Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Java PastSet: A Structured Distributed Shared Memory System

Kei Simon PEDERSEN and Brian VINTER
Department of Computer Science, University of Southern Denmark

Abstract. The architecture and performance of a Java implementation of a structured distributed shared memory system, PastSet, is described. The PastSet abstraction allows programmers to write applications that run efficiently on different architectures, from clusters to widely distributed systems. PastSet is a tuple-based three-dimensional structured distributed shared memory system, which provides the programmer with operations to perform causally ordered reads and writes of tuples to a virtual structured memory space called the PastSet. It has been shown that the original, native code, PastSet was able to outperform MPI and PVM when running real applications and we show that the design translates into Java so that Java PastSet is a qualified competitor to other cluster application programming interfaces for Java.

1 Introduction

Structured distributed shared memory, structured DSM, was first introduced in Linda [1], but the interest into structured DSM has since been low. The general consensus is that it is not possible to achieve performance comparable to that of simpler systems. We started the work on PastSet with the idea that bad performance is not inherent to the concept of structured DSM, and that good performance can be achieved by paying close attention to design and implementation.

The increasing popularity of Java have also spawned an interest in 'easy-to-use' application programming interfaces, APIs, for parallel and distributed applications, which in turn has sparked a renewed interest in tuple space based DSM since this model is truly simple to use. The result of various ventures into tuple space has resulted in a variety of Linda-like systems and two major, widespread, systems; JavaSpaces[2] and TSpaces[3]. JavaSpaces is the result of a collaboration between the original Linda team and SUN, while TSpaces is a product from IBM that seeks to address some of the semantic shortcomings found in JavaSpaces.

The Java PastSet, JPS, project seems like an obvious experiment based on the high performance we achieved with the original, native code, PastSet, and one of the primary goals of the JPS system were to test if the performance of PastSet were due to correct design or simply a result of very good programming. A secondary goal were naturally to provide yet-another tuple space DSM system for Java, hopefully far more efficient than the existing ones.

The remainder of this paper is organized as follows; in section 2 we describe the PastSet model in some detail, and section 3 explains the design of JPS. In section 4 we report on the basic performance of the JPS operations and in section 5 we show the application level performance of JPS. Finally we draw our conclusions in section 6.

2 PastSet

PastSet was first introduced as an interprocess communication paradigm in [4]. The paradigm resembles that of Linda[1], but with some significant differences. Tuples are generated dynamically based on tuple templates that may also be generated dynamically. A tuple template is an ordered set of types. Tuples based on a particular template has an ordered set of variables matching the types in the template. Each type in a tuple template has an associated value-space, or dimension, describing the set of all possible values for that type in this template. Taken together, the types in a template spawns a space encompassing all conceivable type-value combinations for tuples based on that template. A tuple with all singular values represents a singular point in this space.

As with Linda, PastSet supports writing (called move) tuples into tuple-space and reading (called observe) tuples that reside in tuple space. Contrary to Linda's in operation, PastSet observe does not remove tuples from tuple space, the tuple that is observed is marked as observed but remains in the PastSet so that it can be read again if specified so directly. No mechanism is provided to remove individual tuples from PastSet. All PastSet operations return only when the operation has completed, or an error has been detected, no asynchronous calls exist.

In PastSet, each set of tuples based on identical templates is denoted an element of PastSet. An element may be seen as representing a trace of interprocess communications in the multidimensional space spawned by the tuple template. PastSet preserves the causal order among all operations on tuples based on the same or identical templates. There is no ordering between tuples of different elements. Tuples that match the same type, but which the programmer does not wish to place in the same element can be differentiated by an initial flag.

In effect, PastSet keeps a causally ordered log of all tuples of the same or identical templates that has existed in the system. This also allows the processes to re-read previously read tuples.

It is the intention that the added semantics of PastSet will allow programmers to more easily create parallel programs that are not of the traditional 'bag of tasks' type.

Two pointers First and Last are associated with each element in PastSet. First refers to the elements oldest unobserved tuple. Last refers to the tuple most recently moved into the element. A parameter, DeltaValue, associated with each element in PastSet defines the maximum number of tuples allowed between First and Last for that element. A process may change DeltaValue at any time. The move and observe operators update First and Last, and obey the restrictions imposed by DeltaValue for each element in PastSet.

Functionality is provided to truncate PastSet on a per element basis, permanently removing all tuples that are older than a given tuple.

To address some of the performance problems for DSM systems, we have introduced the concept of "User Redefinable Memory Semantics", URMS[5], into PastSet. The principle behind URMS is to offer users the opportunity to redefine the semantics of any or all memory operations for memory areas that are specified by the user. The redefined semantics are specified by providing code that should be executed instead of the memory operation. The specification of the redefinition may also include initialization code that is applied once to initialize the specified memory area. In effect, the redefined memory operation semantics will be applied for memory operations on the specified areas only.

For example, a memory location may be redefined to accumulate a global sum of partial sums that are produced by independent processes. For that location, the Move operation would be redefined accordingly to code that stores the aggregate sum, and keeps track of a completion criterion that may be realized as an access count, process list, or by other means. The Observe operation is redefined to return the aggregate sum only after the reduction has

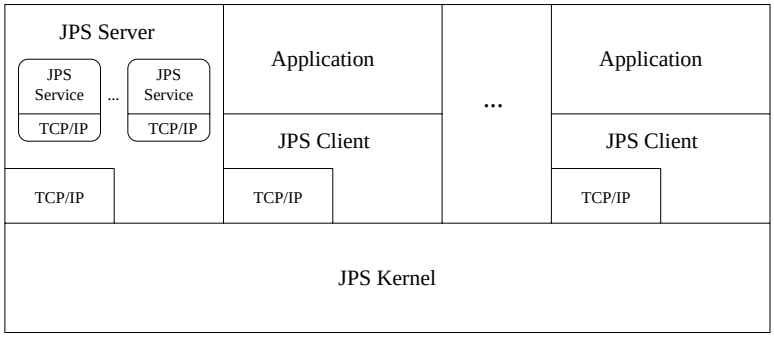

Figure 1: Layout of a node that supports PastSet

completed, in effect blocking until the termination criterion used for the Move operation is satisfied. This approach makes it very simple for programmers to overlap communication and calculation if there is any work that may be done between the partial sum is ready and the global sum is needed.

3 Java PastSet

3.1 Architecture

The Java version of PastSet is focused on achieving high performance, while preserving as much of the original PastSet semantics in an OO-environment. To minimize overhead the architecture comprises two parallel paths, one that executes PastSet operations that may be completed locally and one for operations that requires remote operations. Since the nature of PastSet focus on the elements of PastSet, the distribution also focus on the elements. Each element exists on one node only and is not replicated. An operation on elements that are located on the same node as the process that performs an operation is called a local operation. Operations on elements that are located on other nodes are called remote operations. Thus PastSet supports remote observes and remote moves, similar to remote read and remote write on other DSM systems. Remote operations are wrapped and communicated to a JPS Server on the remote node using TCP/IP.

3.2 Data Location

To achieve low overhead on operations it is imperative that applications can easily determine, at run time, whether an operation should execute locally or remotely, and in the latter case, which node should perform the remote operation. To this end each element descriptor object includes the identity of the server that holds the element. In the case of a local element the JPS Client calls the PastSet operation directly. For a remote element, the call performs a procedure, which sends the operation to the respective server and then reads and returns the answer.

3.3 The JPS Server

Each node in the cluster that holds parts of PastSet runs a JPS Server that services remotely issued PastSet operations on local elements. As the PastSet operations move and observe are potentially blocking, the server must be able to serve several blocking operations simultaneously. It is easily seen why simply performing one operation at a time and queuing other

operations would allow false deadlocks to occur. This problem is solved by making the JPS Server multi-threaded, so that once a thread blocks, another thread is activated.

The JPS Server starts a thread for each remote client who needs access to elements on the local kernel. Each thread is dedicated exclusively to one client and runs until the client is shutdown by the user application.

3.4 User Redefinable Memory Semantics

The URMS functionality in JPS is implemented as an URMS interface which in turn may be implemented as a class that is loaded at run time. This approach is far more elegant than the native-code URMS where any URMS function must be available to the PastSet system at compile-time, and since the native version runs at kernel level it does not allow user-defined URMS functions. In JPS a programmer can easily provide his/her own URMS code.

3.5 Distribution

A central part of the distributed PastSet is the distribution of elements. Elements are placed on a server the first time a process enters the dimensions that specify the element. To this end every application that use JPS is connected to a central name-server that keeps track of all existing elements. When an application issues an EnterDimension operation, the JPS Client first contacts the name-server to find out where the element is or should be located, and then it sends the EnterDimension operation to the corresponding server. This way different data-distribution models can easily be implemented by changes in the name-server only. Currently we have implemented three distribution models: Central-Server, Round-Robin Server and First-Touch.

The Central-Server model does not distribute data at all, but centralizes PastSet on one machine. This model does not scale well, but serves as a reference for the other models. The Round-Robin Server tries to balance the load of the elements by letting each node hold every n'th element. This way the load on the servers is equally distributed, assuming the elements are used in a uniform way. In the First-Touch model an element is created on the node that runs the process that first enters the element. Under the assumption that not all elements are accessed uniformly by all processes, the programmer can take advantage of the First-Touch policy by having an element placed at the node where the most activity of the element will be initiated, thereby improving both local and global performance by reducing unnecessary network traffic.

An in-depth coverage of the distribution models and their impact on application performance may be found in [6].

4 Basic Performance

Good performance is vital to all programming paradigms, and cluster programming APIs even more so. To measure the performance of JPS we have chosen to compare to the best known tuple-space system in Java, TSpaces.

All experiments are run on a cluster of 16 nodes, each having two 450MHz Pentium-III CPUs and 256MB main-memory. The nodes in the cluster are connected via a 100Mb/sec Fast Ethernet connected to a switch.

4.1 Latency

In DSM systems it turns out that the latency of communication, i.e. the time it takes from a package is sent until it is received, is of significance to the task grain size that can be supported efficiently by the system. The higher the communication latency is the coarser the grain size must be for good performance. A lower latency allows finer grained parallelism to perform efficiently. For instance, the Internet project of solving the original RSA cryptography challenge communicated via an e-mail system over a global network. In this case latency was extremely high, but the parallelism was trivial, e.g. no communication required during execution. On a system like a SMP machine where the communication latency is lower than access to main memory, i.e. cache to cache, parallelism can be very fine grained.

```
observe(); //Observe the initial token
get_start_time(); //Get timestamp
for(i=0; i<1000; i++){
  move(); //Send token to right neighbor
  observe(); //Get token form left neighbor
}
get_stop_time();
```

Figure 2: Pseudo-code for the ring latency test

To determine the latency of PastSet operations we look at two basic cases, one where two processes passes a token of varying size back and forth, and one where an increasing number of nodes form a ring that passes a token around. The first experiment will be executed in three scenarios. Firstly where the elements used for the communication and the two processes are placed on the same node. Secondly, where one of the processes is on a remote node. Thirdly, where the elements and the processes are all placed at different nodes. Pseudo code for the ring experiment core is shown in figure 2.

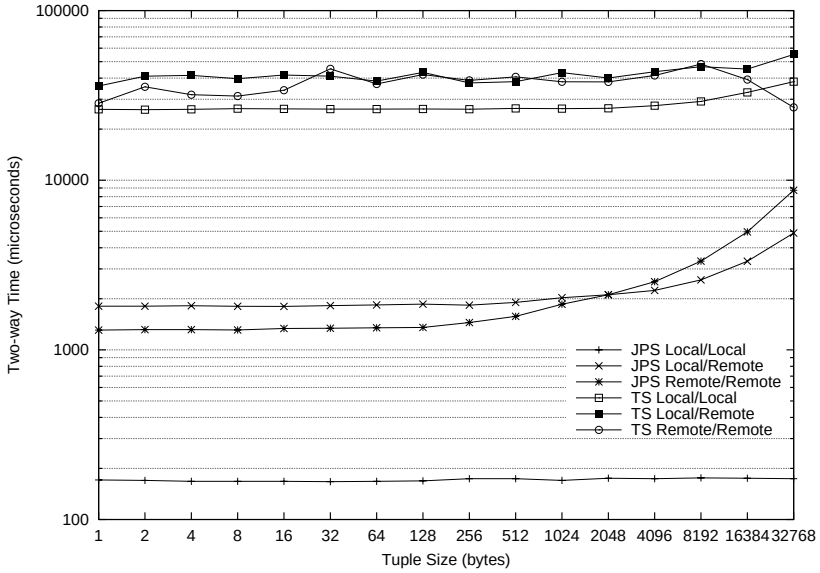

Figure 3: Round trip latency for two process synchronization.

Figure 3 shows the two way latency for process synchronization using JPS and TSpaces. The most noticeable difference between the two models is the inability of TSpaces to efficiently utilize access to a local tuplespace when synchronizing two processes on the same node. In addition TSpaces is more than an order of magnitude slower than JPS, and only

when the data size grows above 4KB is this difference narrowed from the impact of the network propagation delay. At 32K data size JPS is still a factor 5 faster than TSpaces.

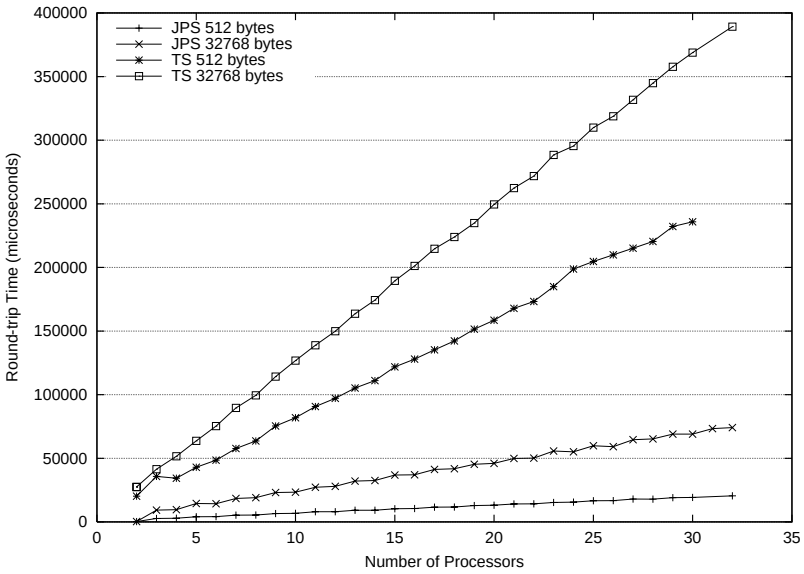

Figure 4: Latency of passing a package in a process ring.

Repeating the experiment for more than two processes, as shown in figure 4, passing the tuple in a ring amongst the processes, the same picture presents itself. Two points are worth while observing in this experiment; first of all that the latency in TSpaces grows far more rapidly on growing number of processes than JPS. And secondly that TSpaces seems to be more sensitive to the size of the tuples that are passed around than JPS is.

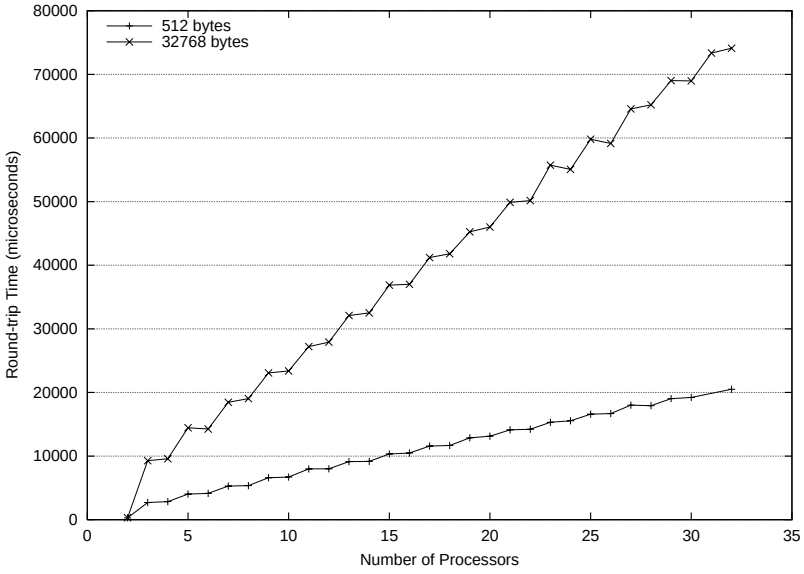

Figure 5: Latency of passing a package in a process ring, JPS only.

In figure 5 we have removed the TSpaces performance to investigate the performance of JPS in further depth. The most important conclusions one may draw from this graph is that the passing of tuples between processes on the same node is very efficient. While the tuples are small the overhead almost hides the effect of the underlying hardware architecture, but once we use 32KB tuples we clearly see the staircase that tells us that we have a cluster of dual-processors.

4.2 Bandwidth

The purpose of testing the communication bandwidth of the PastSet system is to define the limits for writing applications that communicate large amounts of data. Applications that require high bandwidth are frequently found in scientific processing and data-mining. The bandwidth experiments are similar to the latency setup, although somewhat simpler. A process first writes, and then reads varying block sizes. To eliminate noise, and partly to test the behavior when the element grows, each packet size is written and read 1000 times (figure 6).

```
get_start_time();
for(i=0; i<1000; i++)
  move(); //Write data-block to PastSet
get_stop_time();
get_start_time();
for(i=0; i<1000; i++)
  observe(); //Read a data-block from PastSet
get_stop_time();
```

Figure 6: Pseudo code for the bandwidth test.

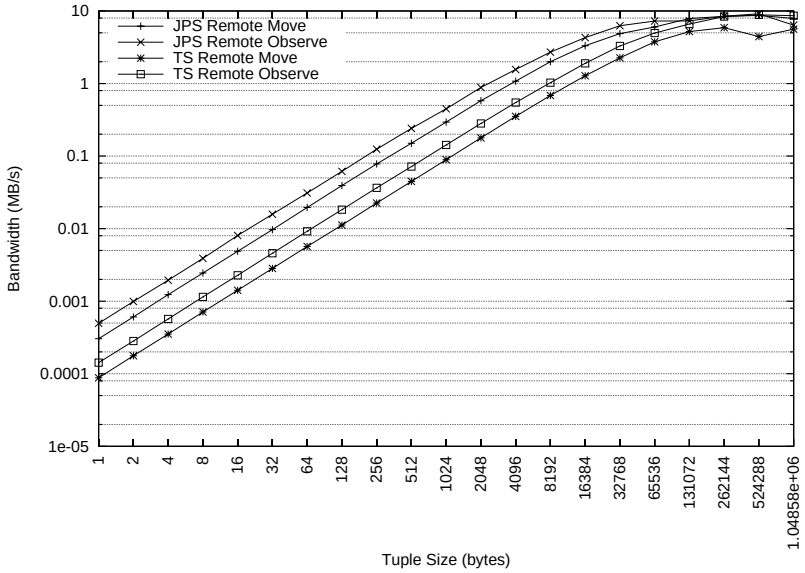

Figure 7: Bandwidth of PastSet local and remote operations.

From the latency-figures (figure 7), we would expect to see a significantly better bandwidth with JPS than with TSpaces and we see just that. At tuple-size 1 byte the bandwidth of JPS is 3.4 times better than TSpaces and at 32KB JPS reads a factor 1.9 faster than TSpaces and writes a factor 2.1 faster.

5 Applications

To test the performance of JPS in real use we have implemented three applications using JPS. In order to compare the performance to the alternatives we have also implemented the three applications in TSpaces and mpiJava[7]. The applications we have chosen are a virtual wind-tunnel, a Lattice Gas Automaton, which use only nearest neighbor communication, an n-body

simulation of stellar bodies, which use all-to-all communication and a Raytracer which uses pipelined communication.

5.1 Lattice Gas Automaton

The Lattice Gas Automaton, LGA, is a complete application that models air-flow in a two-dimensional wind tunnel. The flow is modeled as particles rather than using gas dynamics. The model is both simple and efficient, and is to some degree easy to parallelize. The idea is to model the two-dimensional space using several large bit-matrices. The bit-value one indicates the presence of a gas particle moving in a direction associated with each matrix. The bit-value zero indicates the absence of a particle at this particular location and direction represented by the matrix. The simulation runs for a predetermined number of time steps. The simulated space is divided into bands, and a worker is spawned to service each band.

Each iteration consists of three distinct phases. First all points in the space are checked to see if more than one particle occupies this space, indicating a collision of two or more particles. A set of collision-rules determines the implications of collisions on the particles involved. This first step accounts for most of the time spent in each iteration. Following the collision-test, all gas particles are moved in the matrices. Finally new particles are injected into the system. Each worker starts an iteration by doing collision-detection, it then moves the particles that should be migrated to another partition, moves the particles inside its own block and finally observes and inserts the particles that are migrated into its block. The LGA application uses fairly small messages[1] and may run in phase-lock and thus should have only little contention on the messaging systems. Each worker writes it's border values to one given element. This element is the one most frequently used by the process, and because of this the application is written so that the worker is guarantied to be the first to touch that element which is an advantage when the First Touch model is used.

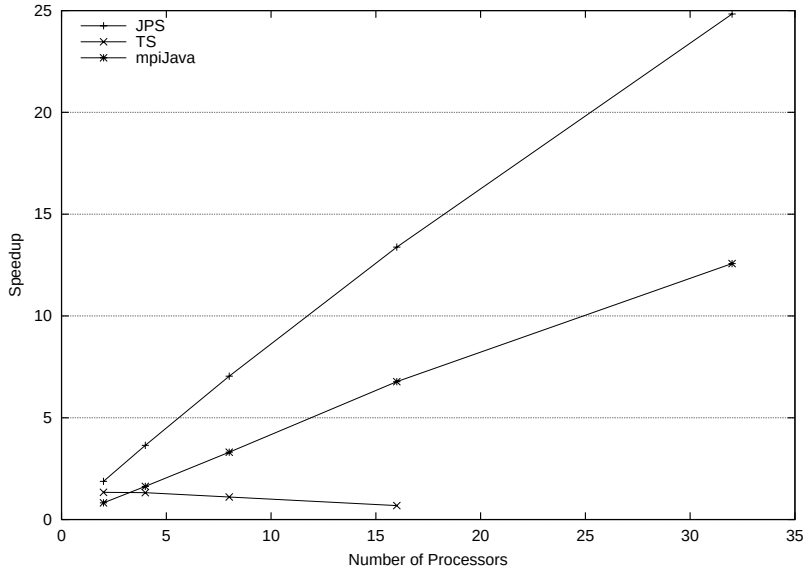

Figure 8: Performance of the LGA application.

The execution times for the LGA application are shown in figure 8. The TSpaces version, which of course differs only in syntax from the JPS version, consistently crashed when running with 32 workers and thus we did not manage to get performance numbers for 32 workers

[1] One bit per point length-wise

using TSpaces[2]. It was the TSpaces server that crashed and not the application. The trend-line for TSpaces is clear enough though and it does not scale at all. JPS however achieves very good performance and gets a speed-up of 25 using 32 CPUs. The mpiJava version runs stable enough but the performance of mpiJava in this application is quite disappointing! Using 32 CPUs the mpiJava version achieves a speedup of only 13, a CPU utilization of less than 0.5 where JPS achieves an utilization of 0.8!

5.2 N-Body

The chosen N-Body algorithm is a trivial $O(n^2)$ complexity simulation of celestial bodies. The JPS implementation works by dividing the responsibility for the bodies amongst the workers. Since updating one body requires knowledge about all the other bodies the JPS implementation exchanges the complete set of bodies in each iteration. To this end we use an URMS function that assembles the bodies from all workers into one single tuple, much like an MPI_Allgather operation. In this way all workers write their local bodies to the PastSet and reads the complete set of bodies.

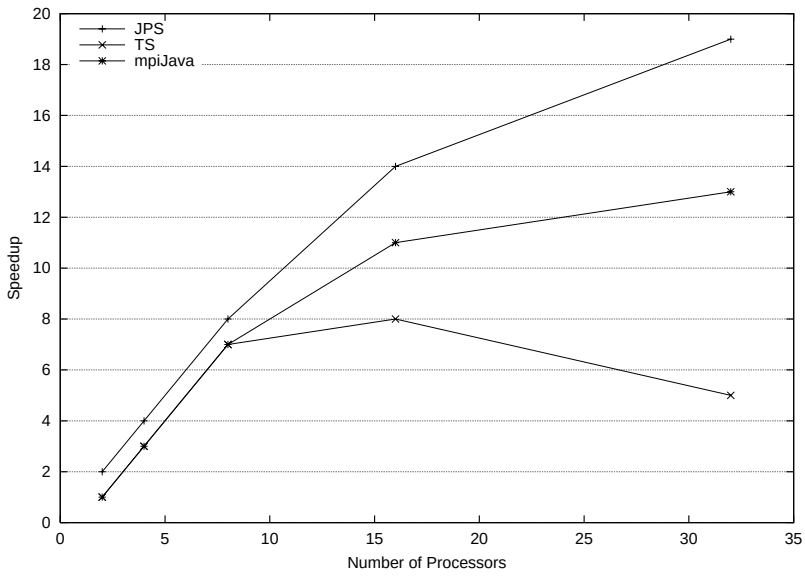

Figure 9: Performance of the N-Body simulation.

Figure 9 shows that JPS achieves close to linear speedup until 16 CPUs while at 32 CPUs the CPU utilization drops to 0.63. It is obvious that collecting results from 32 workers is too much work for the server, and the application looses performance. To address this we need to incorporate the PATHS[8] work in the Java version of PastSet also. In the TSpaces version however each worker need to write its own bodies to the tuple-space and then read *n-1* tuples from the tuple-space, one from each other worker. The mpiJava implementation achieves fair speedup up until 16 CPUs but only gains another speedup of two by adding 16 more CPUs.

5.3 Raytracer

Ray-tracing is another classic target for parallelization which remains popular for two reasons; the need for speedup in ray-tracing is very real; and parallelizing raytracers is seemingly straightforward.

[2]TSpaces also crashed frequently with 16 workers but managed to terminate correctly a few times so that we could get the performance numbers

```
loop:
  Search through all objects to find the first one the ray hits
  If ray hits no objects
    Return ray
  Modify ray to match the hit of the object
  Increase ray-age by one
  If ray-age > ray-max-age
    Return ray
  Goto loop
```

Figure 10: Pseudo-code for the raytracer

The pseudo-algorithm for tracing one point is very simple, as shown in figure 10. This is then done for each pixel in the picture that should be rendered. The intuitive way to parallelize this is to divide the area that is rendered into blocks or stripes and let each worker render one such block. This seems natural since it requires no communication between the workers and thus in effect becomes an embarrassingly parallel application. There are two pitfalls however. First of all the individual blocks won't require equal time to render and thus the parallel execution may become highly unbalanced. This problem could be partially hidden by applying cyclic-striping instead. The second, and more frequently visited, pitfall is that in the embarrassingly parallel version of ray-tracing each worker needs a copy of all the objects in the model to render the portion of the frame that it is assigned to. All workers then need to search exhaustively through all objects for each pixel they render. If the problem is parallelized by distributing the objects along the workers instead one gets the benefit of improved cache locality which can result in super-linear speed-ups for this particular problem, thus we apply the latter approach in this work.

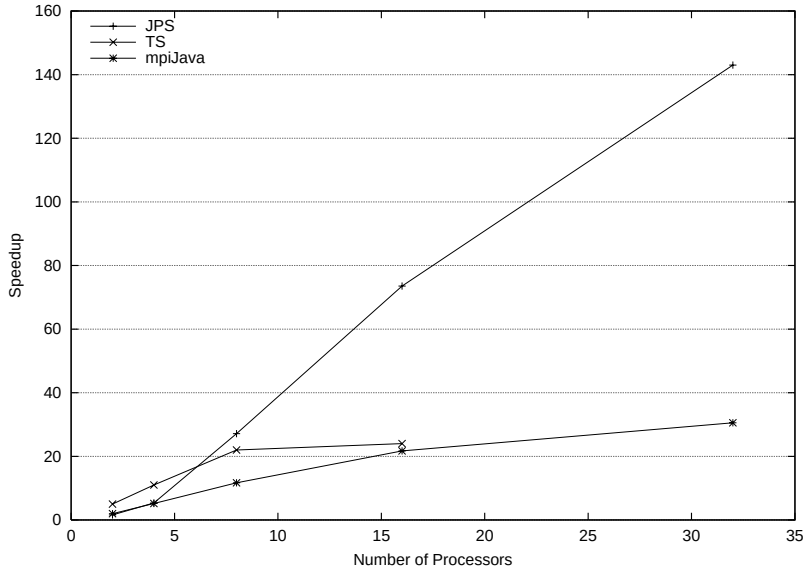

Figure 11: Performance of the raytracer.

The performance of the raytracer is shown in figure 11. Once again we were unable to correctly execute the TSpaces version using 32 CPUs, again it was the TSpaces server that crashed. However, in this scenario TSpaces performs better that JPS using 2 and 4 CPUs. We are not quite sure why this is, but suspect that the TSpaces operations are so slow that the raytracer can successfully run the garbage-collector during an operation in tuple-space, and this may be an advantage in this application. Once the raytracer is run with eight or

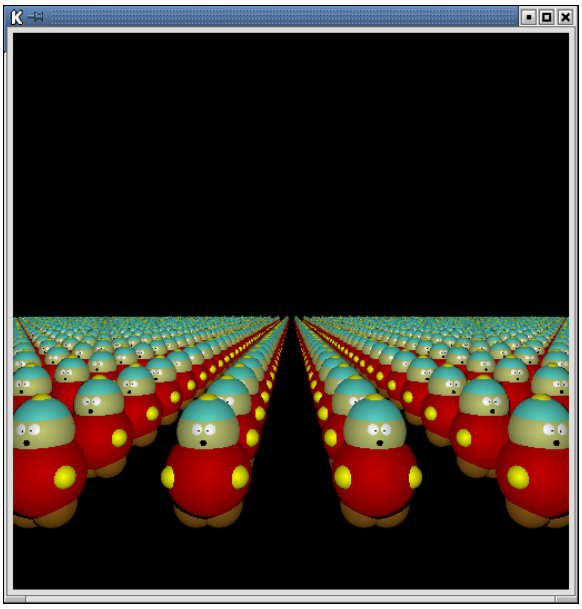

Figure 12: Result of the raytracer.

more CPUs the TSpaces version levels out while JPS takes off and end up with an amazing speedup of 145 using 32 CPUs! The result is very good, but on par with what one might expect with raytracing. The mpiJava implementation turned out to cause us lots of problems as it crashed at various points in the execution. It turned out that mpiJava has problems when an application uses the dynamic memory model heavily, which the raytracer does. The work around we applied was to call the garbage-collector before every send or receive operation. This is clearly visible in the performance of the mpiJava version of the raytracer, which is only super-linear from 4 through 16 CPUs and has a final speedup of 31 using 32 CPUs.

6 Conclusions

Java PastSet, JPS, is an attempt to provide a Java version of PastSet with as few differences as possible. We believe that the design of JPS is sound, and furthermore that the performance we have shown verifies this claim.

Compared to the best existing tuple-based DSM system, TSpaces, JPS is as much as two orders of magnitude faster and it provides a communication bandwidth that is 3.4 times faster than TSpaces.

At application level JPS consistently outperforms both TSpaces and mpiJava. Java PastSet achieves speed-ups of between 19 and 145 which are quite fair considering that we are working with Java.

We believe that JPS provides programmers with all of the convenience of the shared memory paradigm while providing usable performance.

References

[1] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[2] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces(TM) Principles, Patterns and Practice*. SUN MicroSystems, 1999.

[3] IBM. http://www.almaden.ibm.com/cs/tspaces/.

[4] O. J. Anshus and Tore Larsen. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse*, October 1992.

[5] B. Vinter, Anshus, T. Larsen, and J. Bjørndalen. Extending the applicability of software dsm by adding user redefinable memory semantics. In *Parallel Computing 2001, ParCo2001*, September 2001.

[6] B. Vinter, Anshus, and T. Larsen. Data distribution models for a structured distributed shared memory system. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1999.

[7] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpijava: An object-oriented java interface to mpi. In *Presented at the International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999*, 1999.

[8] John Markus Bjørndalen, Otto Anshus, Tore Larsen, and Brian Vinter. Paths - integrating the principles of method-combination and r emote procedure calls for run-time configuration and tuning of high-performance distributed application. *Norsk Informatikk Konferanse*, pages 164–175, November 2001.