Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI)

George K. Thiruvathukal[1], Phil M. Dickens[2], and Shahzad Bhatti[3]
DePaul University, JHPC Research Laboratory, School of CTI

Abstract

Networks of workstations are a dominant force in the distributed computing arena, due primarily to the excellent price/performance ratio of such systems when compared to traditionally massively parallel architectures. It is therefore critical to develop programming languages and environments that can *potentially* harness the raw computational power availab

le on these systems. In this article, we present JavaNOW (Java on Networks of Workstations), a Java based framework for parallel programming on networks of workstations. It creates a virtual parallel machine similar to the MPI (Message Passing Interface) model, and provides distributed associative shared memory similar to Linda memory model but with a flexible set of primitive operations.

JavaNOW provides a simple yet powerful framework for performing computation on networks of workstations. In addition to the Linda memory model, it provides for shared objects, implicit multithreading, implicit synchronization, object dataflow, and collective communications similar to those defined in MPI. JavaNOW is also a component of the Computational Neighborhood [63], a Java-enabled suite of services for desktop computational sharing. The intent of JavaNOW is to present an environment for parallel computing that is both expressive and reliable and ultimately can deliver good to excellent performance. As JavaNOW is a work in progress, this article emphasizes the expressive potential of the JavaNOW environment and presents preliminary performance results only.

Keywords

Desktop supercomputing, sharing, resource management, contention scheduling, relational databases

1 Introduction

Java is rapidly being adopted as one of the preferred languages for writing distributed applications due to its excellent support for programming on distributed platforms. In particular, Java provides automatic garbage collection alleviating programmers from memory management. Also, Java provides compile time and runtime security that can be used as the basis for writing secure applications. Additionally, Java provides excellent support for multithreading, and current Java releases support kernel-level threads allowing the overlapping of computation with communication and I/O. Java can save the state of an object and recreate that object on another machine, supporting both persistent objects and object migration. Recently, a number of distributed frameworks have been developed by Sun, such as Remote Method Invocation (RMI, a client/server remote procedure calling framework), JavaSpaces (a tuple space framework), and

[1] Please visit our main web sites at **http://www.jhpc.cs.depaul.edu** for more information on CN and our various activities and projects.

[2] Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616

[3] PlexObject Technologies, Chicago, IL

Jini (a directory and services framework). Perhaps the greatest benefit of Java is its portability; a Java application can be run on any machine with Java support without recompilation.

Concurrent with the emergence of Java as a preferred language for distributed programming has been the emergence of networks of workstations as a preferred platform for distributed computation. The primary reason workstation clusters have become so important is the excellent price/performance ratio of such systems when compared to traditional massively parallel multi-computers. It is thus natural to explore approaches by which Java can be used to form a virtual parallel machine using ordinary networks of workstations. JavaNOW is one such attempt.

While using networks of workstations as a virtual parallel computer is not a completely new idea, there are many features of JavaNOW which make it unique among message passing systems with similar features. JavaNOW is a pure Java-based system that can execute on any architecture with Java Virtual Machine (JVM) support without recompilation. This is in contrast to other more traditional network-based frameworks, such as MPI [43], PVM [59], Network Linda System [6], and Memo [17] that require architecture-specific binary libraries and executables to be installed on each machine on which the virtual parallel computer will execute. In addition to traditional systems, there are a number of competing pure-Java frameworks (most notably Sun JavaSpaces™) that have also been inspired by the Linda tuple space model; however, these systems have not been designed with high-performance distributed computing in mind and are missing some key features that would enable higher performance, including active tuple evaluation and collective communication facilities. Additionally, most network-based frameworks are based on heavyweight processes, while JavaNOW supports both process-based and thread-based computation. Furthermore, most other network-based systems are strictly message-passing systems, while JavaNOW provides both a message-passing model and a Linda-like distributed shared associative memory for inter-process communication and for mutually exclusive access to distributed shared objects. JavaNOW extends the Linda model (in a significant way) by providing a rich set of collective communication and computation primitives similar to those found in MPI. Finally, JavaNOW augments both Linda and MPI by supporting a data flow model of computation.

In this paper, we provide a detailed discussion of the design and implementation of the JavaNOW computational framework. We begin by comparing JavaNOW to other network-based and Linda-like frameworks. We then discuss the distributed shared associative memory at the core of JavaNOW, and show how it can be used to support MPI-like collective communication and computation, a data flow model of computation, and synchronization primitives such as locks, barriers, and semaphores. Finally, we provide preliminary performance results and conclude with a discussion of the significance of the JavaNOW system.

2 Overview of JavaNOW and its Relation to Linda, PVM and MPI

In many respects, JavaNOW can be viewed as a hybrid system: some PVM (Parallel Virtual Machine [59]), some Linda [13], some MPI (Message Passing Interface [44][46]), and some experimental components. From a usage standpoint, JavaNOW provides a simple facility for starting tasks that is reminiscent of the PVM software. From a programming standpoint, JavaNOW has much in common with Linda and MPI, having a small number of primitives to support producer/consumer style communication (as found in Linda) and collective operations that can be performed on shared objects (as found in MPI).

JavaNOW supports the notion of virtual processors, a technique first used in data-parallel computing languages, such as C*, PVM and MPI. This approach is still used in other projects, such as Globus [24] and Legion [34], and thus has proven value in the community to this day. (A virtual processor is emulated as a process--among a group of processes--that are spawned prior to actually doing communication using the message-passing primitives of JavaNow.) Thus, at the core, JavaNOW supports a SPMD model of computation. It should be noted however that the decision to use Java in fact enables the possibility of an MPMD style of programming since a Java program can dynamically load classes at anytime. This possibility will not be discussed further in this paper, due to space considerations and being slightly beyond the intended scope. It should also be noted that the daemon processes in JavaNOW spawn light-weight threads rather than heavy-weight processes thus providing parallelism in a much more efficient manner.

2.1 Distributed Logically-Shared and Associative Memory

Application processes coordinate and communicate through distributed associative shared memory *similar* to the global tuple space model found in Linda. We emphasize that it is only a similarity since there are significant differences as discussed below. In JavaNOW, the shared objects are referred to as *Entities* and the repository where these objects are stored is termed the *EntitySpace*. Each Entity in the JavaNOW system consists of two components: the name of the Entity and its value. It is important to note that name and value both denote Java objects, which unlike Linda objects, carry both state and behavior information.

We chose to incorporate into JavaNOW a Linda-like memory model for three reasons: First, the Linda memory model has been available for a number of years and has been proven to provide powerful semantics for writing parallel applications. Secondly, it is widely accepted that programming in a shared memory system is easier than programming in a message passing system. Thirdly, a Linda-like shared memory model can be implemented much more efficiently and provide better performance than traditional distributed shared memory architectures. This is because traditional distributed shared memory systems implement sharing at the level of a page which can lead to false sharing and prohibitive communication costs [52].

However, the JavaNOW memory model is distinct from the tuple space model found in Linda. One difference is that JavaNOW allows multiple EntitySpaces rather than the single tuple space supported by Linda. This allows localized name spaces for shared data and the creation of private channels for inter-process communication. Also, the distribution of the EntitySpace in JavaNOW is quite distinct from the implementation of the tuple space in Linda. Conceptually, JavaNOW employs a distributed hash table abstraction to implement a transparent set of entity spaces. It is important to note that a key difference between the JavaNOW abstraction and a distributed hash table does exist—the entries of the hash table are *unordered queues* of objects and a single instance as found in the Java Hashtable class. JavaNOW also provides excellent support for *placement control*, which allows for precise placement of distributed data structures. This placement control can be achieved to a large extent because the Java language Object class provides a default definition of a hashcode() method, which guarantees that any intrinsic or user-defined class that is used to create objects will produce a hash value, albeit not a well-conditioned hash value. Precise placement control can be very easily achieved in JavaNOW due to a design decision that was made to clearly isolate the key from the rest of the tuple, which was first done in previous work published by one of the authors on the Memo system [17][18]. The way precise placement control is achieved (discussed in detail later) is to provide a custom hashcode() function that can be used to compute (mod the number of contexts) the precise destination for an entity. Linda implementations generally replicate the tuple space among the available processors thus incurring the high costs of coherence protocols and unpredictable latency for even the most basic communication primitives.

We stress that JavaNOW is not "yet another tuple space" implementation or "yet another language." It occupies a unique space in its ability to support a number of other message passing "personalities." Programmer familiar with MPI, PVM, and Actors-derived systems (such as Charm++) will find a number of familiar features that make the system eminently usable. We also stress that the choice of tuple spaces in part is validated by Sun's decision to incorporate similar principles in their JavaSpaces and Jini designs; however, these offerings from Sun do not appear to be a sound basis for high-performance computing as many functions clearly missing from the original Linda specification (active tuples) and MPI (collective operations). Emulating these missing operations is somewhat difficult without primitives that support overlapped computation and communication, which is directly supported by active entities, which we discuss in detail in Section 4.2.

2.2 Mutual Exclusion, Blocking and Non-Blocking Primitives

Similar to Linda, MPI, and PVM, the JavaNOW environment provides support for blocking and non-blocking primitives. In particular, JavaNOW defines two operations, put and get, that allow a producer/consumer relationship to be established between two tasks. There is both a blocking and non-blocking form of the get operation. In the blocking form, the get operation is guaranteed to block forever if a corresponding put method is never posted. In the non-blocking form, the get operation will return immediately if no corresponding put operation has been performed.

All **put** and **get** operations that manipulate the same Entity are guaranteed to be mutually exclusive freeing the programmer from concerns related to the synchronization of the EntitySpace. While a detailed description of the implicit synchronization mechanism is beyond the scope of this paper, we note that it is based on an abstraction called a shared directory of unordered queues (a shared data structure that guarantees mutual exclusion and has entries with mutually-exclusive access), which was discussed in two of our earlier papers on Memo [17] and Enhanced Actors [62].

2.3 Collective Communication and Computation Operators

JavaNOW supports collective communication and computation operators similar to those defined in MPI. Such operations are arguably among the most important features of the MPI framework and are supported in MPI by the following two features:

1. collective operators – communication + an operation to be performed.
2. communicators – an abstraction that allows a group of processes to participate in a collective operation.

A predefined communicator, MPI_COMM_WORLD, exists to allow all processes (the normal case in MPI programming) to participate in a collective operation. A number of primitives exist to allow new communicators to be defined to include/exclude processes in/from a communicator.

JavaNOW takes a different approach and supports collective communication using what are termed *active* Entities. An active entity (class ActiveEntity in the JavaNOW library), which is very similar to the notion of a *future* in data flow languages, is an object that performs a computation (a detached task) and then converts itself into a *passive* Entity (or result). We emphasize the word similar; as defined here, *futures* should not be confused as being semantically equivalent to lazy evaluation and closures. (We do have plans to support this in a future implementation of JavaNOW. Christopher and Thiruvathukal—one of the authors—are discussing futures in a forthcoming book on High Performance Java Computing.) The active entity is similar in principle to a future in the respect that a reader can block on the future until the value has been written; however, the difference is that the value can be written multiple times, which does not follow the established definition of futures described in the functional programming literature. Using active entities, collective operations can be easily defined. For example, the reduction operator can be implemented as a task that awaits a certain number of Entities. As the Entities are computed, the ActiveEntity can consume them one at a time to produce a *curried* result. When the result is finally computed, the ActiveEntity becomes a passive Entity (whose value is the final result) and is inserted into the EntitySpace. It should be noted that this example is based on the common use of reduce, which supports commutative operations such as addition. A slight modification is required for non-commutative operations which can also be implemented asynchronously and efficiently. JavaNOW also supports collective communication operations such as reduce, scatter, gather, and barrier, all of which are implemented using the same notion of active Entities.

2.4 Abstract Provider Architecture

Similar to the MPI implementation developed at Argonne National Laboratory, JavaNOW uses abstract factories allowing JavaNOW to be layered on top of any communication mechanism. In JavaNOW however, the design allows the decision of the underlying communication scheme to be deferred until run time. Our early experiences with this mechanism have included TCP/IP sockets, Java Remote Method Invocation (RMI), and the Common Object Request Broker Architecture (CORBA).

2.5 Data Flow

JavaNOW also supports features that are not available in either Linda, MPI or PVM. One such example is a coarse-grained dataflow model of computation. In this model, operations are executed when their data becomes available rather than when dictated by control flow statements. This support for data flow also comes from the concept of the ActiveEntity which, as noted above, is very similar to the notion of a future. This similarity is important since futures have been shown to be useful in many programming paradigms (even outside of high-performance computing) such as suspended evaluation, lazy evaluation, task graphs, and other commonly used techniques for high-performance parallel and distributed computing.

3 Other Related Work

Several other projects are using networks of workstations for building parallel applications. Most of the network-based parallel processing systems are built on top of a message passing layer such as PVM [59] or MPI [43]. Such systems include ORCA [3], Piranha [30] and Legion [34]. Other systems are based on a global address space or distributed-shared memory and include Ivy [42], Munin [12] and TreadMarks [40]. These systems allow networked workstations to be treated as a multiprocessor system with the underlying software providing coherent memory. However, such systems suffer from page shuttling, false sharing, the need for distributed locking, and the lack of fault tolerance [20].

Other parallel frameworks based on Linda and the shared tuple space include C-Linda [48], Glenda [55], and JavaSpaces [61]. C-Linda is a C based implementation of Linda. Glenda is a Linda implementation on top of PVM. Memo is a C library that implements Linda like data structures for storing associative-shared memory. None of these systems (except JavaSpaces as discussed below) take advantage of the power and flexibility of the Java language.

JavaSpaces is a Linda-inspired framework written in Java. JavaSpaces uses a server object to manage a *space* (which is similar to a tuple space or EntitySpace) and, similar to the JavaNOW system, allows the creation of multiple spaces; however, JavaSpaces does not offer the concept of an active entity and does not easily support alternative transport protocols to Remove Method Invocation (RMI). Additionally, JavaSpaces supports even fewer communication primitives than Linda, the most notable being primitives for non-blocking communication. JavaSpaces also has the limitation that actual objects are not stored in the tuple space. Instead, user-defined classes must make use of public instance variables that are then cloned and entered into the space. This can be argued to be an implementation detail; however, it is a rather sloppy design decision that, ironically, is not nearly as clean, elegant, and simple as the original Linda C implementations. Requiring classes to expose member variables violates many of the principles of the object paradigm and cannot be considered much different that programming with global variables in, say, FORTRAN or C. Finally, JavaSpaces does not support many of the most useful features of JavaNOW, most notably placement control and collective communication, which are of proven value in high-performance computing applications written using the Message Passing Interface (MPI) library.

There are a number of other Java-based frameworks for parallel computing. Most of these frameworks can be separated into five different categories. The first category consists of frameworks that use Java as a graphical based coordination system to submit parallel applications to specialized hardware. These systems are generally built on top of either PVM or MPI and include JavaDC [15], and SARA [1]. The second category uses Java as wrapper for existing frameworks. These systems include Java/DSM [65], JavaPVM [64], and the Java wrapper for MPI [47]. The third category consists of Java based languages and frameworks that extend the Java language with new keywords. Some of these systems use a preprocessor to create Java code, others use their own compiler to create Java byte code, and still others create executable programs that lose the portability of Java. These Java based languages include the E language [22], JavaParty [51], and Titanium [54]. The fourth category of Java-based frameworks consists of systems that are Web oriented and use Java applets to execute parallel tasks. As Java applets execute under strict security requirements, most of these systems use a broker for inter-process communication. Such Web-based frameworks are mostly targeted for large-grained parallel applications since network latency between machines connected over a Wide Area Network (WAN) is significantly higher than the latency for machines located on a local area network. These systems include ParaWeb [9], Bayanihan [53], IceT [33], Javelin [16] and Javelin++ [49] and KnittingFactory [5].

The authors believe web-based computing is an interesting direction that presents interesting problems; however, for web-based computing to become viable, the quality of web browser implementations will need to increase significantly. Most web browsers (including the best implementations, which all run on the Windows™ operating system) are crash suddenly when running Java and embedded scripts, except for the most trivial computations. Unix implementations of Netscape crash frequently when doing something as mundane as reading e-mail, let alone when running Java. And virtually all implementations of web browsers do not support the latest versions of the Java Development Kit, and this is not likely to change due to business politics. JavaNOW is not currently a web-based approach; however, there is nothing in its fundamental design that would preclude its use to support web-based computing (in particular the server side). We have no immediate plans to support web-based computing using JavaNOW until the issues described in this paragraph are overcome.

The fifth category, and the one in which JavaNOW would be placed, consists of Java-based frameworks that use pure Java libraries to support parallel and distributed applications. This category also includes JavaSpaces [61], JPVM [23], Ninflet [45], and Java//. [10].

4 Fundamental Abstractions

In the previous section we introduced some of the key ideas behind the JavaNOW system. In this section, we present each of these abstractions in greater detail with some clarifying examples.

4.1 Entity

Probably the most fundamental concept in JavaNOW is the Entity. An Entity is the basic unit of storage in the JavaNOW system and is stored in the associative and logically-shared repository termed the EntitySpace. Entities consist of a *key* and a *value* as a pair, where both the key and value can be an instance of any (serializable) Java object. A user creates an Entity instance as follows:

```
Integer i = new Integer(10);
String s = new String("Data component");
Entity t = new Entity(i, s);
```

In above code, an Entity is defined with the key equal to the integer value "10" and a value equal to the string "Data component". When there is a need to subsequently retrieve this Entity, the key field (the Integer "10") is all that would be needed to match and retrieve it. It should be noted that the above code just creates the Entity and does not add it to the EntitySpace.

We provide a few words about the semantics of creating a key, such as **new Integer(10)** in the fragment of code shown above. Java's Object class provides a function called hashcode(), which is usually defined on a per class basis and is intended for placing instances in a hashed collection, such as a Hashtable. We use the hash code to determine a *destination* for the Entity having a particular key. On average (from previous work on Memo) the distribution achieved by using a hashing scheme tends to be nearly uniform in practice. The approach of having a particular destination is transparent to users; however, any user-defined Entity can override the hashcode() function to customize precisely how the key is mapped to a destination virtual processor.

4.2 Active Entities

An ActiveEntity is derived from an Entity and defines an abstract method execute() that is overridden with user-defined computation. An ActiveEntity is executed as a consequence of executing the eval() operation discussed below. An instance of an ActiveEntity is passed to the eval function, indicating that the ActiveEntity is to be executed. Once its execution is completed, the ActiveEntity instance becomes a (passive) Entity, and the result of the user-defined compuation is stored in the EntitySpace in association with the specified key.

The following code shows the use of subclasses to create a new kind of ActiveEntity called Task.

```
public class Task extends ActiveEntity {
  public Object execute(Object arg, JavaNOWAPI api) {
  Object o;
  int myid = ((Integer)arg).intValue();
  ...
  return o;
  }
}
```

This code fragment shows how to evaluate an ActiveEntity:

```
ActiveEntity task = new Task(new Integer(10));
GetJavaNOWAPI().eval(new EntitySpace("ESKEY"), task, new Integer(1));
```

The semantics of active entities are a bit involved. The ActiveEntity differs from an Entity in only one major respect: it has an execute() method, which represents the “active” aspect of the Entity. We faced two design choices:

- After the execute() method has completed execution, just leave the entire object behind as the result. (i.e. the ActiveEntity instance itself)
- Alternatively, return the result (an Object) and use it to create a new Entity.

Ultimately, we opted for the latter option. Our experience with Actors-derived systems and the concept of a replacement operation led us to the conclusion that an in-place operation is a largely unfamiliar programming technique, especially in scientific programming, where procedural abstraction remains the norm, and programmers are more comfortable with the notion of an invocation having a return value. It is entirely possible to support both designs; however, we are hoping to gain more experience with actual applications before adding more features than necessary.

4.3 Entity Space

An EntitySpace stores shared Entities and is accessed by a unique key which can be any (Serializable) Java object. (We enforce the Serializable restriction, because our current implementation relies on Java Object Serialization and RMI for communication.) As noted above, there is no limit to the number of EntitySpaces that can be defined. JavaNOW distributes the contents of an EntitySpace among the hosts participating in computation using a simple hashing scheme. Note that the EntitySpace itself does not provide the actual storage for the Entities but rather links them onto lists called *folders*. When an entity is inserted or retrieved, the EntitySpace hashes its key into a number that determines the folder into which the Entity will be linked. Whenever an Entity is inserted, retrieved or removed from a folder it is locked to support the mutual exclusion principle discussed earlier.

4.4 JavaNOWApplication

A JavaNOW application must be derived from the JavaNOWApplication class, which stores a JavaNOWAPI handle as a member. This handle is accessed by the user application. The JavaNOWApplication class defines the two following abstract methods that are overridden by the user:

```
void master()
void slave(int id)
```

Given N processes, the process with ID 0 invokes the **master()** method, and the other (N-1) processes invoke the **slave()** method. Note that additional heavyweight processes cannot be dispatched in the present design. However, additional tasks can be created at any time by creating an instance of an ActiveEntity which, when passed to the **eval** operator, will create a new thread to execute the user-defined computation.

The following code demonstrates the use of the JavaNOWApplication class:

```
public class AnApp extends JavaNOWApplication implements java.io.Serializable {
public static void main(String args[]) {
  AnApp app = new AnApp(args[0]);
  }

public Hello(String propertyFile) {
  super(PropertyFile);
  // local initialization
```

```
applicationIsReady();
}
public void master() {
  // . .
}
public void slave(int myid) {
  //. . .
}
```

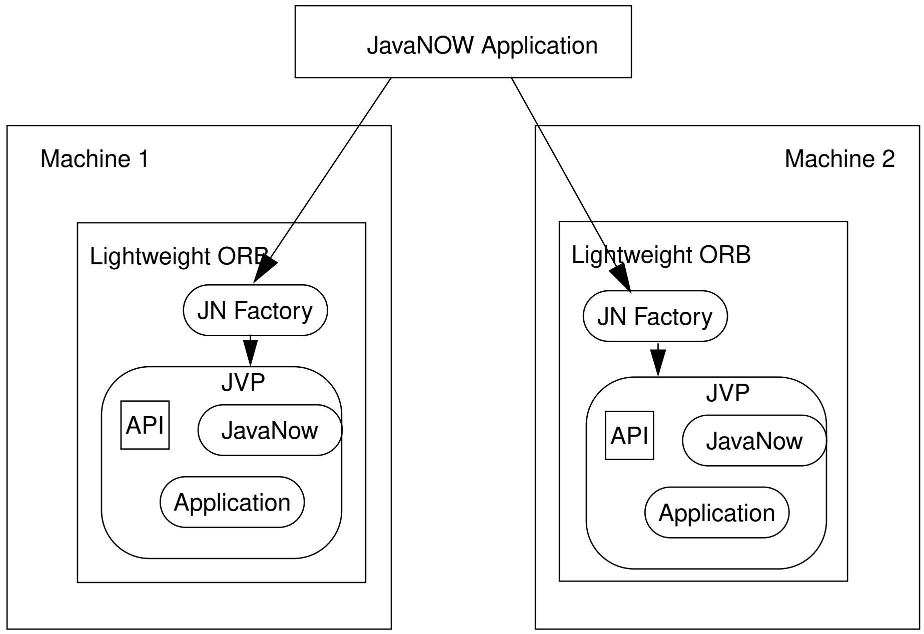
The above code can be construed as a skeleton of the minimum code one would need to create a JavaNOW application. After the application completes its initialization, the **applicationIsReady()** method is invoked and the JavaNOW Spawner (discussed below) is used to create a virtual processor on each host specified by the user.

5 JavaNOW Architecture

JavaNOW is a component-based architecture, wherein each component is designed using a set of Java interfaces. This design was chosen to facilitate different implementation options for the same component. One example of this capability was noted above: the Abstract Provider Architecture can be implemented using a variety of communication mechanisms. Throughout the design of JavaNOW, similar decisions have made to facilitate design changes and to support performance tuning.

The primary components of JavaNOW include a lightweight Object Request Broker (ORB), a virtual processor (VP) factory, the spawner, virtual processes, the kernel, and the application and user interfaces. The overall structure of JavaNOW is shown in Figure 1 and we discuss each of these components in the following sections.

Figure 1: Java Now Architecture



JN = JavaNOW, JVP = Java Virtual Processor, API = Application Programming Interface

5.1 Lightweight ORB Component

The Remote Method Invocation (RMI) framework provides a simple and elegant solution for creating and accessing remote objects and is one approach supported in JavaNOW. In this implementation, JavaNOW uses the RMI programming interfaces to register and discover remote servers. In the socket-based implementation, lower level sockets are used to build a lightweight ORB that provides interfaces to register and lookup remote objects (similar to the designs used in RMI and CORBA). As noted above, JavaNOW can be extended to support other transport mechanisms as well.

5.2 Factory Component

The Factory Component is started on each machine that will participate in the computation and defines an interface to register and start a virtual process (VP) on the local machines. In the RMI implementation of JavaNOW, when the factory component is started it registers itself in the RMI registry. In the socket-based implementation, the registration takes place in a customized registry.

5.3 Spawner Component

When a user submits an application to JavaNOW, the Spawner Component gathers information about the user application including the number of application processes and the list of machines on which the application processes will be executed. The application is then assigned a unique identifier. Next, the Spawner looks for the factory on each participating machine and sends it a request to start the VPs that will execute on that machine. The number of VPs spawned by the JavaNOW Spawner is equal to the number of processes specified by the user (which may involve multiplexing more than one VP on a given processor). Each VP is then assigned a logical host number (much the same as done in MPI and other systems) that is used to uniquely identify the VP within the application. The application id and VP id can be used to establish a unique context identifier (something that is useful when running multiple applications that may need to coordinate beyond the "boundary" of a running application). Finally, a Kernel process is spawned on each processor involved in the computation.

5.4 Kernel

The Kernel Component defines an interface to support the core primitive operations of JavaNOW. (It is at this low-level that JavaNOW appears to be very similar to Linda.) The following is the list of core operations supported in the Kernel Component:

1. **put—**Inserts an Entity into an EntitySpace. Recall that an Entity consists of two contained objects, **key** and **value**. Multiple values can be stored in association with a common key. (Recall from the earlier discussion that the entity space is an implementation of the shared directory of unordered queues abstraction, which is a 1 to N associative map that has a distributed implementation.)
2. **eval—**The **eval** operation starts a thread to perform a user-defined operation asynchronously. The operation is defined by extending the ActiveEntity class and overriding the **execute()** method. The ActiveEntity instance is run (usually, as a separate thread) and leaves the result of the operation after its execution is complete.
3. **get/getIfExists—**Removes an Entity from a given EntitySpace. If the Entity is not present in the EntitySpace, the **get** operation blocks until another thread **put**s the Entity in the EntitySpace. The **getIfExists** function tries to remove an Entity from the EntitySpace without blocking. If the Entity does not exist a value of **null** is returned.
4. **read/readIfExists—**The read and readIfExists operations are similar to get/getIfExists except that instead of removing the Entity from the EntitySpace a *copy* of the Entity is returned.
5. **size—**The size operation returns number of Entities in the EntitySpace.
6. **clear—**The clear operation removes all Entities from the EntitySpace.

5.5 User Interface (UI)

The User Interface component defines a set of operations to manage shared data and create new computation tasks. It provides these services by utilizing the Kernel Component. The User Interface is discussed in detail below.

5.6 User Application

The User Application is not exactly a component but acts as an abstract base class for deriving one's own JavaNOW applications. As noted above, it defines two abstract methods that are implemented by the user: master() and slave().

6 JavaNOW User Interface

The JavaNOW User Interface defines a set of operations that applications use to manipulate the EntitySpace(s) and to start new threads to perform user-defined computation. The User Interface in turn calls on the services of the JavaNOW Kernel to perform these primitive operations. Given the importance of the User Interface it is worth while to elaborate on the many services it provides.

6.1 Defining EntitySpaces and Entities

The user can declare multiple EntitySpaces each of which is identified by a unique key, where the key can be any serializable Java object. As an example consider the following program fragment:

```
EntitySpace ts = new EntitySpace(new "JOBJAR");
```

In this example, JavaNOW creates an EntitySpace ts with the key “JOBJAR”. An Entity is created in a similar manner:

```
Entity e = new Entity("KEY", "VALUE");
```

In this example an Entity with key “KEY” and value “VALUE” is created.

6.2 Inserting an Entity in the EntitySpace

The JavaNOW User Interface defines a **put** operation to insert an Entity into an EntitySpace. The prototype of the **put** operation is:

```
void put(EntitySpace es, Entity e)
```

This operation could be used as follows:

```
EntitySpace es = new EntitySpace("SHARED DATA");
Entity e = new Entity("KEY", new Integer(100));
getJavaNOWAPI().put(es, e);
```

The getJavaNOWAPI() method is defined in the JavaNOWApplication class and returns a handle to the JavaNOW User Interface. The machine on which the inserted Entity will reside is determined by a simple hashing function as previously discussed.

6.2 Retrieving an Entity from the EntitySpace

A user can either retrieve and remove an Entity from an EntitySpace or simply retrieve a copy of the Entity. (Retrieving a copy depends on whether the operation is determined to be local or remote--transparent to the user. In the case of local a deep-clone is performed; otherwise, Object Serialization is used to serialize and deserialize the object from its remote location.)

The following form of the get function actually removes the Entity:

```
Entity get(EntitySpace es, Entity e)
```

The **get** function can be used as follows:

```
EntitySpace es = new EntitySpace("SHARED DATA");
Entity ek = new Entity("KEY");
Entity e = getJavaNOWAPI().get(es, ek);
Integer I = (Integer) e.getEntityValue();
```

The above code defines an EntitySpace with the key “SHARED DATA” and attempts to remove an Entity with the key “KEY” from that EntitySpace. The **get** operation is a blocking operation, so if the Entity does not exist in the EntitySpace the operation will block until the Entity becomes available. If, on the other hand, there are multiple Entities matching the Entity key, the **get** operation will return the Entity in FIFO order.

The JavaNOW User Interface also provides a non-blocking version of the **get** operation that returns the Entity if it exist in the EntitySpace and otherwise returns null. Here is the prototype for the non-blocking **get** operation:

```
Entity getIfExists(EntitySpace es, Entity e)
```

The following code illustrates the use of the **getIfExists** operation:

```
EntitySpace es = new EntitySpace("SHARED DATA");
Entity ek = new Entity("KEY");
Entity e = getJavaNOWAPI().getIfExists(es, ek);
if (e != null) {
  Integer I = (Integer) e.getEntityValue();
}
```

In this example, the **getIfExists** function tries to find an Entity with the key "KEY" in an EntitySpace with the key "SHARED DATA" and returns null if there is no matching Entity.

Another useful operation defined in JavaNOW is the **read** operation which returns a copy of the requested Entity but does not remove it from the EntitySpace. The prototype for this operation is:

```
Entity read(EntitySpace es, Entity e)
```

The following is an example of how the **read** operation can be used:

```
EntitySpace es = new EntitySpace("SHARED DATA");
Entity ek = new Entity("KEY");
Entity e = getJavaNOWAPI().read(es, ek);
Integer I = (Integer) e.getEntityValue();
```

The **read** operation is a blocking operation, so if the sought after Entity does not exist in the EntitySpace, the caller will block until the Entity becomes available.

The JavaNOW User Interface also defines a non-blocking version of the **read** operation:

```
Entity readIfExists(EntitySpace es, Entity e)
```

The **readIfExists** operation is used in a manner analogous to the **read** operation except that it does not block is the Entity does not exist.

6.3 Using an ActiveEntity to Create New Threads of Execution

JavaNOW provides the **eval** operation to spawn a thread to execute user-defined computation:

```
void eval(EntitySpace es, ActiveEnttiy e, Object arg)
```

The user defines the computation by providing the **execute** method within the ActiveEntity class. The **eval** operation is then passed an instance of ActiveEntity which contains the user-defined method. The following example illustrates how a new thread is created and executed:

```
public class ATask extends ActiveEnttiy {
   public Object execute(Object arg, JavaNOWAPI JavaNOW) {
    //...
    return someObject;
  }
}

EntitySpace es = new EntitySpace("SHARED DATA");
```

```
ATask task = new ATask("KEY");
GetJavaNOWAPI().eval (es, task,"USER ARGUMENT");
Integer I = (Integer) getJavaNOWAPI().get(es, ek).getEntityValue();
```

The above code creates an instance of the ActiveEntity class and passes that instance to the **eval** operation. The **eval** operation creates a thread to process the user-defined **execute** method and then stores the result as the value of that Entity. The user then retrieves the result by issuing a blocking **get** operation.

6.4 Data flow

The JavaNOW User Interface provides operations that simulate a data flow model of computation. In a data flow computation, an operation is executed as soon as its data becomes available. The prototypes for the two data flow operations provided in JavaNOW are shown below.

```
void  putDelayed(EntitySpace es1, EntitySpace es2, Entity e)
void  evalDelayed(EntitySpace es1, EntitySpace es2, Entity e, ActiveEntity task)
```

The **putDealyed** method performs a blocking **get** operation to retrieve some Entity e from some EntitySpace es1, and after retrieving Entity e stores it in EntitySpace es2.

The **evalDelayed** method waits for some Entity e in some EntitySpace es2, to become available. When it arrives, the operation removes e from es2. Next, the **eval** operation is called with an AactiveEntity object (denoted by task in this example), using EntitySpace es1 and passing the value of e as the argument to the **eval** operation.

6.5 Collective Communication and Computation

The JavaNOW User Interface also defines a set of collective communications and computations across EntitySpaces. In this section, we briefly describe these operations.

The broadcast operation allows an instance of an Entity to be deposited in multiple EntitySpaces. The scatter operation takes as parameters an array of EntitySpaces and an array of Entities. It removes the nth Entity from the Entity array and inserts it in the nth element of the EntitySpaces array. The gather operation provides the ability to retrieve multiple Entities from a set of EntitySpaces. It takes as parameters a destination EntitySpace and an array of source EntitySpaces. The operation takes the nth Entity from the nth EntitySpace (i.e. Entity one from EntitySpace one, Entity two from EntitySpace two and so forth) and places that Entity into the destination EntitySpace. The concat operations takes an array of Entities and copies that array into multiple EntitySpaces. Finally, the User Interface provides an index operation which is similar to the transpose operation on a matrix.

JavaNow also provides a reduce operator to facilitate collective computation. This operation is effectively equivalent to performing an eval operation on a set of Entities. The combine operation is similar to the reduce operation except that it stores its result in multiple EntitySpaces. The prefix operation is also similar to the reduce operation except that it stores a partial result in multiple EntitySpaces.

6.6 Other Operations

In addition to the operations discussed above, the JavaNOW User Interface provides a **barrier** routine to block a process until some specified number of processes have similarly executed the **barrier,** a routine to determine the number of active processes and a routine to halt the execution of all processes.

7 Distributed Idioms and Patterns in JavaNOW

Over the course of the years of research in parallel and distributed systems a number of idioms and patterns have been developed to promote inter-process communication and the safe manipulation of shared data structures. Our goal is not to re-invent these useful techniques but rather to incorporate them into the JavaNOW framework. In this section, we demonstrate how some of the classical IPC mechanisms can be incorporated into JavaNOW.

7.1 Inter-process Communication

JavaNOW differs from MPI and PVM in that it does not provide direct point-to-point inter-process communication (IPC) primitives but rather provides a producer/consumer model of IPC. This is much the same model that is supported in the Communicating Sequential Processes (CSP) model defined by C. A. R. Hoare (the seminal research on the topic of IPC).

To support IPC, an Entity is created with a key agreed upon by both the sender and receiver (this is similar to the notion of a named channel or a named pipe found in operating systems). The values to be communicated are placed in the value field of the Entity. The sender can issue as many send calls as desired, creating an Entity instance for each object to be transmitted to the receiver. The receiver may post as many **get** operations as desired, allowing for the possibility of multiple communications.

The following code illustrates this form of IPC in JavaNOW.

```
// Sending Process
. . .
EntitySpace esk = new EntitySpace(“ESKEY”);
Integer n = new Integer(100);
Entity e = new Entity(“key”, n);
getJavaNOWAPI().put(esk, e);

// Receiving Process
. . .
EntitySpace esk = new EntitySpace(“ESKEY”);
Entity e = new Entity(“key”);
String msg = (String) getJavaNOWAPI().get(esk, e);
```

7.2 Locks, Mutexes, and Binary Semaphores

Although Java provides a monitor-like abstraction (as part of the language proper) that can be used to support the synchronization of threads, the semantics of this abstraction are local rather than distributed. Thus, data that is shared between two threads (or processes) executing on different processors cannot be easily protected.

JavaNOW guarantees atomic operations at the level of an Entity. Thus a given key can be used to support a basic lock discipline. For example, an initialization process (the master) can deposit an Entity into an EntitySpace, which reflects a lock that can be used anywhere in the application.

The process that needs to perform a lock operation will simply issue a blocking **get** operation to acquire the Entity (representing the lock) from the EntitySpace. Once the process returns from the **get** operation it enters into its critical section. When leaving the critical section, the lock is released (or returned) to the EntitySpace via a matching **put** operation. If another process needs the lock, it must also issue a blocking **get** operation forcing it to wait until the process which controls the lock performs the matching **put** operation. Other popular lock semantics (such as the trylock primitive found in pthreads) are also possible, using the non-blocking **put** and **get** primitives defined in JavaNOW.

Note that this framework supports an unbounded number of locks since locks are nothing but Entities with pre-defined (and agreed upon) common keys. The following example illustrates the use of locks in JavaNOW.

```
public void initLock() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex", new Object());
  getJavaNOWAPI().put(esk, e);
}

public void lock() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex");
  getJavaNOWAPI().get(esk, e);
}

public boolean tryLock() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex");
  return (getJavaNOWAPI().getIfExists(esk, e) != null);
}

public void unlock() {
  EntitySpace esk = new
      EntitySpace("SYNC");
  Entity e = new Entity("mutex", new Object());
  getJavaNOWAPI().put(esk, e);
}
```

7.3 Semaphores

Semaphores are another useful synchronization mechanism that is supported in JavaNOW in much the same manner as the lock mechanism. The primary difference between the lock and the semaphore is in the initialization. In particular, N Entities (where N is the semaphore count), rather than a single Entity, are deposited in the EntitySpace. The down and up operations on the semaphore are performed using the **get** and **put** operations respectively using the semaphore key. Here is an example of how semaphores can be created and used in JavaNOW.

```
public void initSemaphore(int n) {
  EntitySpace esk = new EntitySpace("SYNC");
  for (int i=0; i<n; i++) {
    Entity e = new Entity("mutex", new Object());
    getJavaNOWAPI().put(esk, e);
  }
}

public void down() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex");
  getJavaNOWAPI().get(esk, e);
}
```

```
public boolean tryAllocateSemaphore() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex");
  return (getJavaNOWAPI().getIfExists(esk, e) != null);
}

public void up() {
  EntitySpace esk = new EntitySpace("SYNC");
  Entity e = new Entity("mutex", new Object());
  getJavaNOWAPI().put(esk, e);
}
```

7.4 Producer/Consumer

The Producer/Consumer problem is one of the most fundamental synchronization problems in computer science, and many problems in parallel computing degenerate into a special case of a producer/consumer relationship among tasks. Here is a solution to both the bounded and unbounded producer/consumer in JavaNOW.

```
// Unbounded Producer Process
void unboundedProducer() {
  EntitySpace esk = new EntitySpace("BUFFER");
  while (true) {
   Object data = produceData();
   Entity e = new Entity("key", data);
   getJavaNOWAPI().put(esk, e);
  }
}
// Bounded Producer Process
void boundedProducer(int maxBuffer) {
  EntitySpace esk = new
    EntitySpace("BUFFER");
  while (true) {
   while (getJavaNOWAPI().getSize(esk)
       == maxBuffer) {
      synchronized (this) {
        try {
          wait();
        } catch (InterruptedException e) {}
       } // synchronized
   }  // while buffer exceeds bound
   Object data = produceData();
   Entity e = new Entity("key", data);
   getJavaNOWAPI().put(esk, e);
  } // while forever
}

// Unbounded Consumer
. . .
void unboundedConsumer() {
EntitySpace esk = new EntitySpace("BUFFER");
  while (true) {
    Entity e = new Entity("key");
    String msg = (String) getJavaNOWAPI().get(esk, e);
  }
}
// Bounded Consumer
void boundedConsumer(int maxBuffer) {
EntitySpace esk = new EntitySpace("BUFFER");
  while (true) {
    Entity e = new Entity("key");
    String msg = (String) getJavaNOWAPI().get(esk, e);
    if (getJavaNOWAPI().getSize(esk) == maxBuffer-1) {
```

```
        synchronized (this) {
          notify();
        }
      } // if
    } // while
  }
```

8 Significance and Conclusions

This paper has presented JavaNOW, a framework for enabling both a message-passing and shared-memory model of computation. JavaNOW is designed primarily for networks of workstations, and provides a framework to harness the (relatively cheap) raw computational power available on such systems. Although written in Java, a relatively young and unproven language in many respects, JavaNOW has not been designed in a vacuum. Many of the ideas presented in this paper go back as far as the late 1960's and 1970's, when actors were first introduced as a model of computation. The emergence of Java has made it possible to bring a practical implementation to ideas such as coarse-grained actors and dataflow. The use of Java has also made it possible to compactly implement a powerful and robust library of communication primitives. In fact, the current version of JavaNOW numbers in the low thousands of non-commented lines of code (NLOC).

JavaNOW brings to the Java and High-Performance Computing community a framework that is reminiscent of the Message Passing Interface (MPI). However, many aspects of MPI are not necessary in a Java environment, such as derived data types and detached processes, since Java already supports these concepts quite well. Of course an important issue for any such framework is the level of support provided for the interoperability of existing codes. Our view is that CORBA (the Common Object Request Broker Architecture) provides a much better mechanism than the MPI framework for such interoperability. In particular, CORBA addresses the difficult issues of mapping data structures between dissimilar languages much better than the framework introduced by MPI. (MPI derived datatypes, for example, cannot handle aggregation of heterogeneous data structures very well.) Having said that however, JavaNOW provides full support for the collective communication and computation operations provided by MPI, which are often argued to be among the major research contributions of the MPI effort.

A question remains: Who would use JavaNOW? We believe JavaNOW will be of greatest use to two general classes of programmers:

- People who want to use Java to develop completely new codes and make use of all of the modern features available in the language.
- People who have existing kernels (FORTRAN and C) and wish to use JavaNOW as a coordination environment thus enabling the *reuse* of already proven codes. As an example, we have an active project underway to use JavaNOW as a coordination language for existing CFD codes. We have found that using the Java Native Interface (JNI), it is relatively straightforward to support such legacy codes.

All said, JavaNOW is very much a work in progress. It is, much like the rest of Java technology, a prototype. Thus we have been careful in this paper to focus on the innovations and flexibility of the JavaNOW model of computation rather than making claims about high performance of the current system (although we hope to be able to do so in the future). A number of improvements are planned for the next generation of JavaNOW, including the following:

Completeness: A valid concern has been raised during the review process that there have been few, if any, assurances of the completeness of the API in terms of what is needed for high-performance computing. The JavaNow system has been derived from the Linda and MPI systems. We have included all of the programming interfaces found in these environments and added a number of new elements with the emphasis on supporting *object-oriented* coordination in a Java-only context. Thus a number of features in MPI, such as derived datatypes, have been omitted, because Java (and other object languages) support derived data types as a part of the language. We have employed object-oriented design throughout the process of developing JavaNow and (in making decisions) have relied on a technique, known as factorization, to reduce functional complexity (without loss of expressive potential) and better exploit the Java language itself. We stress that claims about "language equivalence" and "completeness" often degenerate into formally undecidable problems and thus have made no such claims in this paper. We do claim, however, that our approach of a coherent small set of functions certainly lends itself better to practical use.

Performance: The current release of JavaNOW is implemented primarily as a proof of concept, and we have yet to pursue any significant performance tuning. Performance issues will be of paramount importance in future releases. In retrospect, many of our design decisions, such as supporting the easy plug-and-play of different transport layers, a component-based architecture, etc. do come at a cost. It may be the case that, in order to optimize performance, we have to retro-fit services back into the kernel in much the same way that has been done in some Microkernel operating systems such as Windows NT.

Dynamic resource management: The current release of JavaNOW requires that the user statically specify the list of machines on which the application will run. In the next release, JavaNOW will allow users to add/delete machines dynamically. This is an important step in the integration of JavaNOW with another framework we are developing termed the Computational Neighborhood (CN) (see [63] for a thorough discussion of this proposed framework). The CN supports dynamic resource discovery and allocation in a drag-and-drop manner allowing jobs to be started, seamlessly and transparently, on a collection of resources.

Dynamic load balancing: The current release of JavaNOW uses a simple hashing scheme for load balancing. While this approach seems to work well for the limited number of applications we have tested thus far, we do not yet know if it will work well in the general case. If not, then we will be forced to implement a dynamic load distribution scheme in the future.

Fault tolerance: As the number of machines participating in a JavaNOW application increases, the probability of a networking or other type of failure also increases. The current implementation of JavaNOW, similar to other message-passing systems, does not recover from such errors.

Availability: JavaNOW is available from the Java and High-Performance Computing Group, http://www.jhpc.org. For more information on JHPC, please contact George K. Thiruvathukal at the e-mail or postal address mentioned on the first page.

9 Acknowledgments

We wish to acknowledge the reviewers for their many constructive comments. We have endeavored to incorporate every suggestion made for improvement with the hope of producing an article of high quality. We also wish to thank Nina Wilfred, John Shafaee, and Arti Singh for their help in reviewing and editing the final version of this paper.

Bibliography

1 G. Aloisio, M. Cafaro, P. Messina, and R. Williams, “A distributed Web-based metacomputing environment,” *Proceedings of HPCN ‘97*, Vienna, Austria, April 1997.

2 T. E. Anderson, D. E. Culler, and D. A. Patterson. “A case for NOW,” *IEEE Micro*, February 1995.

3 H. E. Bal and M. F. Kaashoek, “Object-Distribution in Orca using Compile-Time and Run-Time Techniques,” *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’93)*, pages 162-177, Washington, D.C., 1993.

4 J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer, “ATLAS: An Infrastructure for Global Computing,” *Proceedings of 7th ACM SIGOPS European Workshop: System support for Worldwide Applications, Connemara, Ireland*, September 1996.

5 A. Baratloo, M. Karaul, H. Karl, and Z. Kedem, “An Infrastructure for Network Computing with Java Applets,” *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, February 1998.

6 R. Bjornson, C. Kolb and A. Sherman, “Ray Tracing with Nework Linda,” *SIAM News*, 1(24), January 1991.

7 R. Bjornson, “Linda on distributed memory multiprocessors,” Ph.D. Thesis, Yale University, 1992. YALEU/DCS/RR-931.

8 R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, H. Randall, and Y. Zhou, “Cilk: An Efficient Multithreaded Runtime System,” *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Parallel Programming*, 1995.

9 T. Brecht, H. Sandu, M. Shan, and J. Talbot, “ParaWeb: Towards World-Wide Supercomputing,” *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

10 D. Caromel, W. Klauser, J. Vayssiere, “Toward Seamless Computing and Metacomputing in Java,” Concurrency: Practice and Experience ed. by G. C. Fox, September-November, 1998, Wiley, pp. 1043-1061.

11 N. Carriero, and D. Gelernter, *Linda in Context*, CACM, 32:4, Apr. 1989.

12 J. B. Carter and J. K. Bennett and W. Zwaenepoel, “Implementation and Performance of Munin,” *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152-164, Oct. 1991.

13 N. Carriero and D. Gelernter. *How to write parallel programs*, The MIT Press, Cambridge, Massachusetts, pp. 45-49.

14 B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen, “HPJava: Data Parallel Extensions to Java,” *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, February 1998.

15 Z. Chen, K. Maly, P. Mehrotra, R. K. Vangala, and M. Zubair, “Web Based Framework for Distributed Computing,” *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

16 B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, and K. E. Schauser, *Javelin: Internet-Based Parallel Computing Using Java*. In Proceedings of ACM Workshop on Java for Science and Engineering Computation, Las Vegas, NV, June 1997.

17 W. T. O'Connell, G. K. Thiruvathukal, and T. W. Christopher, “Distributed Memo: A Heterogeneously Parallel and Distributed Programming Environment,” Proceedings of the 23rd International Conference on Parallel Processing, August 1994.

18 W. T. O'Connell, G. K. Thiruvathukal and T. W. Christopher. “The Memo Programming Language,” *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, October 1994.

19 P. E. Crandall and M. J. Quinn. *Data Partitioning for Networked Parallel Processing*, IEEE Press, 1993, pp. 376-379.

20 P. Dasgupta, Z. Kedem, and M. Rabin, "Parallel processing on networks of workstations; Fault-tolerant high performance approach," Proceedings of 15th IEEE International Conference on Distribute Computing Systems, 1995.

21 J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Integrated PVM Framework Supports Heterogeneous Network Computing," *Computers in Physics*. April 1993, Vol. 7, No. 2, pp 166-175.

22 The Original-E Extensions to Java, http://www.erights.org/.

23 A. Ferrari, "JPVM: Network Parallel Computing in Java," Proceedings of ACM Workshop on Java for Science and Engineering Computation, February 1998.

24 I. T. Foster and C. Kesselman, "The Globus Project," http://www.globus.org/.

25 G. C. Fox and K. Dincer, "Using Java and JavaScript in the Virtual Programming Laboratory: A Web-Based Parallel Programming Environment," *Concurrency: Practice and Experience*, 9:485-508, 1997.

26 G. A. Geist and V. S. Sunderam. "Network Based Concurrent Computing on the PVM System," *Journal of Concurrency: Practice and Experience*, 4, 4, pp 293--311, June, 1992.

27 G.A. Geist and V.S. Sunderam, "The Evolution of the PVM Concurrent Computing System," *Proceedings of 26th IEEE COMPCON Symposium*, pp. 471-478, San Fransisco, February 1993.

28 D. Gelernter, "Generative Communication in Linda," *ACM TOPLAS*, 7:1, Jan. 1985.

29 D. Gelernter, "Multiple tuple spaces in Linda," *In E. Odijk, M. Rem, and J.C. Syre, editors, PARLE '89: Parallel Architectures and Languages*, pages 20-27. Springer-Verlang, Lecture Notes in Computer Science Volume 366, 1989.

30 D. Gelernter and D. Kamisnsky, "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha," *Proceedings of Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992.

31 G. K. Thiruvathukal, "An Enhanced Actors Model for Parallel and Distributed Computing," *Proceedings of First International Conference on Parallel Computing (HiPC) 1994*, Bangalore India, December 1994.

32 G. K. Thiruvathukal, "An Enhanced Actors Model for Parallel and Distributed Computing," Ph.D. Thesis, Illinois Institute of Technology, Chicago, IL, 1995.

33 P. A. Gray and V. S. Sunderam, "IceT: Distributed Computing and Java," *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, June 1997.

34 A. S. Grimshaw and W. A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, pp. 39-45, volume 40, number 1, January, 1997.

35 S. C. Hupfer, "Melinda: Linda with multiple spaces," Technical Report YALEU /DCS/RR-766, Yale University, 1990.

36 L. V. Kale and J. M. Yelon, "Threads for Interoperable Parallel Programming," *Proceedings of Languages and Compilers for Parallel Computing*, 1996.

37 L. V. Kale, M. Bhandarkar, and T. Wilmarth, "Design and Implementation of Parallel Java with Global Object Space," *Proceedings of Parallel and Distributed Processing Technology and Applications*, Las Vegas, Nevada, 1997.

38 L. V. Kale, M. Bhandarkar, R. Brunner and J. Yelon. "Multiparadigm Multilingual Interoperability: Experience with Converse," *Proc. of Second Workshop on Runtime Systems for Parallel Programming (RTSPP)*, March 1998.

39 H. Karl, "Bridging the Gape between Distributed Shared Memory and Message Passing," *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, February 1998.

40 P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proceedings of the 1994 Winter Usenix Conference*, pages 115-132, January 1994.

41 LAM/MPI – Local Area MPI, http://www.mpi.nd.edu/lam/

42 K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proceedings of the 1988 International Conference on Parallel Processing*, pages II:94-101, Aug. 1988.

43 B. Gropp, R. Lusk and T. Skjellum, *Using MPI: Portable Parallel Programming with the Entity-Passing Interface*, 1994.

44 L. Clarke, I. Glendinning, and R. Hempel, MPI: A Message-Passing Interface Standard, *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3), 1994.

45 H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, U. Nagashima, Concurrency: Practice and Experience ed. by G. C. Fox, September-November, 1998, Wiley, pp. 1063-1078.

46 MPI-2: Extensions to the Message-Passing-Interface, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

47 MPI-Java Home Page, http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

48 J. Narem. "An Informal Operational Semantics of C-Linda V2.3.5," Technical Report 839, Yale University Department of Computer Science, Dec. 1990.

49 M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, P. Capello, "Javelin++: Scalability Issues in Global Computing," Proceedings of the ACM Java Grande 1999 Conference, June 12-14, 1999, San Francisco, California.

50 S. W. Otto, M. Snir, and D. Walker. "An Introduction to the MPI Standard" In J. Dongarra, CS-95-274, January 1995.

51 M. Philippsen and M. Zenger, "JavaParty: Transparent Remote Objects in Java," *Proceedings of the ACM PpoPP Workshop on on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

52 M. C. Rinard, D. J. Scales, and M. S. Lam, "Jade: A High-Level, Machine-Independent Language for Parallel Computing," *IEEE Computer*, 1993.

53 L. F. G. Sarmenta, S. Hirano, and S. Ward, "Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java," *Proceedings of the 2nd Intl. Conference on Worldwide Computing and its Applications*, Tsukuba, Japan, March 1998. http://www.cag.lcs.mit.edu/bayanihan.

54 K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

55 Seyfarth, J. Bickham and S. Arumugham. *Glenda*, http://sushi.st.usm.edu/~seyfarth/research/glenda.html

56 A. Sinha and L. V. Kale, "A Load Balancing Strategy For Prioritized Execution of Tasks," *International Symposium on Parallel Processing*, Newport Beach, CA, April 1993.

57 Ahuja, Sudhir, Carriero, and Gelernter, "Linda and Friends," *IEEE Computer*, Aug. 1986.

58 V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, 2, 4, pp 315--339, December, 1990.

59 G. A. Geist and V. S. Sunderam, "Network Based Concurrent Computing on the PVM System," *Journal of Concurrency: Practice and Experience*, (4), pp. 293-311, June 1992.

60 V. Sunderam, J. Dongarra, A. Geist, and R Manchek. "The PVM Concurrent Computing System: Evolution, Experiences, and Trends," *Parallel Computing*, Vol. 20, No. 4, April 1994, pp 531-547.

61 Sun MicroSystems, Inc., *JavaSpaces Specification*, http://java.sun.com/products/javaspaces/

62 G. K. Thiruvathukal, "Toward non-von Neumann Computation: An Enhanced Actors Model for Parallel and Distributed Processing," *Proceedings of the First HiPC Conference (Workshop)*, Bangalore, India, 1994.

63 G. K. Thiruvathukal, B. Cameron, T. Christopher, L. Oliveira, and J. Shafaee, The Computational Neighborhood, *Proceedings of ICS Workshop on Java,* Rhodes, Greece. To appear in a special issue of FGCS, edited by V. Getov.

64 D. A. Thurman. *JavaPVM: The Java to PVM Interface*, Decemeber 1996. http://www.isye.gatech.edu/chmsr/jPVM.

65 W. Yu and A. Cox. "Java/DSM: A Platform for Heterogeneous Computing," *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.