An Open System for SPMD Programming

Nenad Stankovic
FSJ Inc., Daiwa Naka-Meguro Bldg. 5-8F, 4-6-1 Naka-Meguro
Meguro-ku, Tokyo 153-0061, Japan
nstankov@ics.mq.edu.au

Abstract

We present here our work aimed at developing an open, network based visual software engineering environment for parallel processing called Visper. It is completely implemented in Java and supports the message-passing model. Java offers the basic platform independent services needed to integrate heterogeneous hardware into a seamless computational resource. Easy installation, participation and flexibility are seen as key properties when using the system. We believe the approach taken simplifies the development and testing of parallel programs by enabling modular, object oriented technique based on our extensions to the Java API.

Keywords: MPI, PVM, message passing, performance, Java

1. Introduction

Wide and local area networks represent an important computing resource for parallel processing community. The processing power of such environments is deemed huge. One of the problems is that they are made up of heterogeneous hardware and software. In this paper we focus on describing a novel metcomputing environment named Visper. The aim of the project is to allow a programmer to make an efficient use of networked computing resources by employing techniques such as visual program composition and analysis, checkpointing and fault tolerance. Visper is implemented in and for Java [16] that, due to its platform independence and uniform interface to system services, simplifies the implementation of SPMD parallel applications and the system software needed to support them.

1.1 Native Message-Passing Systems

A number of tools and libraries have been designed to foster the use of networks to run parallel programs. In the message-passing domain, products like the MPI [19] and the PVM [15] transform heterogeneous networks of workstations and supercomputers into a scalable virtual parallel computer. While MPI is focused on message passing aspects, PVM acts as a runtime system. The aim of both is to provide a standard and portable programming interface for several high level languages. They have been adapted to a variety of architectures and operating systems. Both systems require programmers to build a different executable for each target architecture or operating system and programs must be started manually. The insufficient security and dependence on shared file system limits their use for large or widely distributed applications. For example, while it is possible for a PVM process to run code that is loaded from other network nodes, the virtual machine provides no protection for its hosts against malicious or errant behavior from the downloaded code.

To enable message-passing programming across intranet boundaries, on a global scale, a number of systems have been developed. Globus [14, 21] is a popular toolkit that comprises a set of interrelated components for wide area computing. As part of the project, the MPICH-G library was developed to run MPI on a wide area network using the same standard MPI commands. Nexus [12] is a communication library that enables transparent accessibility of remote computers with support for lightweight threading, communication, synchronization and address space management. Legion [18, 21] is a powerful system that provides both greater throughput and improved response time via parallel execution. It builds system components on a distributed objects model. Unlike many other systems, security is an integral component of the metacomputing core, providing users with mechanisms to select policies that meet their needs without jeopardizing local administrative requirements.

1.2 Java Programming Model

Java is an object-oriented (OO) language whose syntactical structure is similar to C++. The language model, though somewhat limited, finds its strongest point in its simplicity. With the advent of Java and the Internet the world of computing has been enriched by a new programming technique that is inherently driven by platform independence. This explains why Java should not be recognized merely as yet another programming language but rather as a concept or an environment. Java is a sequential programming language that features object-oriented concepts and preemptive multithreading, where thread methods offer a set of synchronization primitives based on the monitor paradigm by C.A.R. Hoare [20]. In Java, the *synchronized-wait-notify* construct provides such a high-level monitor.

Java threads, however, do not have built-in point-to-point communication primitives needed for message-passing parallel programming. Instead, the API provides general concepts for implementing communication primitives that are very flexible. Thus created freedom allows various kinds of communication facilities to be used. Java binaries are shipped across a network and executed on client machines. Security is therefore a critical issue and strongly enforced in Java. A Java interpreter executes binaries at client side. Hence, the model is safe since the sent bytecodes are verified at the client that prevents corrupted or evil classes from causing problems [41]. Based on these premises we can extend the network OO programming from the procedural one based on RPC or RMI [40] to the class one based on the fact that classes and objects can be sent over a network rather than plain data or a method invocation. Moreover, it is possible to preserve object's state, to initialize it, etc. We can think of classes (threads) that are distributed over networks, rather than just methods that execute on remote hosts.

While being a concurrent multithreaded language with dynamic (remote) loading and linking capabilities, Java offers little help for distributed programming. Objects and classes in Java are mobile. They may be unknown to the receiver upon reception, but since new classes can be added to a running program, objects downloaded from network can be resolved and linked in at runtime. Thus, dynamic linking can be employed to implement elegantly mobile code systems allowing, for example, dynamic system configuration or programming. However, Java threads are not mobile, since thread migration is not supported. This means that both data and code can be moved or transmitted from one machine to another at any point in time, but not the current state of execution, if any, associated with the object.

1.3 Motivation

Distributed and parallel programming have been a topic of active research for some time with the main areas of interest being networking and architecture related issues. In 1995 the Pasadena Working Group #7 has drawn out the following recommendations regarding tools for high-performance computing environments [26]. Efficient support for (object-oriented) programming on parallel architectures in particular requires elaborate programming tools. The aim of the research is to support high-bandwidth, low-latency messaging, and transparent marshaling and demarshaling of message content. The ability to instantiate or schedule processes (threads) on shared and distributed memory architectures with low overhead is also important, together with support for the right division of work between various software layers and employed machines. This work needs to be progressively extended to address the more general issues of an infrastructure for parallel objects in an environment that is not dedicated to one application and is controlled by external agents, and therefore extensible. In our research, we have followed the Pasadena Workshop recommendations.

Even though Java was not designed for parallel programming, it is not hard to identify the reasons to implement a parallel application in Java. As an OO language, Java provides us with strong typing, and the *objects everywhere* feature that has as few implementation dependencies as possible. Java code is compiled to a form that runs on any architecture that implements the Java Virtual Machine (JVM) [23]. Therefore, with Java, the architecture dependent problems of metacomputing have been resolved since the environment and the API are inherently architecture independent. However, the price for that has been in slow execution speed of Java programs. This problem has been addressed by the just-in-time and HotSpot compilation techniques, with some encouraging results being reported [36]. The Java Native Interface [38] and support for CORBA [37] enables interoperability with native code for better performance or reuse. Security is part of the language as well as of the environment. The lack of pointer arithmetic and garbage collection allows fast prototyping. The missing support for parallel processing can be implemented either by adding new keywords to the language, or by extending the API. The implementation of the missing functionality to enable efficient group operations (e.g. control and method invocation) and message-passing parallel programming while exploiting the advanced features of pure Java is another aspect of our research.

Visper is conceived as an integrated metacomputing environment that provides services to design, develop, test and run parallel programs. It provides an MPI like communication library, and features to spawn and control lightweight processes from within the environment. In this paper we will expand upon the mentioned points. Section 2 describes the organization and the main services supported by the environment. Section 3 describes the API and the programming model, while Section 4 provides various performance data. Section 5 informs on the related metacomputing work in Java. A description of older systems can be found in [2]. The paper concludes with Section 6.

2. Visper

Visper is an interactive, object-oriented environment implemented in Java, with a set of tools for construction, execution and testing of SPMD applications. It is conceived as a tool for research into parallel and distributed Java programming, and as an efficient, reliable and platform independent environment that produces useful results. It can be used by multiple users and run multiple programs concurrently. Figure 1 shows the tool architecture and the main components. Visper consists of a frontend (i.e. console) and a backend (i.e. system services). The purpose of the frontend is to implement the graphical user interface and to encapsulate as much of the syntax of the model as possible. The backend implements the semantics of the model independent of the front design. The overall system operation can be described in terms of the two components.

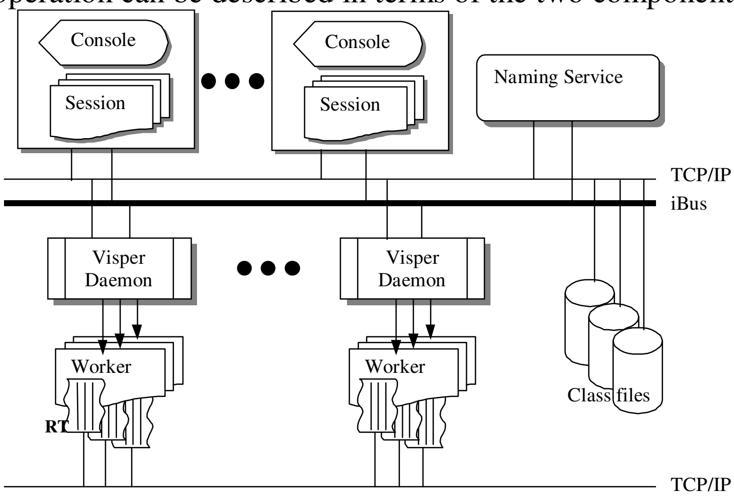

Figure 1 Architecture

Visper is built on top of two communication techniques: a reliable point-to-point (TCP/IP), and an intranet multicast (iBus [31]). Both techniques can be used simultaneously, with iBus being optional. We have enhanced the iBus with a routing subsytem that transfers iBus messages across domain boundaries. Each user starts a console that represents the frontend to the system. The console provides the means to compose, start and analyze parallel programs. The user than creates one or more sessions, where each session runs only one parallel program at a time. A session is an ordered collection of hosts and represents a virtual parallel computer. It allows control and data collection of a running program. It can dynamically grow or shrink while a program is running, since hosts can join or leave (i.e. crash). The host that dynamically joins a session, remains part of it until manually removed by the user.

The backend consists of the services to run parallel programs and generate debugging and runtime data. The naming service represents a database of available resources, sessions and acts as a port-mapper and detector of faulty machines for point-to-point mode. Each machine that is part of a Visper environment must run a Visper daemon. Each daemon forks one worker process per session that runs parallel programs, and maintains workers' state (e.g. active, dead). To minimize the start-up cost, each session maintains its set of workers as long as active. As presented in Figure 1, workers can run multiple remote threads (RT). To allow location transparent programming and more efficient loading of Java class files (i.e. active and passive objects), the user can define multiple loading points, and different access modes (e.g. http://…, file://…).

2.1 The Console

To a programmer, the console is the only tangible component of the tool. The purpose of the console is to enable user-to-system interaction and to implement the syntax of the programming model or program specification. Its main strength is in the visual parallel program composition and in the visualization of program execution. Thus, it is the console for the system in which the user constructs a parallel program, then exercises it through an interface.

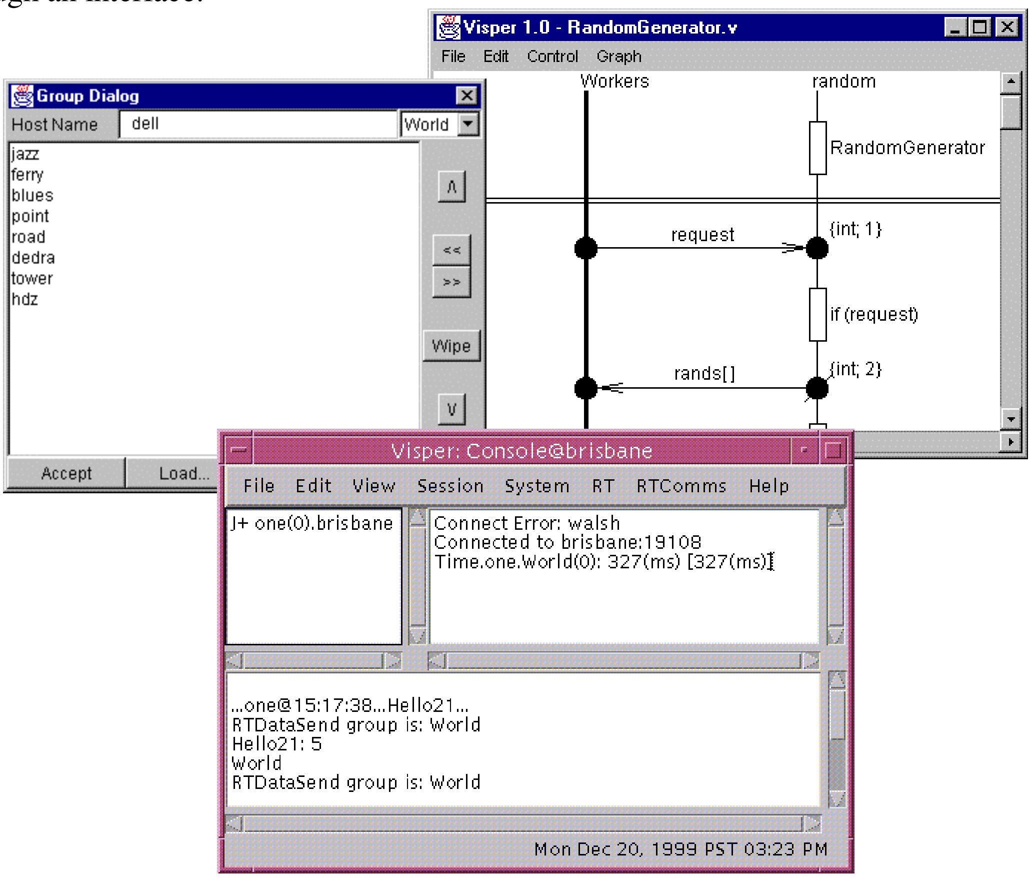

Figure 2 Frontend Components

Figure 2 shows various frontend components. For example, the frontend provides a visual programming environment with a pallet of model primitives that can be placed on a canvas and which can be interconnected using arcs to create a Process Communication Graph (PCG), details of which can be found in [32, 34]. The frontend also generates a structured internal representation of the model, performs syntactic analysis of the graph the programmer is constructing, and translates the graph into a Java program. Finally, the frontend is the user interface to the backend that allows program execution, control and debugging.

To run a program, the user first connects to a Visper daemon, from where the currently available hosts are obtained. In a group dialog, the user then creates one or more sessions. Hosts can be shared between sessions, as each session has its own set of workers. Then, a system wide path is defined that instructs the workers where to look for the application class files. Finally, in an execution dialog, the user defines input arguments and starts a program within a selected session. These operations do not require special scripts or valid user accounts on the involved hosts. When running a program, the run-time data collection can be switched on and off, and later analyzed in a space-time diagram (some initial results have been reported in [22]). To help with debugging, the system also intercepts the exceptions generated by the parallel program and displays it in the console. If they are system related, they are displayed in the top right window, and if program related, then in the bottom window where program output gets displayed. The top left window displays backend messages initiated by user actions (e.g. *J+* means the host has joined the session). Console does not yet provide a system wide interface to jdb.

2.2 The Backend

The backend consists of a naming service and a set of Visper daemons, resource managers and workers running on a network or networks of computers (Figure 3). The backend reacts to directives from a console, then notifies the frontend of the changing program status as the execution process takes place. At a direction from a frontend, the backend will report information regarding program activity and host allocation per session, and generate space-time diagram data.

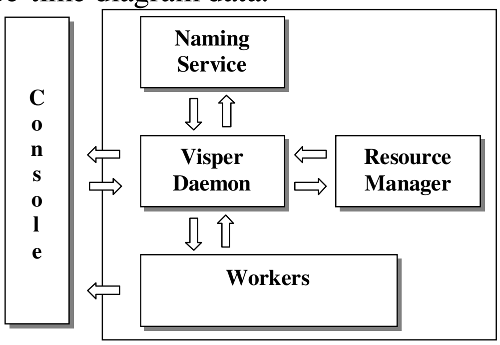

Figure 3 Backend Control Flow

Visper is built on top of the socket API that comes with Java and the iBus intranet middleware. Java sockets provide the basic reliable point-to-point communications by TCP/IP protocol, and local intranet or subnet boundaries do not affect their scope. The problem with this approach is its limited scalability. Ethernet and Token ring based networks allow efficient hardware multicast facilities. However, in Java they are not reliable and therefore we use iBus that enables a model of spontaneous networking, where applications can join and leave reliable multicast channels dynamically. Along with the multicast capabilities, iBus also provides a simple interface to receive membership changes and failure detection. When a view change occurs, either due to a new node joining or leaving the channel, the iBus membership protocol detects it and delivers a view change notification to the registered listeners.

2.2.1 Visper Daemon

A Visper daemon (or just daemon for short) is a standalone Java process that enables its host machine for processing in Visper. Upon startup, each daemon registers with the naming service, and forks one worker that is initially not assigned to any session. If the daemon is designated as an iBus router, it creates a thread that forwards iBus messages across domain or subdomain boundaries to the designated daemons via TCP/IP channels. After that, the daemon is ready to accept requests from a console or other backend components. Requests are processed by a daemon agent, installed at startup. The default agent can be replaced without restarting the daemon.
When a user creates a session, the console broadcasts the user-defined group of hosts into the environment. Each affected daemon assigns the non-assigned worker to the session, and then forks a new non-assigned worker to speed up future new session requests. Daemons use the *java.lang.Process* class to control and communicate to the workers. To simplify the design, console-to-worker communication is directed through a parent daemon.

2.2.2 Resource Manager

Each Visper daemon is accompanied by a resource manager. Similar to the Visper daemon, the resource manager implements only the basic synchronous and asynchronous message-passing services, and a message handler that understands system-related messages. Resource managers are configured by the agents that implement different resource allocation policies. By default, each resource manager offers two allocation policies: the random and the round robin. New policies can be added dynamically, at runtime. When a resource manager adds a new host to a session, the parent console gets informed about the change.

2.2.3 Worker

A worker is a standalone Java process that executes remote threads. From the system's perspective, a worker represents a virtual processor with memory. Workers are persistent, meaning that as long as the session they belong to is active, they remain active, too. Therefore, they are capable of storing the bytecode locally across multiple executions. Workers load bytecodes by a customized class loader. There is only one class loader per worker. It extends *java.lang.ClassLoader* by allowing multiple loading modes and points. When testing a program, the user can reload the stored classes without restarting the worker, by creating a new loader. Workers can execute concurrently multiple remote threads, if those threads belong to different groups. If not, they are executed as a FIFO. Each worker provides a default communication channel for the parallel programs it runs, which eliminates the need for an initial synchronization among a group of running remote threads (see Section 3.6.2).

3. Programming Model

A program in Visper is a union of three (orthogonal) entities: the code, the hosts, and the class files location. From the programmers' perspective, Visper provides a simple API that enables object communication and parallel processing. In Figure 4, the dashed lines designate inheritance, and the solid lines represent aggregation. The system-level classes are prefixed with a *V* (for Visper), and the user-level classes with a *RT* (for remote thread). Most RT classes are just wrappers for the corresponding V superclasses. Their purpose is twofold. First, to abstract away implementation issues by providing a simple and coherent API. Second, to present the API as just one package (*visper.rt*), rather than the programmer having to deal with the internal package structure.

A *RTSession* is an object allocated by the system that keeps track of all the resources allocated by a program, and contains information about the current session status and configuration. For each run, there is only one session object. The *RTComms* is an MPI like library that provides point-to-point, collective, synchronous and asynchronous communication and synchronization primitives. The group and process package implements classes to group remote threads, to provide communication scope, and to create and control them from within a program. *VProcess* is a system class that represents a controllable virtual process. *RTRemoteProcess* and *RTRemoteGroup* provide methods to spawn and control one or more remote threads. The *RTGroup* (*VGroup*) class of objects provides primitives to order processes. It is unique within its session, representing the smallest unit of process organization in Visper. The *RTCheckpoint* class implements checkpointing.

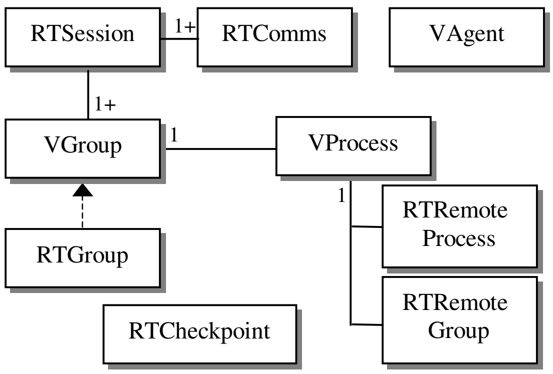

Figure 4 API

As a metacomputing environment, Visper is a collection of daemons that are dynamically configured and controlled by agents. All agents are derived from the *VAgent* class. Agents are system-level, self-contained, interactive and autonomous objects. They communicate by (a)synchronous messages that are offered by the system. Upon arrival, the host daemon allocates a Java thread for the agent to run on, and informs the agent about its message queue.

3.1 Message-Passing Primitives

At the application level, Visper provides a TCP/IP based communication class called RTComms that enables direct process-to-process communications in pure Java. The

RTComms style follows the MPI standard, but in its implementation is driven by the features of and services provided by Java. Unlike MPI, all the primitives take a user defined tag. We allow intergroup collective communications. The design is partially a choice of simplicity over efficiency. While the Java Object Serialization [39] enables simple programming, it is reportedly slow [4, 28] (also Section 4.2). Further, RTComms hides the addressing and communication mechanisms from the programmer. Issues, such as the establishing of communication channels and the handling of network exceptions are performed internally, therefore making the programmer's code smaller than the corresponding socket version. For brevity, we list below some of the implemented methods. (For performance reasons, buffered MPI primitives were not implemented.)

```
public class RTComms {
	public Object[] AllGather(RTDataGroup dg,Object data);
	public Object[] AllGather(RTDataGroupInter dg,Object data);
	public Object[] AllReduce(RTDataGroup dg,Object data,RTDataOp reduceOperation);
	public Object[] AllReduce(RTDataGroupInter dg,Object data,RTDataOp reduceOperation);
	public Object[] AllToAll(RTDataGroup dg,Object data);
	public boolean Barrier(RTDataBarrier db);
	public boolean Barrier(RTDataBarrier db,int count);
	public Object Bcast(RTDataBroadcast db,Object data);
	public boolean Cancel(VData d);
	public Object[] Gather(RTDataGroup dg,Object data);
	public Object[] Gather(RTDataGroupInter dg,Object data);
	public void Probe(RTDataSend ds);
	public boolean Probe(RTDataSend ds);
	public boolean ProbeAsync(RTDataRecv dr);
	public boolean ProbeAsync(RTDataSend ds);
	public Object Recv(RTDataRecv dr);
	public RTDataRecv RecvAsync(RTDataRecv dr);
	public Object Reduce(RTDataGroup dg,Object data,RTDataOp reduceOperation);
	public boolean Send(RTDataInter dr,Object data);
	public boolean Send(RTDataSend ds,Object data);
	public RTDataSend SendAsync(RTDataSend ds,Object data);
	public boolean SendInter(RTDataInter di,Object data);
	public Object SendRecv(RTDataSendRecv dsr,Object data);
	public Object Test(RTDataRecv dr);
	public boolean Test(RTDataSend ds);
	public Object[] WaitAll(RTDataRecv[] dr);
	public boolean[] WaitAll(RTDataSend[] ds);
}
```

RTComms supports blocking and non-blocking point-to-point and collective message-passing primitives in a raw and trace mode. Messages can be sent within a group or among different groups. The synchronization capabilities are represented by the barrier method. There are two variants, one that waits for all processes in a group and the other that releases the block after the specified number of processes have made the call. All the methods follow the same pattern, where first argument represents the message envelope, and second argument represents the data. Similar to MPI, the programmer is reasoning in terms of transparent process identifiers, rather than IP names and ports. However, the RTComms is an object based communication library taking advantage of the Java Object Serialization mechanism, method overloading, and it does not support directly native data types. The benefit of this approach is that the programmer does not have to define the type of the data being sent to the system. The drawback is that the programmer must use a wrapper object to send native types.

As in MPI, the following attributes define a message envelope: the tag, the process identifier, and the group. They are combined into an object of the *VData* class. Figure 5 presents the (simplified) hierarchy of envelope classes used by the communication methods.

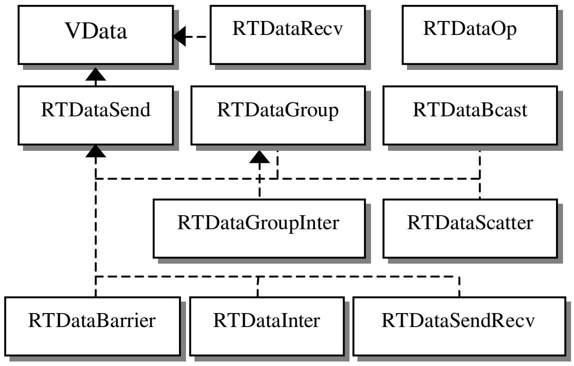

Figure 5 Envelopes

VData is an abstract class that defines methods to match itself against other envelopes.

```
public abstract class VData implements java.io.Serializable,Cloneable {
        VData(int tag,int sendToID,String sendToGroup,int recvFromID,String recvFromGroup);
        public abstract boolean Match(VData d); // exact match
        public abstract boolean MatchAnySource (VData d);
        public abstract boolean MatchAnyTag(VData d);
        public abstract boolean MatchAnyTagSource(VData d);
}
```

The constructor takes a message tag and 4 arguments that fully define the scope. For an intragroup message the *RTDataSend* defines the *tag* and the *sendTo** part, while the *RTDataRecv* defines the *tag* and the *recvFrom** part (see Section 3.6.1), and the system adds the missing data, by default. Each subclass implements the match methods, depending on the desired behavior. For example, the *RTDataBcast* class returns *false* for all the *MatchAny** methods, allowing only the exact match to return a valid result. On the other hand, a point-to-point send allows matching based on all the criteria. For intergroup messages the *RTDataInter* requires all the 5 arguments to be coded. *RTDataOp* defines the reduction operation for *Reduce* methods.

3.2 Remote Thread

Remote threads are the basic building blocks when programming in Visper. They can be described as protocol and platform independent components that dynamically extend the functionality available at remote hosts. Similar to *applets* they are Java application components that are downloaded, on demand, to a part of the environment where they get scheduled to run on a Java thread allocated by the worker. Remote threads have the Java advantage: memory access violations and strong typing violations are not possible, so that faulty remote threads will not crash processes the way that is common in most C language environments. A remote thread is any class that implements the *RTRunnable* interface:

```java
public interface RTRunnable extends java.io.Serializable {
        public void Run(String[] args,RTThreadGroup rttg,RTSession rts);
}
```

This solution has been chosen to promote code reuse, since Java does not support multiple inheritance. It allows any Java class to be turned into a remote thread simply by implementing the interface, rather than forcing the programmer to inherit from a remote thread class. Therefore, a remote thread is a class, but not all the classes that a remote thread makes use of or reference are remote threads. They, nevertheless represent this remote thread context. Upon creation and initialization by the system, such an object executes its *Run* method. This method defines the remote thread script or body: the sequence of actions that it executes while alive. It takes three arguments. The first argument contains the input arguments (if any) defined by the user. The second argument is used in those cases when the program makes use of Java threads. All the spawned Java threads should belong to that thread group for the system to block before all of them join. The third argument represents the session on which the program executes. When Run terminates, the remote thread terminates, too. Remote thread termination is unified with garbage collection, making it unreachable.

3.3 Remote Process and Remote Group

Remote threads provide us with the basic mechanism for parallelism that makes a Java class runnable in Visper. However, we still need a structured and constrained parallel-programming model that will allow remote instantiation, execution and control of tasks. Object-oriented software development methods deal with concurrency according to either the implicit concurrency model or the explicit concurrency model. In the implicit model, the objects themselves have concurrent execution capabilities. Each object represents an autonomous unit that performs a requested service concurrently with other objects. In the explicit model, there are two abstraction concepts: processes and objects. Objects are encapsulated inside processes, the latter providing concurrent execution capabilities by allocating the resources the objects will make use of.

Due to the design of Java described in Section 1.2 our concurrency model is explicit. Along with being explicit, our model can be classified as sequential since each process has only one (remote) thread of control. Consequently, there is a one-to-one mapping between remote threads and processes. As shown in Figure 4, the model comprises four classes. VProcess is a system class that defines methods to control a remote thread. The methods are simple as they only return control protocol objects. It has three attributes: the group, the remote thread class name, and the input arguments.

```java
class VProcess implements java.io.Serializable {
        VMessage Invoke(java.lang.reflect.Method method,Object[] args);
        VMessage IsAlive();
        VMessage Join();
        VMessage Migrate(int processID);
        VMessage Resume();
        VMessage Start();
        VMessage Stop();
        VMessage Suspend();
}
```

RTRemoteProcess and RTRemoteGroup allow asynchronous creation of remote threads. Remote threads are started automatically when scheduled for execution, by invoking their

Run methods. Both classes implement the same interface together with the communication mechanisms between a parent thread and a child thread or threads. RTRemoteProcess uses TCP/IP, and RTRemoteGroup uses iBus multicast if not talking to an individual remote thread. RTRemoteProcess has two attributes: the VProcess, and the process ID. RTRemoteGroup has two attributes: the VProcess, and the list of process IDs relative to the group, if not homogeneous.

```
public class RTRemoteProcess implements java.io.Serializable {
	public RTRemoteProcess(Class rtclass,String[] args,RTGroup rtg,int processID);
	public boolean Invoke(java.lang.reflect.Method method,Object[] args);
	public boolean Invoke(String method,Object[] args);
	public boolean IsAlive();
	public boolean Join();
	public boolean Migrate(int processID);
	public boolean Resume();
	boolean Start();
	public boolean Stop();
	public boolean Suspend();
}
```

The RTRemoteProcess constructor takes three arguments. The *processID* defines the host relative to the group on which to spawn a new remote thread. When an object owns a reference to a remote process or a group of processes it is able to control it by invoking methods. To simplify programming, all the interface methods return a *true* or *false* status. For example, to test if a remote process is still alive (either active or suspended), we use:

```
	IsAlive()
```

Explicit synchronization with remote process termination is provided by the *Join* method. It blocks until the processes or the group of processes completes its execution, i.e. exit Run. The *Invoke* method enables method invocation on remote threads. This mechanism is based on the Java Reflection API [35], which is an example of structural reflection that gives us an insight into structural aspects of the classes and objects in the current JVM. If the security policy permits, the Reflection API can be used to construct new class instances, invoke methods on objects and classes, etc. A more transparent model of method invocation would require a preprocessor similar to the one used by Java and RMI. It, nevertheless, allows invocation of methods on processes as well as groups. Also, any process that holds a reference to another process may invoke methods on it. A subset of the methods defined by the VProcess class is implemented by the console to control programs on a world basis (e.g. Stop, Resume).

3.4 Group

A group is an ordered set of processes. It is the smallest form of resource organization in Visper. A process is instantiated as a member of a group. Groups can be static (i.e. defined at compile time) or dynamic (i.e. allocated by a resource manager). Groups are unique, meaning that within a session no two groups can share the same name. They are system-wide objects, meaning that if a host crashes, all the groups that reference that host will get notified. RTGroup exports the group size, name, and process ID relative to itself. By default, each session has a group called *RTWorld* (Figure 6), that comprises all the processes employed by the session, whether allocated statically or dynamically. This differs from the MPI-2 [24] model in which an intercommunicator binds a dynamically

created group to the static. Groups may have a virtual topology, represented by the *RTGraph* and *RTGrid* classes (not implemented yet).
Visper provides two more groups to simplify programming with RTRemoteProcess and RTRemoteGroup constructs. The idea is to have a programming model that does not depend on a specific naming convention, or forces the programmer to pass configuration information when allocating new processes. One group is called *RTParent*, and has only one member that is the process that has created the remote process or the remote group of processes. The other group is called *RTLocal* and is an alias for the actual group the process belongs to. Both classes are pseudo groups, since they do not allocate resources.

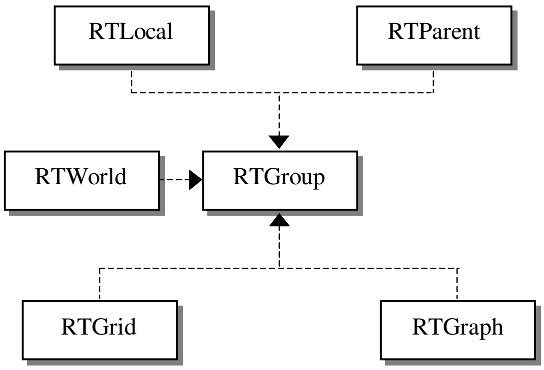

Figure 6 Groups

For example, in a master-worker scenario, a process *P1* from *Group1* creates a group called *Group2*, and populates it with processes (Figure 7). The *RTRemoteGroup* object is used when referencing the remote threads it has spawned. After the workers from *Group2* perform some calculations, *P1* collects the results via an intergroup gather.

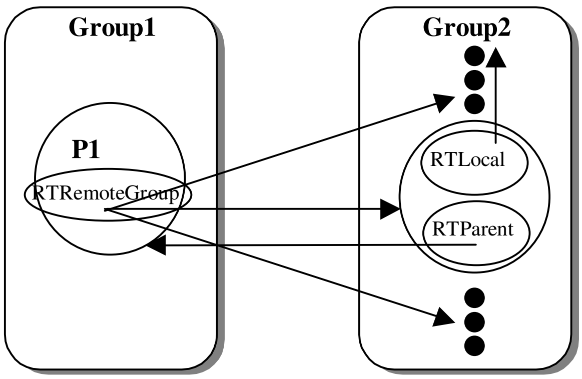

Figure 7 Local and Parent Group

The following program excerpts implement the master part:

```
RTGroup Group2 = new RTGroup(rts); // create new group in this session (rts)
RTRemoteGroup rg = new RTRemoteGroup(Worker.class,null,Group2);
Object[] obj = comms.Gather(new RTDataGroupInter(10,P1,Group1,Group2),null);
```

and the worker part:

```
public class Worker implements RTRunnable {
    Object result = …; // calculate result
    RTGroup parent = new RTParent(rts); // get parent, i.e. P1 of Group1
    RTGroup local = new RTLocal(rts); // get local group, i.e. Group2
```

```
comms.Gather(new RTDataGroupInter(10,0,parent,local),result);
}
```

The Gather primitive reads as: collect all the results with tag *10* from *Group2* at process *P1* of *Group1*. Process *P1* does not add anything to the result as *null* is passed into the gather. At the worker, the system automatically maps group *local* to *Group2*, and process *0* of group *parent* to process *P1* of *Group1*.

3.5 Remote Thread Migration and Checkpointing

In distributed programming, process migration is important since it allows processes to be restarted from a known state in a different address space. Depending on the application, it can be used either to offload the host, or to intentionally continue execution on a new host. Remote threads can migrate only within the same group. Before a migrated thread is restarted, all the system wide references to it must be updated automatically. In a parallel-programming environment where programs take long time to complete, we are also interested in the ability to recover the application in the case of a failure. For a standalone remote thread that is idle or wants to migrate itself, the scenario is simple since its state is known at the time of migration, and the migration can be performed immediately. For a communicating/interactive remote thread, however, we can only restart the thread from a preserved state, not the current since it is unknown.

To support migration and fault tolerance, Visper provides a checkpointing mechanism that allows saving state of an object periodically. Checkpointing is implemented at high level as an API class called RTCheckpoint, since the specification of the JVM prohibits a migrated object to be restarted from the last execution point transparently. Checkpoints are inserted at appropriate places in the code, and the already computed operations must be skipped over when the remote thread resumes execution from a snapshot state in a new worker. Multiple checkpoint objects may be created, each with a unique name. Each checkpointed object must have a unique name that is used to recover it. By calling commit, the checkpoint is marked as complete. Subsequent calls to write form a new checkpoint.

```
public class RTCheckpoint {
	public RTCheckpoint(RTSession rts,String name);
	public boolean Commit(); // mark checkpoint as complete; also a hint to save checkpoint
	public boolean Initialize(); // initialize checkpoint
	public boolean Recover(); // get last consistent program checkpoint
	public boolean RecoverLast(); // get last process checkpoint
	public Object Read(String name);
	public boolean Write(String name,Object obj);
}
```

To accommodate for the standalone case, the complete remote thread is checkpointed upon a request for evacuation, by default. To minimize the cost in time, the process is synchronous and unbuffered, since the mechanism does not make or maintain in-process copies of the checkpointed objects. The data are passed locally, through a socket, to the parent Visper daemon that implements the optimistic checkpointing policy where each process takes checkpoints independently [10]. The daemon may either buffer in memory or save checkpoints to file. A message is sent to flush the buffers when the process they belong to terminates. Upon recovery, the system tries to construct a consistent program state from the files. Both, the checkpointing and the evacuation are based on the Object Serialization mechanism and stream compression filters, and they require that all objects registered with the fault tolerance mechanism are serializable. Objects are serialized to a byte array to relive the daemon from downloading the class files when reading (i.e. deserializing) the passed data.

3.6 Example Programs

Visper console provides two modes of execution: the MPI mode and the dynamic RT. In the MPI mode, the system executes a requested remote thread by sending it to all the workers in a session (i.e. World). This resembles the static SPMD style as in MPI 1. In the RT mode the console sends requests only to the first process in a session (the one with ID 0 relative to the World). That remote thread acts like a bootstrap. To populate other processes in a session, we use a RTRemoteProcess or a RTRemoteGroup. It is important to notice that when writing a program, these modes are not relevant. They are simply primitives provided by the console to simplify program deployment.
Here, we present a simple example program. It consists of two processes, one that sends a message and the other receives it. The receiving process sends a message to the console that displays the number being sent.

3.6.1 The MPI Execution Mode

The MPI execution mode version consists of only one class called *SendReceiveMPI*. It implements only the Run method as dictated by the RTRunnable interface.

```
import visper.rt.*;
public class SendReceiveMPI implements RTRunnable {
        public void Run(String[] args, RTThreadGroup rttg,RTSession rts)
        {
                RTComms comms = new RTComms(rts); // use default communications
                RTWorld world = new RTWorld(rts); // use default group
                if (world.HostID() == 0) {
                        RTDataSend ds = new RTDataSend(10,1,world);
                        comms.Send(ds,new Integer(5)); // blocking send
                } else if (rts.HostID(world) == 1) {
                        RTDataRecv dr = new RTDataRecv(10,0,world);
                        Integer msg = (Integer)comms.Recv(dr); // blocking receive
                        rts.Out(msg.toString()); // send to console
                }
        }
}
```

The program first creates a RTComms called *comms*, and a default group called *world*. They are both registered with the session (*rts*). Similar to MPI, we use group-based process identifiers to identify individual processes. In the example, process *0* sends an *Integer* object, while process *1* receives it. Both processes use blocking primitives. The send envelope defines a message tagged as *10*, to process *1* of group *world*. The receive envelope defines a message tagged as *10*, from process *0* of group *world*.

3.6.2 The RT Execution Mode

This program, when written for RT mode, requires two remote threads, following the *producer-consumer* pattern. The producer (i.e. boot) thread is called *SendReceiveBoot*. It creates a new remote thread called *SendReceiveEcho* on process *1* and sends a message to it. We will extend the program by using a dedicated communication port.

```
import visper.rt.*;
public class SendReceiveBoot implements RTRunnable {
	public void Run(String[] args, RTThreadGroup rttg,RTSession rts)
	{
		RTComms comms = new RTComms(rts,2000); // use dedicated port
		RTWorld world = new RTWorld(rts);
		RTRemoteProcess rtrp = new RTRemoteProcess(SendReceiveEcho.class,null,world,1);
		rts.GetComms().Barrier(world); // synchronize
		RTDataSend ds = new RTDataSend(10,1,world);
		comms.Send(ds,new Integer(5));
	}
}
public class SendReceiveEcho implements RTRunnable {
	public void Run(String[] args, RTThreadGroup rttg,RTSession rts)
	{
		RTComms comms = new RTComms(rts,2000); // use dedicated port
		RTWorld world = new RTWorld(rts);
		rts.GetComms().Barrier(world); // synchronize
		RTDataRecv dr = new RTDataRecv(10,0,world);
		Integer msg = (Integer)comms.Recv(dr);
		rts.Out(msg.toString());
	}
}
```

Since we do not use the default communication port, the system will not buffer the message. Therefore, due to the asynchronous nature of execution, we first synchronize at the default communication port (*rts.GetComms()*), before sending a message.

4. Performance

We have measured the basic performance of Visper using: 7 UltraSparc/Solaris2.5 with 256MB RAM, three Pentium200 with 64 MB RAM and two Pentium400 with 128 MB RAM/NT 4.00.1381, and two HP A9000-780/HP-UX B.10.20 with 512MB RAM connected with a 10 Mbps Ethernet. On the HP nodes we have used HP-UX Java C.01.15.05, on the Sun nodes Sun JDK1.1.6, and on the PCs Sun JDK1.2. Since the Solaris version does not support just-in-time compilation (JIT), the just-in-time compiler was disabled for the test, except were noted. We have also used MPICH 1.1.1 [25] and PVM 3.4 [29], built with Sun's C/C++ 4.2 compilers. The non-JIT results are included to compare Visper (Java) to MPICH (C) on the same hardware.

4.1 Speedup

The speedup test represents a simple Jacobi iteration with 2000 columns and 300 rows with a maximum of 100 iterations. The graph in Figure 8 displays the average execution time and speed-up over 8 runs for different number of nodes (from 1 to 10). For a 500 by 500 matrix, the best speedup value was only 2.7 and it peeked at 7 nodes. This is due to the fact that the computation time was not significantly greater than the communication time. It is interesting to compare these speedup values to the one obtained by a similar C program written for MPICH. For the 2000x300 example, the speedup was 2.11 on the 7

UltraSparc machines, and for the 500x500 example it was 2.27. For 1 node, the computation times were 15.162 seconds (2000x300) and 5.094 seconds (500x500). For 7 nodes, the 2000x300 example took 7.183 seconds to complete, and the 500x500 example took 2.242 seconds.

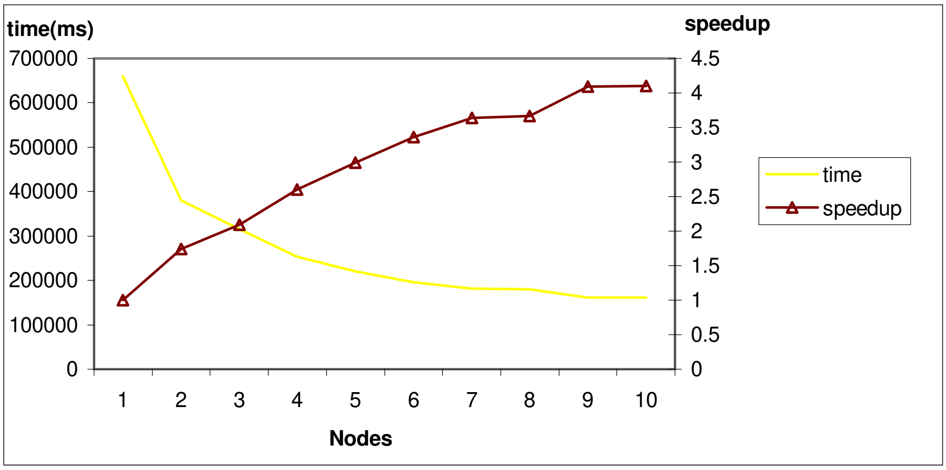

Figure 8 Time and Speedup (2000x300)

The graph in Figure 9 displays the average execution time and speed-up over 8 runs for a matrix multiplication problem with 1000 rows and 1000 columns. The *PC* values were collected only on the five PCs with JIT, therefore showing significantly better execution times. The *NOW* values were collected only on Unix. The speedup lines, however, are similar, as the problem is course grain.

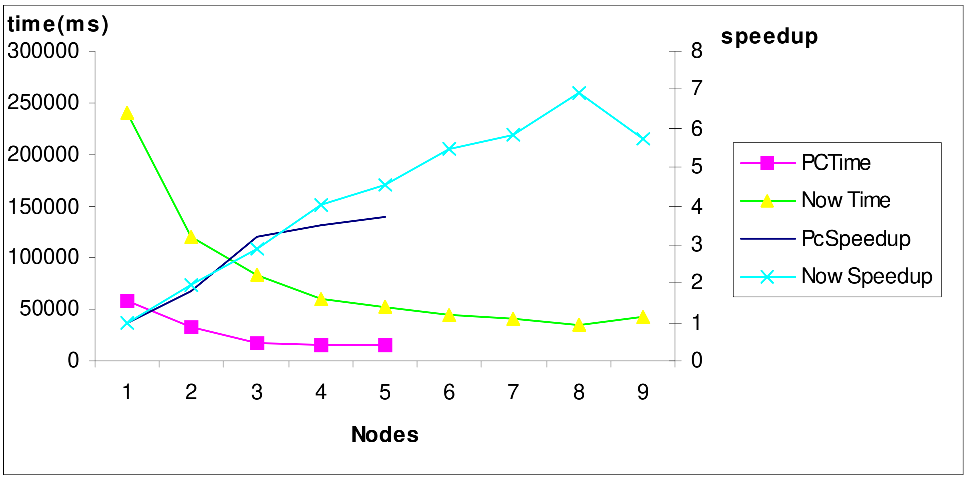

Figure 9 Time and Speedup (1000x1000)

4.2 Point-to-Point Communication

Generally speaking, to send a message, the following steps are required:

- Address resolution,
- Data marshaling, and
- Data Transfer.

With message-passing systems, the address resolution involves conversion of a logical process identifier with respect to a communication group into the actual IP address and port. The data marshaling involves the conversion of the data from the local host format into the network (i.e. architecture independent) format, and back. The data transfer takes place over a network, the characteristics of which usually have large impact on the overall performance of parallel programs. These three steps were included in the time cost benchmark, below.

Here we present the test results for the COMMS1 (i.e. Ping-Pong) [9] test performed on the mentioned libraries. The purpose of this benchmark is to measure the basic communication properties of a message-passing environment: latency and unidirectional bandwidth. The master process sends a message to the slave process, and the slave immediately returns it to the master. Messages are of a variable length, and it is assumed that the amount of time to send a message from the master to the slave is equal to the time required for a return. The tests were performed for messages of 1, 100, 1000, 10000, 100000, and 1000000 bytes in length. The latency and bandwidth values were calculated by a linear function via a least squares linear regression.

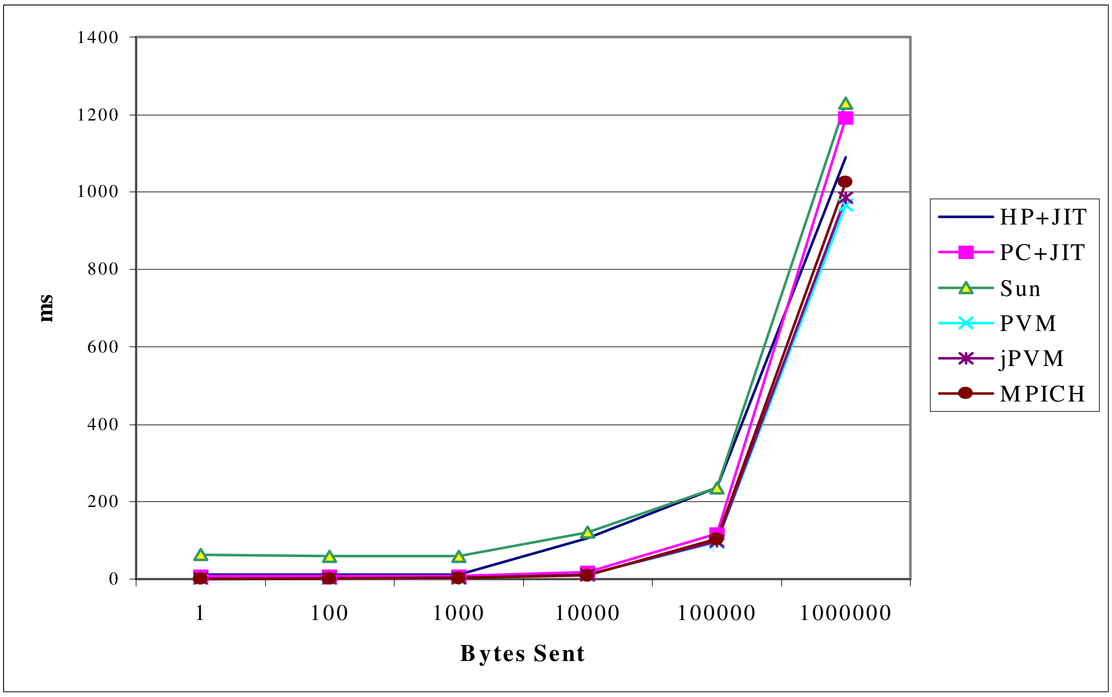

Figure 10 COMMS1 Times (ms)

As a reference, in Figure 10 (and Table 3 in Appendix) we present the COMMS1 results taken for 3 native systems: the PVM, the MPICH and the jPVM [43], collected on Sun. As expected, the PVM and MPICH times are very similar. The jPVM times are marginally inferior to the PVM times, due to the two causes of overhead: an additional call to switch between Java and C, and the cost of the Java to native data types conversion. Figure 10 also shows the RTComms times (*HP+JIT*, *PC+JIT*, and *Sun* curves) for the three target platforms. The tests were performed between 2 computers of the same type. The HP and PC results were obtained by using JIT. It is interesting to notice that although RTComms was built directly on top of the Java Object Serialization mechanism [39], so the impact of the introduced bytecode is minimal, it is two orders of magnitude behind jPVM for small messages. This is due to a high socket startup and serialization cost, and threads scheduling in Java. As messages were getting larger, the impact of Java on the overall performance has decreased. The slower node dominated the cross-architecture results. For example, the Sun-to-PC test yielded results similar to the Sun curve in Figure 10, while the PC-to-HP test produced results slightly better than the HP curve.

Table 1 Communication Cost (ms)

	jPVM	MPICH	Sun	HP+JIT	PC+JIT
Latency	0.784	0.905	81.554	51.249	5.782
Time/Byte	0.00098	0.00097	0.0016	0.0013	0.0012

Table 1 shows the communication costs for jPVM and MPICH on Sun, and RTComms on the three target architectures. The RTComms values reflect the high cost of networking in Java for small messages. jPVM and MPICH have outperformed Java on Sun by two orders of magnitude, but it is interesting to notice that JIT has significantly reduced the PC latency, while on HP the impact was only marginal. Good time-per-byte values can be explained by the fact that as messages were getting larger, the differences in times were getting smaller, for all the three JVMs.

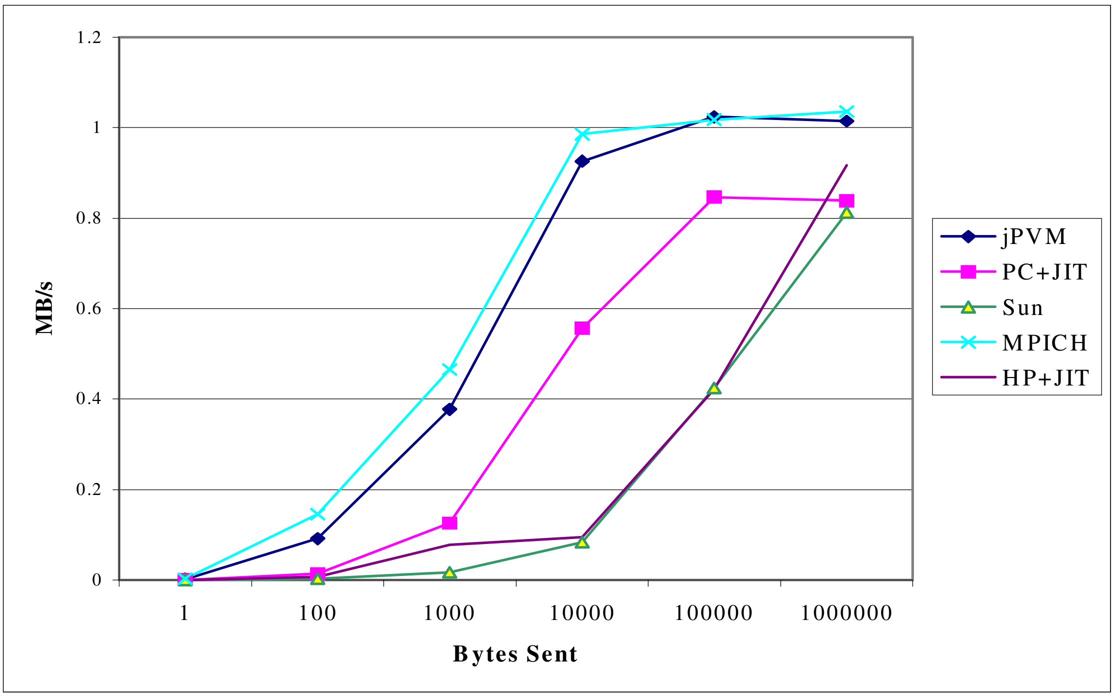

Figure 11 Bandwidth (MB/s)

As a result of the rather poor latencies, the effective bandwidth is reduced for small messages in RTComms compared to jPVM and MPICH by one to two orders of magnitude (Figure 11 and Table 4 in Appendix). Again, the HP results are rather surprising regarding the fact that all the tests were performed with JIT compilation. On the other hand, our implementation and tests have proved that a pure Java message-passing library, though inferior in performance to the systems based on native code, is a viable option. A more detailed presentation and discussion can be found in [33].

5. Related Java Systems

The WWW has the potential to be the infrastructure in integrating heterogeneous and remote computer systems into a global computing resource. There is a growing body of work on how to utilize best this new technology. Metacomputing systems, even on a local area network, are complex software and hardware structures and their development is a difficult, but challenging task. The issues that have to be addressed comprise scalability, heterogeneity, security, resource management, fault-tolerance, multi-language support, extensibility, interoperability and ease of use. While the systems mentioned in Section 1.1 address some if not all of these issues, the research work in Java has been progressing in three directions. One direction, represented by the so-called Java-enabled systems, has taken advantage of the Java Native Interface to enable execution of Java programs on top of a native system. This way, a parallel Java program takes advantage of the entire underlying infrastructure, while utilizing the platform independence and portability of Java. For example, NexusJava [13] provides Java bindings for Nexus, for creating and exchanging global pointers and performing remote service requests to methods defined in objects referenced by these global pointers. The binding also allows interoperability of Java and Nexus programs written in other languages. The efforts to enable message-passing libraries like the MPI and PVM to run under the JVM include jPVM [43] and MPI Java [6]. Similar to NexusJava, they both provide wrappers for the C-calls. Because Java types do not map easily to native types (e.g. C), the Java-MPI API [3] has been defined in an effort to standardize the interoperability. The approach of Netsolve [5] is somewhat different, but rather than implementing a new message-passing system, it opens up the existing resources to public. Netsolve implements a Java based system for exporting computational resources and libraries of preconfigured modules to solve computationally intensive problems. The user communicates with the system via a GUI that provides a set of available modules, an input window to specify the input data and an output window to view the results. Upon receiving the problem name and input data, the Netsolve system automatically allocates resources to carry out the computation and sends back a result.

The second direction is represented by the early pure Java systems taking advantage of Java applets running in WWW browsers to achieve distribution and parallelism. For example, KnittingFactory [2] is an infrastructure to facilitate building collaborative and parallel computations on the WWW. Load balancing and fault masking are provided by the runtime system transparent to the programmer. Due to the Java security policy of *host-of-origin*, direct applet-to-applet communication uses Remote Method Invocation [40]. In Javelin [7] there are three components: brokers, clients and hosts. A client is a process that seeks computing resources to solve a task. A host is a process offering computing resources. A broker is an HTTP server that coordinates the supply and demand for computing resources, based on the registrations of intentions by hosts and tasks by clients. All communication must be routed through the server, which makes a single server a bottleneck. This model is suitable for solving large, decomposable

problems such as the RSA factoring [30]. The main drive behind these projects was the recognition of the *write once run many* capability of Java.
The third direction is represented by the Java systems that do not rely on the WWW browsers and Java applets, but rather implement standalone parallel environments. Such systems are built either by extending the language with new keywords [27], or by providing a pure Java API for parallel processing. While these systems are relatively new, they are better described as prototypes or ongoing work than complete when compared to the native systems of Section 1.1. We concentrate here only on pure Java systems among which we classify Visper, since new keywords require nonstandard Java components. JPVM [11] is a PVM library written in Java, but misses some important PVM features, e.g.: dynamic process group, group broadcast and barrier synchronization. JPVM utilizes Java mainly for heterogeneity, since it lacks the ability to download application code from network, and fault tolerance is weak. Consequently, its applicability is limited to local networks. IceT [17, 21] is also a PVM like environment for parallel processing, but enhanced with classes for collaborative work where multiple users can work together. DOGMA [8] is a metacomputing system designed for running parallel programs on networks and supercomputers (IBM SP/2) based on the MPI model. It provides a communication library, the MPIJ that implements MPI completely in Java. Unlike the RTComms in Visper, the MPIJ implementation of MPI is based on the MPI C++ bindings as much as possible, and the programming style in DOGMA resembles closely the one found in MPI. There is no support for passing objects, since the authors have concentrated on native types and efficiency. Ninflet [42] is a Java based global computing environment that builds on the experience acquired by the Ninf system. Similar to Javelin, Ninflet is a three-tier architecture that comprises the dispatcher, the server and the ninflet. A ninflet is a schedulable client program that executes on the Ninflet system. Ninflets interact by invoking methods based on the RMI. None of the systems has referenced the Pasadena Workshop recommendations [26].

Table 2 Comparison

Feature	DOGMA	JPVM	IceT	Ninflet	Visper
Scalability	N	N		Y	Y
Scheduling	Y	N	N	Y	Y
Load Balancing	N	N	N	Y	N
Fault-Tolerance	N	N	N	Y	Y
Extensibility	N	N	N	N	Y
Interoperability	N	N	Y		N
Migration	N	N	Y	Y	Y
Data Scheme	Y	N	N	N	N
Clusters	Y	N	N	Y	N

Table 2 summarizes the more important features implemented so far. It is based on the list of runtime goals and programming interface that DOGMA seeks to meet. Ninflet is included primarily as a fairly complete parallel processing system, as it lacks message-passing primitives. Scalability in DOGMA and JPVM is questionable, since both systems use persistent communication channels to improve performance. While the performance results of MPIJ justifies the approach, in JPVM they depend very much on thread scheduling [33]. In pure Java, load balancing is an open issue due to a rather limited interface to the OS and, depending on the approach, may require some native code. Interoperability requires resolving two main problems. One is the initialization of the native PVM or MPI virtual machine from the Java system and vice-versa, and the other is in the data representation. As a proof of concept, IceT managed to soft-install C-based MPI processes on remote environments and dynamically install FORTRAN-base PVM. The support for clusters in DOGMA is part of its hierarchical system topology. In Visper, the network is a flat resource that is organized into sessions. However, all the nodes can be manually ordered in a group dialog (Figure 2) if the user wants to group them in a particular order. In all the systems security, if addressed, is based on the *java.lang.SecurityManager* class. While all the mentioned systems have been tied to the ideas and the style of the native parallel processing that they follow, Visper is perceived primarily as an environment that allows different programming styles, models and applications to be exercised within the same extensible framework. Therefore the support for parallel processing in Visper is just a service, not the ultimate goal.

6. Conclusion

Visper is a novel, pure Java based tool for SPMD parallel programming. It combines the novel features provided by Java with the standard practices and techniques pioneered by systems like the MPI and PVM. It allows remote execution of Java bytecodes, by transforming a network of machines into a virtual parallel computer. A parallel program executes as a group or groups of asynchronous remote threads that communicate by sending messages. In the shared variable paradigm, processes communicate by writing to and reading from shareable memory locations. This paradigm although simple violates, however, the principles of abstraction and encapsulation, making it difficult to implement large systems reliably [1]. The problems with such implementations include the mutual exclusion, the condition synchronization and the lack of control over communication modes for better performance. On the other hand, remote threads are autonomous, interacting computing elements that encapsulate data and procedure. They can be dynamically created and configured, thus providing flexibility in organizing their activity. Visper is designed as a three-layer system, where system services and system configuration are cleanly decoupled from the message-passing API. It provides secure, object-oriented, peer-to-peer message-passing environment in which programmers can compose, run and test parallel programs within persistent sessions. The communication primitives represented by the RTComms class follow the primitives of the MPI standard. They provide synchronous and asynchronous modes of communication between processes. The programmer can choose between point-to-point and collective communications. Visper supports synchronous and asynchronous modes of execution and primitives to control processes and groups of processes as if they were real parallel computers. The performed tests have showed that Java is a viable option for parallel processing, especially when enhanced at runtime by a JIT compiler. Visper will be made available from the following site: http://www.comp.mq.edu.au/~nstankov.

References

[1] Agha, G. A., and Jamali, N. Concurrent programming for Distributed Artificial Intelligence. In Weiss, G. (ed.). Multiagent Systems: A Modern Approach to DAI. MIT Press, 1998.

[2] Baratloo, A., Karaul, M., Karl, H., and Kedem, Z. M. KnittingFactory: An Infrastructure for Distributed Web Applications. TR1997-748, Department of Computer Science, New York University, New York, NY, 1997.

[3] Carpenter, B., Getov, V., Judd, G., Skjellum, T., Fox, G. MPI for Java: Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998. http://www.javagrande.org/

[4] Carpenter, B., Fox, G., Ko, S. H., and Lim, S. Object serialization for Marshaling Data in Java Interface to MPI. ACM 1999 Java Grande Conference, San Francisco, CA, June 12-14, 1999. http://www.cs.ucsb.edu/conferences/java99

[5] Casanova, H., and Dongarra, J. J. Netsolve: A Network Solver for Solving Computational Science Problems. Technical Report CS-95-313, University of Tennessee, Knoxville, TN, November 1995.

[6] Chang, Y-J., and Carpenter, B. MPI Java Wrapper Download Page, March 27, 1997. http://www.npac.syr.edu/users/yjchang/javaMPI

[7] Christiansen, B. O., Cappello, P., Ionescu, M. F., Neary, M. O., Schauser, K. E., and Wu, D. Javelin: Internet-Based Parallel Computing Using Java. Concurrency: Practice and Experience, Vol. 9, No. 11, November 1997, pp.1139 - 1160.

[8] DOGMA: Distributed Object Group Metacomputing Architecure. September, 1998. http://zodiac.cs.byu.edu/DOGMA

[9] Dongarra, J. J., Meuer, H-W., and Strohmaier, E. The 1994 TOP500 Report. http://www.top500.org/

[10] Elnozahy, E. N., Johnson, D., and Zwaenepoel, W. The Performance of Consistent Checkpointing. In Proceedings of the 11th Symposium on Reliable Distributed Systems, IEEE Computer Society, Houston, TX, October 1992, pp.39-47.

[11] Ferrari, A. JPVM. http://www.cs.virginia.edu/~ajf2j/jpvm.html

[12] Foster, I., Kesselman, C., and Tuecke, S. The Nexus Approach to Integrating Multithreading and Communication. Journal of Parallel and Distributed Computing. Vol. 37, 1996, pp.70-82.

[13] Foster, I., and Tuecke, S. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5), Syracuse, NY, August 1996, pp.112-119.

[14] Foster, I., Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. The International Journal of Supercomputer Applications and High Performance Computing. Vol. 11, No. 2, 1997, pp.115-128.

[15] Geist, G. A., et al. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[16] Gosling, J., Joy, B., and Steele, G. The Java Language Specification. Addison Wesley, 1996.

[17] Gray, P., and Sunderam, V. The IceT Project: An Environment for Cooperative Distributed Computing. http://www.mathcs.emory.edu/~gray/IceT.ps

[18] Grimshaw, A. S., and Wulf, W. A. The Legion Vision of a Worldwide Virtual Computer. CACM, Vol. 40, No. 1, January 97, pp.39-45.
[19] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
[20] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. Communications of the ACM. Vol. 17, No. 10, 1978, pp.549-557.
[21] Java Grande Working Group on Concurrency/Applications. Status: November 1998. Notes of the 1st International Workshop on Desktop Access to Remote Resources. http://www.javagrande.org/
[22] Kranzlmüller, D., Stankovic, N., and Volkert, J. Debugging Parallel Programs with Visual Patterns. In Proceedings of VL'99 - the 15th IEEE International Symposium On Visual Languages, Tokyo, Japan, September 1999, pp.180-181.
[23] Lindholm, T., and Yellin, F. The Java Virtual Machine Specification. Addison Wesley Developers Press, Sunsoft Java Series, 1996.
[24] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. University of Tennessee, Knoxville, TN, July 18, 1997.
[25] MPICH-A Portable Implementation of MPI.
http://www-unix.mcs.anl.gov/mpi/mpich
[26] Pasadena Working Group #7. DRAFT: Message-Passing and Object-Oriented Programming. In Proceedings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments, 1995.
http://cesdis.gsfc.nasa.gov/
[27] Philippsen, M., and Zenger, M. JavaParty - Transparent Remote Objects in Java. Concurrency: Practice and Experience, November 1997, pp.1225-1242.
[28] Philippsen, M., and Haumacher, B. More Efficient Object Serialization. In Parallel and Distributed Processing, International Workshop on Java for Parallel and Distributed Computing, LNCS Vol. 1586, San Juan, Puerto Rico, April 12, 1999, pp.718-732.
[29] PVM: Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html
[30] RSA Factoring-By-Web project. http://www.npac.syr.edu/factoring.html
[31] SoftWired AG. Programmer's Manual. Version 0.5. August, 20, 1998.
http://www.softwired.ch/ibus.htm
[32] Stankovic, N., and Zhang, K. Visual Programming for Message-Passing Systems. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol.9, No.4, August 1999, pp.397-423.
[33] Stankovic, N., and Zhang, K. COMMS1 Benchmark and Java. Technical Report C/TR99-05, Macquarie University, Sydney, NSW, Australia, June 1999.
[34] Stankovic, N., and Zhang, K. A Parallel Programming Environment for Networks. In Polychronopoulos, C., Joe, K., Fukuda, A., and Tomita, S. (eds.). High Performance Computing. Proceedings of ISHPC'99 - the 2nd International Symposium, LNCS Vol. 1615, Kyoto, Japan, May 1999, pp.381-390.
[35] Sun Microsystems, Inc. Java Core Reflection, API and Specification. February 4, 1997.
[36] Sun Microsystems, Inc. Java HotSpot Performance Engine. http://java.sun.com/
[37] Sun Microsystems, Inc. Java IDL. http://java.sun.com/products/jdk/1.3/docs
[38] Sun Microsystems, Inc. Java Native Interface. http://java.sun.com/docs/index.html

[39] Sun Microsystems, Inc. Java Object Serialization Specification. Revision 1.4, July 3, 1997. http://java.sun.com
[40] Sun Microsystems, Inc. Java Remote Method Invocation Specification. Revision 1.42, October 1997.
[41] Sun Microsystems, Inc. Java Security Architecture (JDK 1.2). Revision 0.7, October 1, 1997. http://java.sun.com
[42] Takagi, H., Matsuoka, S., Nakada, H., Sekiguchi, S., Satoh, M., and Nagashima, U. Ninflet: a Migratable Parallel Objects Framework using Java. ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, CA, pp.151-159.
[43] Thurman, D. JavaPVM. http://homer.isye.gatech.edu/chmsr/JavaPVM

Appendix

Table 3 presents numerically the values drawn out in Figure 10.

Table 3 COMMS1 Times (ms)

Bytes Sent	HP+JIT	PC+JIT	Sun	PVM	jPVM	MPICH
1	12	8	63	0.563	0.95	0.603
100	13	8	60	0.766	1.1	0.709
1000	13	8	60	2.363	2.65	2.203
10000	106	18	121	10.41	10.8	10.21
100000	237	118	236	95.84	97.6	103.1
1000000	1089	1192	1230	966.3	985.05	1026

Table 4 presents numerically the values drawn out in Figure 11.

Table 4 Bandwidth (MB/s)

Bytes Sent	jPVM	MPICH	Sun	HP+JIT	PC+JIT
1	0.001	0.002	1.60E-05	8.30E-05	1.30E-04
100	0.091	0.145	0.002	0.007	0.013
1000	0.377	0.465	0.017	0.077	0.125
10000	0.926	0.986	0.083	0.094	0.556
100000	1.024	1.018	0.424	0.422	0.847
1000000	1.015	1.036	0.813	0.918	0.839

NENAD STANKOVIC received a B.S. in electrical engineering from the University of Zagreb, Croatia in 1983 and a M.S. degree in computer science from Macquarie University, Australia in 1997. Currently he is employed with FSJ Inc., and also working towards his Ph.D. in computer science at Macquarie University. Prior to coming to Macquarie University he worked on different projects in computing. His research interest is in the areas of parallel programming environment, parallel and distributed system and architecture design and artificial intelligence.