

7.3.1 Results

The overall speedup of the ocean model application is shown in Figure 5.

This gives us a high high-level overview of the behaviour of the system, however it would be useful to separate the performance of the systems in terms of its sequential and parallel execution. The runtimes of each of these execution phases are illustrated in Figures 6 and 7.

As we can see, an increase in the number of nodes substantially decreases the time taken for the initial application data to be delivered to each worker node, and for the final processed panels to be returned to the master node. This would most likely be due to the use of a switched network, which would allow data to be sent simultaneously to each worker node over each point-to-point circuit. Increasing the number of worker nodes also reduces the size of data being transferred, as the width of each panel would be smaller.

These figures also show that in the cases of fourteen and sixteen nodes, the decrease in sequential runtime becomes negligible, or in some cases increases. This is likely due to the reliance on the master node, which is responsible for either transmitting or receiving all of the data. If we extrapolate these results to a larger number of nodes, then it is likely that these times would continue to slightly increase. However, quite plainly there are significant speedup benefits attained by adding extra nodes to the system in terms of these sequential runtimes.

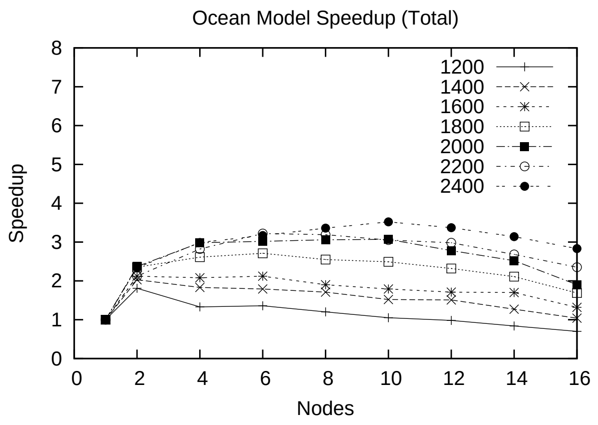

Figure 5: Ocean model's overall speedup.

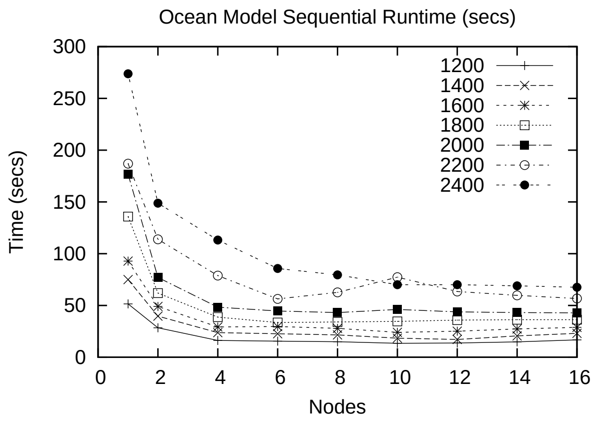

Figure 6: Sequential runtimes (secs) for varying number of nodes in the ocean model.

Following on from the sequential runtime of the application, we can turn our attention to the behaviour during parallel execution. During this phase of execution, the worker nodes behave in a completely decentralised way, and communicate directly in order to share boundary values at each timestep.

The first conclusion which can be drawn from this result is that increasing the number of nodes decreases the width of each panel, resulting in fewer grid points which need to be computed, and hence less time spent performing processing. However, with the size of each panel's boundary region remaining the same, the ratio between computation and communication inevitably becomes smaller.

Secondly, an increase in the number of nodes also increases the likelihood that some required boundary values will not be available between timesteps, resulting in additional time a node must spend searching for or waiting for the values to become available. These factors combine to produce a disappointing level of efficiency during the parallel phase of execution.

However, it can also be seen that an increase in grid size generally results in increased efficiency, something particularly apparent for systems with six or less nodes. This is due to an increased grid size resulting in a linear increase in boundary size along with an exponential

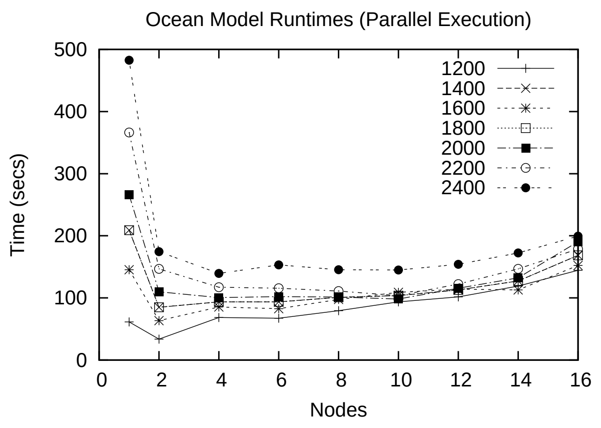

Figure 7: Parallel execution runtimes of the ocean model.

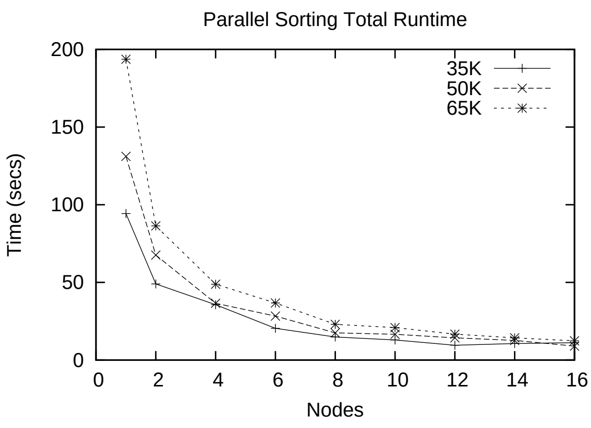

Figure 8: Total runtimes of the parallel sorting application.

increase in the number of grid points being computed. Whilst it would have been an interesting exercise to experiment with grid sizes greater than 2400x2400, it was at this point that the cluster nodes began to need to use virtual memory, which artificially effected the results.

7.3.2 Ocean Model Summary

The results of the ocean model's performance allow us to conclude the following:

The Tupleware system does provide the application with an overall speedup gain by distributing the application and processing it in parallel. However, the level of speedup is limited, with the best result being experienced with the largest grid size used.

A significant part of this speedup gain is due to the decrease in time taken to perform the beginning and end sequential stages of the application's execution. This is due to the increased efficiency of network data transfer and, as the number of nodes is increased, smaller total panel sizes. Thus, the runtime of this sequential phase is not fixed, despite it being reliant on the single worker node.

The parallel phase of execution also provides a limited level of speedup, and this is due to smaller panels requiring less time to compute. However, the increased time spent on network communications as the number of nodes grows cancels out these benefits.

The overall performance of the ocean model is to be expected given the application's characteristics, in particular its tightly-coupled nature and the fact that each node's execution is synchronised to a high degree with the nodes that are processing neighbouring panels.

Some encouragement can be taken from the fact that scalability tends to increase along with the problem size. Therefore we can conclude that the scalability would likely continue to improve if the grid size were increased to greater than 2400. However, the increase would need to be very significant, as the efficiency of this application clearly showed that the communications time significantly dominated the processing time of the application.

7.4 Modified Quicksort

Much like the ocean model, the sorting application was tested on a varying number of worker nodes, from one through to sixteen, with an additional master node to set up and finalise the application's execution.

7.4.1 Results

The total runtimes of the sorting application are presented in Figure 8.

As can be seen from the runtime for a single node system, sorting the array with the algorithm being used requires a significant amount of processing. Comparing this against a sixteen node system, we can see that the distribution and parallelisation of the application results in a substantial decrease in the overall runtime.

As a whole, the speedup experienced by the application is very pleasing. These speedup values can be found in Figure 9.

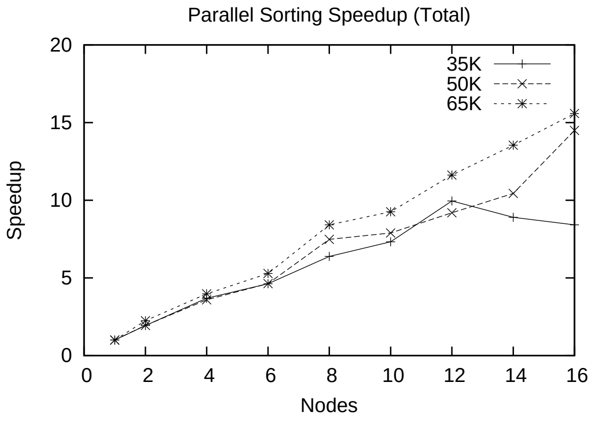

Figure 9: Total speedup of the sorting application.

In the instances of a thirty-five thousand threshold being used, the speedup peaks at 9.95 on twelve nodes before decreasing on fourteen and sixteen node systems. Nonetheless, this still provides a reduction in total runtime from 94.3 seconds to 9.5 seconds, a total decrease of 84.8 seconds. Considering that this threshold size is the smallest used for testing, and entails the greatest amount of network communication relative to other thresholds used, this is a pleasing result.

The two other threshold values used for testing gave a constant speedup up to sixteen nodes. In particular, for a threshold of sixty-five thousand, the speedup is near to optimal, and provides a total reduction in runtime of 181.3 seconds from 193.7 seconds on a single node system to 12.4 when sixteen nodes are used.

Parallel runtime is the sum total of the time spent performing network communications and processing once an initial unsorted array segment has been obtained. Network communications consist of obtaining additional unsorted segments once a worker's own storage in local tuple space has been exhausted, and also transferring sorted segments back to the master process. This will be affected by the threshold size: a smaller threshold requires more frequent communication with the master process, whereas a larger threshold requires less frequent. The speedup in terms of the parallel phase of execution is illustrated in Figure 10.

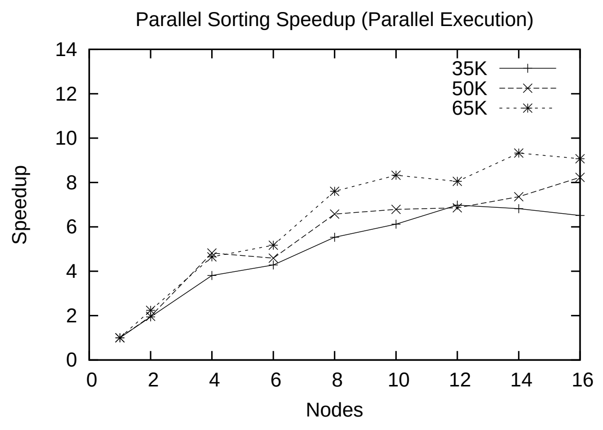

Figure 10: Speedup of parallel phase of sorting application execution.

As this shows, the speedup of the parallel phase of execution closely correlates to the speedup in terms of total runtime. For the smaller threshold value of thirty-five thousand, the speedup peaks at twelve nodes, however the speedup for the fifty thousand threshold continues through to sixteen nodes.

7.4.2 Modified Quicksort Summary

The results of the performance testing of the sorting application in the previous section have shown that the Tupleware system provides the application with significant performance gains and speedup. The speedup is most pronounced when a larger threshold is used. This is to be expected as increasing the threshold increases the granularity of the applications, increasing the ratio computation to communications time.

These results are very pleasing, and demonstrate that the system is able to provide speedup and performance gains for medium-grained applications. Based on Gustafson's Law [8], with the average communication time for each process remaining relatively constant as the number of nodes increases, while the total workload becomes larger, we would expect the application to continue to provide a high level of speedup as the number of nodes increases past sixteen. This prediction is further strengthened when we consider that increasing the problem size (via an increased threshold) actually greatly increases the efficiency of the system.

7.5 Performance Evaluation Summary

This section has presented the performance results of two applications: an ocean model and a parallel sorting application.

The findings of this performance evaluation were pleasing in terms of the sorting application, which displayed a high level of speedup on up to the maximum number of sixteen nodes, and was effective in evenly distributing the processing workload amongst all participating nodes in the system. The performance of this application also clearly illustrated the effect of varying the granularity of each processing task, with the larger threshold size exhibiting a higher degree of speedup than the smaller threshold sizes. This was due to the time each process spent on network communications remaining relatively constant, while the processing performed per process decreased as more nodes were added to the system. This is a result typical of an application such as this, and we can conclude that the Tupleware system has met its aim in this case of providing a scalable platform upon which to develop this style of medium-grained application.

In terms of the ocean model, the results show that the overall speedup gain was limited, and that as the number of nodes increased, the time each node spent performing network communication placed limiting factor to the continued scalability of the application. However, we also found that in increase in problem size, in this case the size of the grid, did not place a disproportionate load on any processes, and so there remains scope for the grid size to be increased further on a cluster with nodes with more than 1GB of memory.

8 Conclusions & Further Work

This paper has presented a dynamic, decentralised search function for the retrieval of tuples in a distributed system which contains multiple or a distributed tuple space. The search function optimises its behaviour based on the historical success or failure of previous tuple requests, which allows an accurate representation of the relative probability of remote nodes being able to fulfil a particular request to be formed by each individual node.

A further benefit is that the search function's logic is decentralised, without any need for sharing request data or global state information between nodes. If this was required, it would further add to the communication requirements of the system, and in turn lower the efficiency and performance of the applications.

Performance testing was carried out in order to determine the effectiveness of the search function, and the results presented in the previous section show that the system can provide performance gains for certain classes of distributed parallel applications, and that it can scale in terms of the number of nodes and also in terms of the problem size. While the performance of the tightly-coupled ocean model is not optimal, this is a common problem with tuple space-based systems, and the performance of Tupleware in this instance is comparatively good. It should be noted that the distribution of the tuple space in Tupleware in itself will introduce a certain amount of overhead, and yet this does not seem to cause the performance of applications running on Tupleware to suffer noticeably.

Further work on the system will entail implementing some sort of mechanism by which to measure the accuracy of tuple requests, so that we may compare the effectiveness of the search function being presented here to other alternatives.

Also some additional work is planned to address dynamic reconfiguration of the system at runtime. At the moment the number of participating nodes in the system must be known at compile-time in order to partition of array being processes and initialise the application. One of the strengths of the tuple space paradigm is its flexibility, and so it would be desirable to implement additional functionality to allow nodes to join and leave the system without interrupting to completion of the application being executed.

References

[1] Amdahl, G 1967, 'Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities', *AFIPS Conference Proceedings*, (30), pp. 483-485.

[2] Atkinson, A. and Malhotra, V 2004, 'Coalescing idle workstations as a multiprocessor system using Javaspaces and Java Web Start', *In: Eighth IASTED Intl. Conference on Internet and Multimedia Systems and Applications*, August 16-18, 2004, Kauai, Hawaii, USA.

[3] Atkinson, A 2008, 'A Distributed Tuple Space for Cluster Computing', *Proceedings of the Ninth In-*

ternational Conference on Parallel and Distributed Computing and Techniques, Dunedin, New Zealand, pp. 121-126.

[4] Carriero, N & Gelernter, D 1990, *How to Write Parallel Programs*, MIT Press, London.

[5] Charles, A et al. 2004, 'On the implementation of SwarmLinda'. *In ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference, pp. 297-298*, New York, NY, USA.

[6] Cormen, T. et al. 1999, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts.

[7] Gelernter, D 1985, 'Generative Communication in Linda', *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80-112.

[8] Gustafson, J 1988, 'Reevaluating Amdahl's law'. *Communications of the ACM,* vol 31, no 5, pp. 532-533.

[9] Hoare, CAR 1961, 'Algorithm 64: Quicksort', *Communications of the ACM*, vol 4, no 7.

[10] JavaSpaces^{TM}Service Specification, 2003, Sun Microsystems, California.

[11] Menezes, R and Tolksdorf, R 2003, 'A new approach to scalable Linda-systems based on swarms', *Proceedings of the 18th Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA.

[12] Merrick, I. and A. Wood 2000, 'Coordination with scopes', *In SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, New York, NY, USA, pp. 210-217.

[13] Merrick, I 2003, 'Scope-based coordination for open systems', PhD thesis, University of York.

[14] Nielsen B & Slrensen T, 1994, 'Distributed Programming with Multiple Tuple Space Linda', Masters Thesis, Aalborg University, Denmark.

[15] Noble, M. S. and Zlateva, S. 2001, *Scientific Computation with Javaspaces*. Lecture Notes in Computer Science 2110, 657-667

[16] Rowstron, Antony I. T., and Alan Wood. 1996. 'An Efficient Distributed Tuple Space Implementation for Networks of Workstations'. *In Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume 1*, pp. 510-513.

[17] Rowstron, A 1998, WCL: A Coordination Language to Geographically Distributed Agents, World Wide Web Journal, Volume 1, Issue 3, pp. 167-179.

[18] Wells, GC et al. 2004, 'Linda implementations in Java for concurrent systems: Research Articles', *Concurrent Computing: Practice and Experience*, vol 16, no 10, pp. 1005-1022.