Tupleware: A Distributed Tuple Space for Cluster Computing

Alistair Atkinson
School of Computing, Faculty of Business & Computing
Eastern Institute of Technology, Hawke’s Bay, New Zealand
aatkinson@eit.ac.nz

Abstract

This paper presents Tupleware, a cluster middleware which provides a distributed tuple space intended for use by computationally intensive scientific and numerical applications. It aims to add no extra burden to the application programmer due to the distribution of the tuple space, and uses a decentralised approach and intelligent tuple search and retrieval to provide a scalable and efficient execution environment. Tupleware is evaluated using two applications: a modified quicksort and an ocean model, which demonstrates good scalability and a low system overhead.

1 Introduction

The middleware presented in this paper, Tupleware, implements a distributed tuple space which provides a scalable platform for the development and execution of large, computationally intensive scientific and numerical applications. The tuple space paradigm of parallel programming is well established [1][9], as is its use as the foundation of distributed systems due to the success of systems such as JavaSpaces [4]. The benefits of the tuple space paradigm are its logical decoupling of cooperating parallel processes, its temporal as well as spatial distribution, and its clear separation in the program between computation and coordination.

Tuple space systems have long been recognised as being an effective tools for parallelising coarse-grained applications where there are very few dependencies between each parallel task [10]. Such applications tend to require less communication, and therefore do not suffer from the overhead introduced by transmitting data over the network, and from the computer hosting tuple space having to service requests.

However, this overhead imposes a serious limit to the scalability of a distributed tuple space system when it is used to run applications which are more finely-grained, and as such have greater dependencies between tasks and more frequent communication. Inevitably, a centralised tuple space implementation will become overwhelmed trying to service requests as the number of processes in the system and/or communication frequency becomes large. Therefore, it is desirable to decentralise the tuple space, and to spread the load of servicing requests between those computers which host each part of the distributed space.

Distribution of the tuple space presents several challenges which must be addressed in order to maintain, and hopefully improve, the performance of parallel applications which utilise it. Namely, the time taken to search for and retrieve tuples from the space must not increase too dramatically compared to a single centralised space. To this end, tuples need to be stored in the distributed space in such a way that requests are, as evenly as possible, spread amongst the computers which host each part of the space.

Tupleware addresses these issues using a decentralised approach along with intelligent tuple search. Additionally, it aims to not place any extra burden on the application programmer due to the distribution of tuple space. As far as is practicable, the distribution of the tuple space should be transparent to the programmer, and behave exactly like a global tuple space.

2 Background

2.1 Linda

Gelernter’s original coordination language, Linda [1], introduced a new paradigm for parallel program development. Linda was distinct from other techniques such as message passing and shared memory due mainly to its use of generative communication and distributed data structures, and also for its attempt to separate the coordination of parallel programs from their computation.

2.1.1 Concepts

The fundamental concept of Linda is its idea of a tuple space. This is a globally accessible, persistent area of stor-
age used to store data objects, or tuples. A tuple is an ordered list of typed data.

Tuple space acts as an associative shared memory, via which processes in a parallel system may communicate, store data and coordinate their actions. Unlike other shared memory systems, data are not explicitly referenced or named. Rather, tuples in tuple space are addressed by their content, and accessed via associative lookup. This process involves the use of a tuple template, in which some of the values of the desired tuple are specified, and others are left empty (empty fields are usually denoted by nil or null values). A template is said to match a tuple provided the following two conditions are true: a) the template is the same length as the tuple, and b) any values specified in the template are equal to the tuple’s values in their corresponding fields.

2.1.2 Operations

Linda in itself is not a complete programming language, but rather a small set of operations which are intended to be added to an existing language in order to create a parallel programming environment. These operations allows processes to interact with tuple space, and by extension, other processes.

Linda consists of six operations: out(), rd(), in(), rdp(), inp() and eval(). Each operation’s functionality is briefly described below.

- out() places a new tuple into tuple space,
- in() retrieves a tuple from tuple space which matches the given tuple template, permanently removing it from the space. If a matching tuple does not exist, then the operation blocks until one is placed into the space,
- rd() behaves exactly like in(), except that it retrieves a copy of a matching tuple rather than removing the tuple permanently,
- inp() and rdp() are non-blocking variants of in() and rd() respectively, returning a null value if a matching tuple cannot be found,
- Finally, eval() is used to spawn a new process. This operation, though included in the original Linda model, tends to be excluded from most other implementations in favour of more traditional process creation mechanisms, usually largely influenced by the host implementation language.

2.2 A Distributed Tuple Space

Section 2.1 provided an overview of the operation of Linda’s global tuple space. As outlined in Section 1, traditional global tuple space implementations can experience scalability issues which limit performance gains.

This scalability issues has most often been addressed by adding extra tuple spaces to the system, the idea being that the load of requests can be spread amongst each individual tuple space and therefore increase peak performance. For example, in a JavaSpaces system, it is straightforward to start additional JavaSpace services, creating a system multiple instances of these services. However, the major drawback of this approach is that each of these services operate completely independently to each other. They do not coordinate their actions to provide optimal servicing of client requests, and therefore it becomes the programmer’s responsibility to add extra functionality to the application in order to utilise the additional services.

A more beneficial approach to this situation would be to aggregate all of the JavaSpace services into a single, distributed space. This distributed space would appear to the programmer to behave as though it were a single, centralised space; that is, the distribution of the space should be completely transparent at the application level. A distributed space such as this would result in less complexity in the application code, whilst still providing a scalable platform for the execution of distributed applications.

There have been quite a few implementations of the Linda model which have attempt to address similar problems. Rowstron’s WCL [5] features a runtime system which supports multiple tuples spaces in a wide area network setting. WCL also introduced bulk primitives to the Linda model, which allowed multiple tuples to be moved or copied from on tuple space to another.

Another notable approach to this problem is Swarm-Linda [6]. This system used techniques based on the swarm intelligence of ant colonies to optimise tuple search and retrieval. However, whilst it is a promising approach, there are no published results as to its performance for real-world applications.

Linda in a Mobile Environment (LIME) [8] is a distributed tuple space designed for system with physically mobile devices and/or logically mobile software agents. It uses the concept of a federated tuple space, whereby hosts which have network connectivity merge each of their individual local tuple spaces into a single space accessible by all participating hosts. However, LIME is focussed on maximising availability of data in the situations described above rather than on improving performance. Nonetheless, the techniques it uses for dynamically integrating multiple tuple spaces are informative, and have some similarity to those used by Tupleware.

Overall, there seems to be a dearth of applied research into distributed tuple spaces in cluster environments. Many theoretical models exist, yet there are far fewer concrete implementations which apply these theories to real world
applications. This is what has motivated this particular research.

3 System Architecture

Tupleware has a modular design which exhibits a high level of flexibility in terms of on which node a particular component executes. The system itself is a low-level framework upon which distributed applications may be implemented and executed.

This section describes the main components in the system, and details the general operation of the system for typical applications. Tupleware is implemented using Java 6 Standard Edition, and all communication is carried out using TCP sockets.

3.1 Tuple Space Services & Stubs

At the lowest level, the system consists of the components which implement the functionality of a tuple space service and the stub objects which are used to communicate with them. The relationship between these stub and service objects shares similarities to that of stubs and remote objects in Java RMI, in that the stub object provides an interface for interacting with the service. This interaction generally involves sending messages across the network, specifically the sending and receiving of objects via Java object I/O streams. The stub object’s interface provides the common tuple space operations such as out(), in() and rd(), and the network communications are carried out transparently when these methods are invoked.

3.2 Runtime System

Tupleware’s runtime system implements the core logical functionality of the system. It provides the API used by applications to interact with the underlying system, and also implements the algorithms used for tuple search and retrieval. Every process in the system has exactly one instance of the Tupleware runtime, which handles that process’ local tuple storage, and retrieves required tuples from other processes. This layered approach is illustrated in Figure 1. Further detail on the operation of the Tupleware runtime system is provided in Section 3.4.

3.3 Master & Worker Processes

The Master is the process which submits a task to be executed by the system. Briefly summarised, its role is to submit a task to be executed by the Worker processes, and to reassemble the results of the computation once it has been completed. The role of Worker processes is to take this submitted task, and to execute the task in parallel with other Workers processes in the system.

The definition of the Master and Worker processes is the only responsibility of the application programmer; the remaining components of the system are generalised in such a way that any suitably implemented Master/Worker process combination may use them without modification. The operation of Tupleware is covered in further detail in Section 3.4 below.

3.4 Execution Environment

The components described in Section 3 combine to make up a complete distributed system capable of executing a parallel application. These components interact by communicating across the network to implement the algorithms which form the basis of the system. The operation of each of the components which comprise the middleware are described in the sections which follow.

3.4.1 Initialisation

Initialisation of a Tupleware application requires all participating Worker processes to register with the Master process. The registration process involves all Worker processes making their IP address and TCP port available in the Master process’ tuple space in order to facilitate the discovery of all other Worker processes that exist in the system.

Once the initialisation phase is complete, all Worker processes are able to connect to and communicate with each other, and with the Master process.

3.4.2 Task Submission

After initialisation, Worker processes will wait until the Master submits some task to be executed, and provides any required initial data. Sine the Master and Worker processes are implemented as two logical cooperating parts of the application, the precise format of this task submission may vary widely. Generally, the Master will define how the application is to be parallelised, and assign each Worker a particular role in the application’s execution.
Once the Master has submitted a task to be executed and each Worker has retrieved the initial data, then the system moves into the execution phase.

### 3.4.3 Execution

The execution of an application involves the Worker processes carrying out whichever task they may have been assigned, and communicating intermediate data with other Worker processes as required. All communication during the execution phase is carried out in decentralised fashion. That is, Worker processes communicate directly with each other, without requiring any network traffic to pass through the Master process.

All instances of communication during the execution phase involve a Worker making a request for a tuple which is not stored in its own local space. When a Worker requires a tuple, it will first search its own local portion of the tuple space. If it is found, then the process will continue; however, if the tuple is not available locally, then the spaces of other remote Worker processes must be queried.

The algorithm used to perform this remote search is based on the success factor of previous requests. As discussed in Section 3.1, each runtime has a collection of stub objects, containing one stub for each remote Worker process in the system. This collection takes the form of an array, and the ordering of the stub objects in this array determines the search order of the remote tuple spaces. When a Worker needs to search for a remote tuple, it will first query the remote Worker represented by the first stub in the array. If the request is unsuccessful, it will resend the query using the second stub, and so on, until either the request is successful or all stubs are tried unsuccessfully. Note that this initial round of requests are non-blocking, despite whether the actual operation (i.e., inp() or inp()) is a blocking or non-blocking operation.

In the instances where a request is eventually successful, the stub which was used to perform this successful request is inserted at the beginning of the array, and all other elements are moved one step toward the end of the array. In this way, over time, the remote processes with which a Worker most often needs to communicate will be among the ones queried first, followed by those less likely to contain the required tuple. This algorithm, while simple, is an ideal solution for applications such as the ocean model presented in Section 4.2, where a Worker will frequently communicate with a small set of “neighbouring” processes, and rarely, if ever, with all others.

In the instances where this sequence of requests does not return the desired tuple, the runtime system will perform one of two options, depending on the semantics of the operation being implemented. If the operation being carried out is a non-blocking operation, such as rdp(), then a null value will simply be return to signify that the requests tuple is not available. If, however, the operation is one of the blocking operations, such as rd(), then in order to conform with the semantics of these operations it is necessary to persist with the request until it can be fulfilled. Therefore, in these cases the runtime will send blocking requests to all remote processes in the system, and wait until a matching tuple becomes available, at which point all other requests are terminated and the retrieved value is return to the application.

### 3.4.4 Result Gathering

When an application has completed execution, the final data must be returned to the Master process. This involves each Worker process writing its result data back to the Master’s local tuple space. It is then the Master’s responsibility to merge these data back into a meaningful whole result.

### 4 Applications

Two applications were implemented to run on the middleware: a modified quicksort and an ocean model simulation. These applications are both ideally suited to Tupleware, as they both involve parallel array processing. Also, they have some contrasting characteristics which make them appropriate choices for performing a thorough performance evaluation.

#### 4.1 Modified Quicksort

Quicksort [2] is a well known sorting algorithm with an average case execution time of \( O(n \log n) \). It is recognised as an efficient general-purpose sorting algorithm which rarely exhibits its worst-case execution time.

Several characteristics of the quicksort algorithm lend it to being suited for evaluating Tupleware. Firstly, parallelisation of quicksort is reasonably straightforward, and produces a tasks which are loosely-coupled and have minimal 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 quicksort. 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 partitioned only until their length is greater than 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 [2].
4.2 Ocean Model

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

The body of water is represented by the model in the form of a grid, and each cell in the grid represents a single unit of computation. 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 universal and static across the grid. When executed, the model iterates through a fixed number of time-steps; at each time-step, each grid point’s surface elevation and current velocity values are recalculated based on the values stored at neighbouring grid points. This process continues for the specified number of time-steps.

The model is parallelised by performing domain decomposition on 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 boundary update is relatively communications intensive, and as such must be carried out as efficiently as possible, and the size of each boundary has a significant effect on the scalability of the application.

As in the modified quicksort application, the granularity and degree of parallelism of the ocean model can be altered by varying the size of the grid and the relative size of each panel.

5 Evaluation

The system’s performance was evaluated, using the two applications discussed, on a cluster of sixteen Pentium 3 (800MHz) PCs with 256MB of RAM, running Linux (kernel 2.6). Nodes were connected by a 100Mbps Ethernet network on its own segment. Performance profiling was carried out using the Clarkware Profiler [3].

5.1 Modified Quicksort

The total execution times of the modified quicksort application for sorting an array of twenty million elements and a threshold value of fifty thousand are shown in Figure 2. This array and threshold size was chosen due to it being one of the largest arrays and smallest thresholds tested. Arrays larger than this caused the cluster nodes to begin thrashing, artificially effecting the results.

As this chart clearly shows, speedup for the modified quicksort as the number of nodes increase is near to optimal. Such a result is not unexpected for a reasonably coarse-grained application such as this. It is still pleasing, however, due to the fact that the application does required the sharing of some intermediate data between processes, and this does not seem to have an adverse effect on its speedup.

5.2 Ocean Model

As discussed in Section 4.2, the ocean modelling application is must more communications intensive than the modified quicksort. Due to this characteristic, it is not surprising that it exhibits some limitations to its scalability as the number of nodes and/or the size of the grid increases. Nonetheless, the ocean model is a good example of the difficulties associated with parallelising a tightly-coupled application, and is, in the author’s opinion, a good test of how Tupleware handles the worst case scenario of frequent communication requirements.

This scenario is well illustrated by the data presented in Table 1, which details the execution of the ocean model on a 500x500 grid on varying numbers of nodes. As shown, the time spent performing communication dominates the time spent performing processing, and increases with the number of nodes. This is to be expected, as increasing the number of nodes in the system has two consequences: decreased panel size and an increased total number of boundary points. These factors combine to increase communication overhead while the amount of processing required of each per time-step is reduced.

However, a more representative measure of the actual system overhead introduced by the Tupleware system can be shown by analysing how varying the total grid size ef-
Table 1. Ocean model runtimes (in milliseconds) for a 500x500 grid.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Communication</th>
<th>Processing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18158</td>
<td>46574</td>
<td>64732</td>
</tr>
<tr>
<td>4</td>
<td>107965</td>
<td>5978</td>
<td>113943</td>
</tr>
<tr>
<td>9</td>
<td>127364</td>
<td>2538</td>
<td>129902</td>
</tr>
<tr>
<td>12</td>
<td>233887</td>
<td>2156</td>
<td>236043</td>
</tr>
</tbody>
</table>

Table 1. Ocean model runtimes (in milliseconds) for a 500x500 grid.

flects the communications overhead for a fixed number of nodes. Considering the scenario with the maximum communications overhead, a system with twelve Worker nodes, the graph shown in Figure 3 demonstrates how the total execution time is effected by different sized grids. This graph shows that the increase in communications overhead is approximately proportional to the increase in grid size, which would suggest that the scalability issues are mainly due to the nature of the application rather than the underlying system itself.

6 Conclusions & Further Work

The system presented in this paper, Tupleware, implements a distributed tuple space which provides efficient performance for parallel numerical applications executed in a cluster computing environment. The results presented are consistent with what one would expect from a distributed application given the nature of the two applications used for the evaluation.

On a qualitative level, the system does not burden the application programmer with any additional barriers compared to a traditional centralised tuple space system. The distribution of tuple space is, as far as is practicable, transparent to the programmer.

Possibilities for further work include continued enhancements to the algorithmic foundations of the runtime system for tuple search and retrieval. In particular, a publish/subscribe style of communication could conceivably reduce the time each process spends waiting for required data in timesteped applications such as the ocean model. Another useful addition to the system would be to introduce dynamic reconfigurability so that Worker processes could be added to or subtracted from the system during execution, without interfering with the correct execution of the system.

Acknowledgments

This research was partially supported by an Australian Postgraduate Award scholarship. The cluster used to develop and evaluate Tupleware was provided by the School of Computing at the University of Tasmania.

References