Proceedings of the Eighth IASTED International Conference
INTERNET AND MULTIMEDIA SYSTEMS AND APPLICATIONS
August 16-18, 2004, Kauai, Hawaii, USA

COALESCING IDLE WORKSTATIONS AS A MULTIPROCESSOR SYSTEM USING JAVASPACES AND JAVA WEB START

Alistair Atkinson and Vishv Malhotra
School of Computing, University of Tasmania
Private Bag 100, Hobart, Tasmania 7001
{alatkins, vishv.malhotra}@utas.edu.au

ABSTRACT
This paper describes a distributed system which aggregates the unused and usually wasted processing capacity of idle workstations. The aggregation is achieved through the use of now ubiquitous internet infrastructure and web technology. And, it delivers a powerful yet inexpensive execution environment for computationally intensive applications. The prototype system described here makes use of Sun Microsystems Jini technology, particularly JavaSpaces, along with Java Web Start, to produce a dynamic, flexible and reliable system. Two example applications used to evaluate the system are described: (a) the n-Queens problem and (b) a parallel sorting (shearsort) application. The results of the evaluation clearly show that, for certain classes of applications, the system is capable of delivering significant performance.

KEY WORDS
Parallel and Distributed Processing, Java-based web applications, Jini, JavaSpaces.

1. Introduction

A large proportion of an average organization's workstations spend the majority of their time either unused or relatively idle. If the wasted processing capacity of these workstations could be aggregated, it could be used to provide processing for computationally intensive applications. Such a system could potentially provide an inexpensive alternative to costly custom built parallel computers or clusters. Many Networks of Workstations (NoW) systems have been designed and reported in the past to take advantage of available resources over the periods of low computer demand – for example, during off-peak night periods – in the organizations. Arguably the most well known example of such a system is SETI@Home [1].

Many of these organizations have a large number of workstations, which are often under-utilized even during the 'active' periods. These workstations, even when in use, may only require a small fraction of the total processing power of the machine. If all of this wasted processing capability could be aggregated, the resultant computing power could potentially be substantial and available at the times when the organization is active.

Computational resources can not be quickly moved from a physical location to another. Nor is it possible to easily shift demand for computational resources in an organization from a time period to another period without significant cost implications. Typically, organizations respond to these pressures by installing computational resources in each work center, to match the peak demand for the location. Needless to say, that this leads to gross over-installation of the computers in many organizations. Ubiquitous internet infrastructure that inter-connects virtually every computer today provides an opportunity to quickly and dynamically match organizational centers experiencing higher demands with those with computational capacities to spare. Furthermore, the cost of doing so would be vastly less than the traditional approaches, as the needed infrastructure is already in place, and all that is required is appropriate software to put it to use.

The prototype system presented in this paper attempts to achieve this goal. There are two important aspects of the system: a distributed environment to which users can submit jobs for execution, and a framework which can be used to build distributed applications suitable for deployment on the system. The distributed environment is based on the master/worker distributed system architecture. This allows large, computationally-intensive tasks to be divided up into smaller subtasks and distributed out to worker computers, in this case idle workstations, for processing. The development framework enables the development of applications for the system, and also provides generic coordination mechanisms which are capable of controlling the execution of reasonably complex applications.

A key characteristic of the system is that its processing capacity is not fixed; computers can be dynamically added and removed from the system at any time, thereby dynamically changing the overall processing capability. This is in contrast to custom-built machines or clusters, which often require costly upgrades in order to gain a significant increase in its processing capacity. These upgrades may also require a complete system shutdown, during which time no processing could be performed.

The system is evaluated using two sample applications: the n-Queens problem, and a parallel sorting (shearsort) application.

In Section 2, we briefly introduce the background technologies the proposed system uses. The section also contains references to some related works. The system is then introduced and described in Section 3. In Section 4, we describe the two example applications used for testing the effectiveness of the system. These applications have relatively contrasting characteristics, which highlight several important properties of the system. The performance results are given the following section, Section 5. The paper concludes in Section 6 with some comments and suggestions for further work.

2. Background: Distributed Processing, JavaSpaces, Web Start

Distributed systems are a valid and inexpensive alternative to supercomputers or dedicated clusters for meeting the processing demands of computationally intensive applications. This is achieved by taking the large task that is to be completed, and dividing it into smaller sub-tasks. These sub-tasks can then be distributed out to individual computers for processing in parallel. Many distributed computing technologies have been developed, the most relevant in terms of the system being presented being the Linda Coordination Language [2] and Sun Microsystems' Jini [3], in particular JavaSpaces [4].

A distributed system runs by placing its subsystems (or components) on different computers for execution. Beside the speed advantage, distribution may benefit its users by providing better response, security and reliability. On the other hand, such an approach requires mechanisms to communicate data between the components, their synchronization and scheduling. Shared memory and message passing are two common approaches to model the communication and synchronization needs of the concurrent systems. Linda uses a different approach to these goals.

The Linda Coordination Language was developed by David Gelernter at Yale University in 1985. The approach is distinguished from other distributed system technologies by its use of a *space-based* communication medium rather than the traditional shared-memory or message-passing medium. Processes in the system communicate indirectly by reading and writing tuples to and from a shared tuple space accessible to all components of the distributed system. Three fundamental operations supported by a tuple space are: get a tuple matching the specified attribute, put a tuple into the tuple space, and remove a tuple. This concept forms the basis of the JavaSpaces distributed system technology from Sun Microsystems. JavaSpace stores objects rather than the tuples, as in Linda.

JavaSpaces is a distributed computing technology that provides a persistent shared object store and simple yet powerful object exchange mechanisms. The design of JavaSpaces was heavily influenced by the Linda Coordination Language. The JavaSpaces shared store is provided by a service called a space. A space is used to hold entries (Java objects). A space has some useful properties, namely persistence, and the ability to be associatively searched. Entries stored in a space remain there until they are explicitly removed or their lease expires, even if the process that placed the object there ceases to exist. Furthermore, it is possible to locate objects stored within a space by performing a search based on an object's type and values of its attributes.

Java Web Start [5] is a technology that allows standalone Java applications to be deployed over the World Wide Web (WWW). A Web Start application can be launched simply by clicking on a hyperlink embedded in a web page, and spares the user from having to install and configure the software manually. Web Start applications offer many advantages over Java Applets, as they are not tied to the browser after being launched, and don't have be repeatedly downloaded the code each time the user runs the program.

In the following section we describe how these technologies are used to develop a system for the distributed execution of programs.

3. Distributed Execution Environment: Coalescing Idle Workstations

There are three main components of the proposed Distributed Execution Environment (DEE) constructed by coalescing idle workstations: (1) Identifying the available computer; (2) Constructing a distributed system by using the available computers; and (3) Programming environment for developing the application programs.

Identification of available computer systems is based on a simple voluntary step. An owner of a computer system could donate the computer for use as a node in the distributed system by simply clicking on a hyperlink on a web page which would launch the worker Web Start application. This step is entirely voluntary and a donated computer similarly can be removed from the distributed system by exiting the worker application. The distributed environment enables a user to donate their computer to the system, effectively adding their computer's processing capability to a pool of other worker computers. These workers are used to collectively process large, computationally intensive applications.

The design of the distributed system follows the master/worker, or agenda parallelism design suggested by Carriero & Gelernter [6]. In such a system, a master process will divide a large task into multiple smaller sub-tasks. These sub-tasks are distributed out to however many worker processes are available. The worker processes will execute the sub-task, and return the results of the computation back to the master. Once all of the sub-tasks have been executed and the results returned, the master will merge the results into a meaningful result of the original large task. An arrangement such as this is shown in Figure 1.

The programming environment relies on suitably constructed Java class. A generic task executor is started on the donated machine. It contacts the JavaSpace store to find next problem specific task to execute. The problem task is loaded. This task in turn will obtain the necessary input data from JavaSpace. On completion, the results of the task are stored into the JavaSpace for access by follow-up tasks. Clearly, one needs to coordination the activities of these tasks to ensure that the tasks can find their relevant input data and store their outputs for the follow-on tasks. There is little benefit in attempting to run a task which has some of its input data unavailable. The coordination is achieved by using a *dependency graph*.

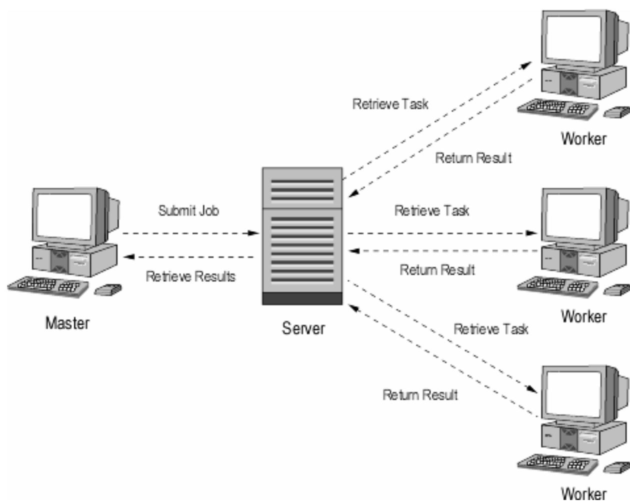

Figure 1: System overview.

The activities of each worker process in the system are coordination by a shared *dependency graph*. The dependency graph is a JavaSpace entry that stores the current state of an application's execution, and is used by workers to determine which task should be executed next. The graph itself is a directed acyclic graph, with vertices representing an application's tasks, and the edges denoting the data dependencies between them.

So, before a worker obtains a task for execution, it first takes the dependency graph from the space to see which task it should execute. The worker then takes this task from the space, and marks the corresponding node in the dependency graph as being *in-progress*. Also, a worker will use the graph to determine if the task depends on the results of any previously executed task, and, if so, will obtain these results before executing the current task. When a worker has completed the execution of a task, it will obtain the dependency graph and mark the node as *complete*, before returning the results of the task's execution to the space. Workers continue this process until all nodes in the graph are marked as complete, at which time the master process takes all of the results from the space and assembles them into some meaningful whole, depending on the particular application.

The system will continue to function correctly and maintain a consistent state in the event of partial failure, due to the use of the Jini Transaction Manager. All objects taken from or written to the JavaSpace are done so under a transaction. All transactions are leased from the transaction manager for a given period of time; this lease time is specific when the transaction is created. If a transaction's lease expires, the transaction is automatically aborted, thus canceling all operations performed under it. The use of transactions becomes especially important in cases where a worker process fails for some reason after having acquired some object from the JavaSpace. Instead of these objects being lost indefinitely, they are instead only rendered unavailable until the expiry of the transaction under which they were taken from the JavaSpace, at which time they will be made available once more to other worker processes. The use of transactions also gives the system the added advantage of preventing potential deadlocks from occurring, as objects are only acquired for a finite amount of time.

4. Example Applications

Two example applications were used to evaluate the system: the n-Queens problem and a shearsort application. These applications have some widely contrasting characteristics which are useful in assessing various aspects of the system.

4.1 n-Queens Problem

The n-queens problem [7] is a generalization of the well known eight-queens problem. The basic premise of the problem is this: find the number of ways that n-queens can be arranged on an n x n chess board so that no two queens can attack each other, according to the rules of the game.

This particular implementation of this combinatorial problem uses a simple backtracking algorithm. This algorithm has an exponential execution time, which makes it one of the least efficient methods of solving the problem. However it is also one of the most straightforward algorithms to parallelize.

The problem is parallelized by dividing the entire space that needs to be searched, in this case all possible combinations of queens placings on the chess board, into smaller sub-tasks that will each search a subset of the space. This is achieved by placing a queen in each of the first two columns of the board, in positions where they cannot attack each other. This produces $(n\text{-}1) \times (n\text{-}2)$ sub-tasks of approximately equal size, given a board of size n, with each sub-task involving a search of n^{n-2} possible combinations. The execution of the sub-task will result in every possible combination of queens placings being searched for the remaining columns, and testing if each combination is a valid solution to the problem, given the position of the queens in the first two columns.

4.2 Shearsort

Shearsort [8] is a parallel sorting algorithm that we have adapted for a distributed environment. The shear sort algorithm takes a list of *n* numbers that are to be sorted, and arranges them into a two dimensional mesh of size $n^{1/2} \times n^{1/2}$ Once this has been done, exactly $\lceil n^{1/2} \rceil$ phases of execution will be completed. If an odd-numbered phase of execution is being completed, each odd- and even-numbered row will be sorted in ascending and descending order respectively. If an even-numbered phase is being completed, then all columns are sorted in ascending order. After all phases have been completed, the mesh will be sorted. Pseudo-code for the shear sort algorithm is shown below.

```
for step = 1 to ceiling(sqrt(N)) do
   if odd(step)
       if odd(row)
      sort_left_to_right(row);
       else
      sort_right_to_left(row);
    else
        sort_top_to_bottom(col);
```

The shear sort algorithm has an average execution time of $n^{1/2}$ when executed on $n^{1/2}$ processors. It can be parallelized by sorting each row or column in parallel, depending on the phase of execution.

5. Performance Evaluation

The two example applications discussed in Section 4 were used to assess the system to see whether it met its goal of utilizing the wasted processor cycles of idle workstations.

All testing was conducted using the computer labs in the School of Computing, Launceston. All required Jini services were set to run on an 800 MHz Pentium 3 machine with 256 MB of RAM, and running Slackware GNU/Linux 9.0. Also running on this machine was the master process of an application, along with the Apache web server, which was used to service requests for the client Web Start application.

The rest of the system consisted of the client (worker) software running on numerous independent computers. Unless otherwise stated, these worker machines were 400 MHz G4 Apple Mac machines with 512 MB of RAM, and running Mac OS X. All computers used for testing were connected via a 100 Mb switched fast Ethernet network. Also, all computers were using the Java Runtime Environment 1.4.2.

5.1 Performance Measurement

The usual measures of the performance of a parallel system, suggested by Carriero & Gelernter [2], are speedup and efficiency. Following their definitions, *speedup* is "the ratio of sequential run-time to parallel run time", and *efficiency* is "the ratio of speedup to number of processors". Efficiency can also be viewed as the average utilization of the system's total processing capacity. Both of these measures give a good indication of the effectiveness of a parallel system in using available processors to their maximum capacity. In particular, the efficiency of a system is an excellent indicator of a system's scalability. Parallel applications should therefore strive to maximize efficiency in order to achieve the highest possible speedup.

We will define the sequential run time of an application as the time it takes to execute when there is a single worker computer present in the system. This definition establishes a valid basis representing the full (100%) utilization of the workstation time. The speedup and efficiency for all other number of workers will be calculated based on this value.

Unlike the multi-workstation environment, a single workstation environment does not incur contention for resources (JavaSpace and the dependency graph being the primary resources). Further, a single workstation environment schedules the sub-tasks in a way that provides for full utilization of the donated workstation. In a multi-workstation environment some workstations may have to wait for their next sub-tasks as the sub-tasks wait for the enabling inputs. These waiting will impact the processor utilization (efficiency) of the system. A problem well suited to the environment will continue to provide high efficiency (workstation utilization) in the multi-workstation environment. Low workstation utilization as the number of workstations increase indicates a problem ill-suited for the system.

5.2 n-Queens Problem

Testing was carried out for the n-queens problem, with a value of n equal to sixteen (ie. we wish to place sixteen queens onto a chess board of dimension sixteen). This is a non-trivial problem, which requires the search of 16^{16} possible states. The process of dividing this job into sub-tasks will yield two-hundred and ten sub-tasks, each of which will search 16^{14} possible board states.

The performance results of the n-Queens problem, where *n* equals 16, are show in Table 1. This data clearly shows that the n-queens application achieves excellent speedup, made possible by its high level of efficiency on up to sixteen machines. The problem took almost two hours to compute using a single computer; however on sixteen machines the time was reduced to around seven minutes. These results are especially pleasing, as they indicate that doubling the amount of computers working on the problem will go very close to halving the execution time.

These positive results are mostly due to the problem being relatively coarse-grained, in that each sub-task requires the worker to do a significantly large amount of work. Also, the subtasks can be executed in parallel due to the absence of any data dependencies.

5.3 Shearsort

The shearsort application was firstly tested on a list of randomly-generated long integers. The results obtained from executions of the application are shown in Table 2.

Table 1: Performance results of n-Queens problem where n equals 16

Workers	*Average Run Time (secs)*	*Optimal Run Time (secs)*	*Speedup*	*Workstation Utilization*
1	6392.6	6392.6	1	100.00%
2	3190.8	3196.3	2	100.00%
3	2133.9	2130.9	2.99	99.67%
4	1620.4	1598.2	3.95	98.75%
6	1083.7	1065.4	5.9	98.33%
8	810.8	799.1	7.88	98.50%
12	552.5	532.7	11.97	96.42%
16	427.5	399.5	14.95	93.44%

Table 2: Performance results of shearsort.

Workers	*Average Run Time (secs)*	*Optimal Run Time (secs)*	*Speedup*	*Workstation Utilization*
1	2673.9	2673.9	1	100.00%
2	1334.2	1336.9	2	100.00%
4	1305.4	668.5	2.05	51.25%
8	1229.3	334.2	2.18	27.25%
12	1184.4	222.8	2.26	18.83%
16	1090.0	167.1	2.45	15.31%

These results show good speedup on two machines, followed by a dramatic leveling off on any additional machines. This coincides with a steep fall in the efficiency of the system. The efficiency of this application indicates that is does a very poor job of utilizing the workstations in the system.

The main contributing factor to this poor scalability is the very fine-grained parallelism of the application. A relatively large amount of JavaSpace operations must be carried out for each task; however for all of these communications, only a relatively small amount of actual work is performed by a task.

The CPU usage of the server machine during these tests was observed to be consistently between 80%-85% when there are two worker machines and 95%-100% when there are four or more workers in the system. This would suggest that the server is not able to service the large amount of JavaSpace operation in a timely manner, thus explaining the sudden leveling off of performance.

5.4 Shearsort with introduced delay

To determine whether the fine-grained nature of shearsort is indeed the cause of its poor performance, each task was programmed to sleep for five seconds during execution. This modified approach was then tested on a list of randomly-generated long integers. This had the overall effect of producing tasks that will take a greater amount of time to execute, effectively making the application more coarse-grained, and thereby decrease the communications and server CPU load. Note that this test was conducted using 700 MHz G4 Apple iMacs, with 384 MB of RAM and running Mac OS X. The performance results of this modified shearsort are shown in Table 3.

These results show a marked improvement on those previously presented in Table 2, suggesting that the fine-grained tasks of the shearsort application are indeed the cause of the poor speedup. It is clear that this coarser-grained approach is much more efficient than when a delay is not used.

Table 3: Performance results for shearsort with introduced delay.

Workers	*Average RunTime (secs)*	*Optimal Run Time (secs)*	*Speedup*	*Workstation Utilization*
1	670.2	670.2	1	100.00%
2	337.0	335.1	1.99	99.50%
4	240.9	167.6	2.78	69.50%
6	176.4	111.7	3.8	63.33%
8	151.4	83.8	4.43	55.38%

5.5 Discussion

The performance results presented in this section highlight how the contrasting characteristics of each application impact on the actual level of performance gained. The most influential factors are task granularity and sequential execution.

The most obvious contributing factor to the difference in scalability lies in the granularity of the sub-tasks that make up an application. In this case, the granularity should be thought of as the amount of work done compared to the amount of communication overheads (ie. JavaSpace operations) incurred during the execution of each task.

The shearsort application performs many more JavaSpace operations than the n-queens application, as it must read each cell of each row or column of the data mesh individually and also the step counter entry, in addition to the standard dependency graph and task entry. For all of this overhead, only a relatively small amount of work is actually performed.

The n-queens application is vastly different to shearsort, in that each sub-task does not need to fetch any data objects in order to execute. All data is encapsulated in the task entry itself, and the task execution involves heavy computation, which takes a reasonably significant period

to complete. This coarse grained parallelism results in excellent speedup and scalability.

The level of performance gain is inevitably associated with the level of inherent concurrency of an application; it is unlikely that a purely sequential program will achieve any speedup at all, most likely the opposite would prove to be true.

The n-queens application can be fully parallelized; every task entry can be executed in parallel if there are enough workers available. However the shearsort application must execute sequentially in part, due to its different sorting phases which alternate between sorting rows and columns. This means that every row or column must be finished being sorted before all other workers can continue on in the subsequent phase of execution. This problem would not be pronounced in a system where the workers each have approximately equal processing capability. However, in a scenario where there is one worker that is particularly slow, the performance could be seriously degraded as all of the faster workers would be continually waiting on this slow worker.

6. Conclusions & Further Work

This paper has presented a dynamic distributed system which aims to make use of the wasted processing capacity of idle workstations for the execution of computationally intensive tasks. Such a system could allow an organization much greater flexibility of its computing resources, and reduce costs by removing the need to have redundant resources at each work center simply to meet short periods of high demand.

The system is capable of operating in a heterogeneous computing environment, and allows workstations to dynamically join and leave the system at any time, even during the execution of an application. The use of transactions adds robustness to the system, and keeps it in a consistent state in the event of partial failure.

The system caters to dynamic arrivals and departures of the donated workstations well. A workstation that is removed while executing a sub-task only affects that particular sub-task. The transaction system would, in due course, reschedule the sub-task for execution on another workstation.

Among the future directions, we plan to expand the role of dependency graph to create support dynamic creation of new sub-tasks by the running sub-tasks. This could lead to a more efficient and flexible system.

Yet another area of interest is the addition of support for the use of multiple JavaSpaces to alleviate performance bottlenecks. The efficient and transparent use of multiple spaces is a topic of ongoing research interest [9, 10].

Finally, another possible area of further work involves reducing the level of intrusiveness of the worker software on a donor's computer. The software which is downloaded onto a donor computer will use all of the available CPU cycles. This may lead to noticeable performance degradation for a user which is using their machine whilst taking part in the system; this could also potentially discourage users from donating their computer to the system. This obviously undesirable situation could be avoided by monitoring the resource requirements of a user, and suspending processing during those times when they require all of the processing capacity of their machine.

References

[1] Search for Extraterrestrial Intelligence: SETI@Home, http://setiathome.ssl.berkeley.edu/ (accessed on 30th March 2004)

[2] D. Gelernter, Generative communication in Linda, *ACM Transactions on Programming Languages and Systems*, *7*(1), 1985, 80-112.

[3] Jini Network Technology, http://java.sun.com/developer/products/jini/ (accessed 30th March 2004)

[4] E. Freeman et al, JavaSpaces Principles, Patterns and Practice (Massachusetts: Addison-Wesley, 1999).

[5] Java Web Start Technolgy, http://java.sun.com/products/javawebstart/ (accessed 30th March 2004)

[6] N. Carriero & D. Gelernter, *How to write parallel programs* (London: MIT Press, 1990).

[7] The N Queens Problem, http://www.math.utah.edu/~alfeld/queens/queens.html (accessed 30th Match 2004)

[8] Shearsort, http://www.cs.mu.oz.au/498/notes/node35.html (accessed 30th March 2004)

[9] R. Menezes and R. Tolksdorf, A new approach to scalable Linda systems based on swarms, *Proceedings of the 2003 ACM symposium on Applied computing,* New York, NY, USA, 2003, 375-379.

[10] I. Merrick & A. Wood, Coordination with scopes, *Proceedings of the 2000 ACM symposium on Applied computing*, Como, Italy, 2000, 210-217.