Design and Implementation of Parallel Java with Global Object Space*

L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth
Parallel Programming Laboratory
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL, U.S.A.

Abstract *Java[1] has emerged as a dominant language that could eventually replace C++. It is believed that using Java would boost programmer productivity because of its well-thought design, independence from backward-compatibility with C, absence of arbitrary pointers, etc. We present the design and implementation of a parallel extension to Java. The parallel extension provides dynamic creation of remote objects with load balancing, and object groups. The language constructs are based on those of Charm++[1]. The parallel Java extension is implemented using the Converse[2] interoperability framework, which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application.*

Keywords: parallel, message-driven programming, object-oriented, Java

1 Introduction

Java, as a programming language, has seen phenomenal growth in the past few years. The initial appeal of Java was related to its connection to the Web. However, its benefits as a good programming language are also being recognized. Its clearly-designed object-oriented structure uncluttered by the backward compatibility requirements (such as those for C++), makes it a language that can potentially enhance programmer productivity substantially. As the compiler technology catches up, with the just-in-time compilers as well as direct stand-alone compilers, the speed differential between C++ and Java will narrow down. Deployment of java for the purpose of large-scale application development is therefore a distinct possibility in the near future.

Clusters of workstations within the intranet, as well as parallel servers are increasingly being used as part of the computational infrastructure. To utilize the computing power of such platforms for applications of the future, it is desirable to extend Java with parallel constructs. This paper presents the design and implementation of such an extension.

The parallel Java project is a part of our broader effort towards *multilingual* parallel programming. This effort is based on the Converse interoperability framework. It emanates from our belief that no single parallel programming language or paradigm is adequate to support all the complex parallel applications, or even diverse subcomponents of a large parallel application. Converse allows individual parallel modules to be written using different paradigms, and in different languages. Converse also facilitates implementation of the runtimes of new parallel languages by providing portability and support for commonly needed runtime facilities. As our parallel Java extension is implemented using Converse, it allows incremental adoption of parallel Java: existing libraries written in C-MPI,

*This research is supported in part by National Science Foundation Grant BIR 93-18159.

[1] Java is a registered trademark of Sun Microsystems, Inc.

Charm++, PVM, etc. can be utilized in a new application, with new modules written in Java.

Our implementation does not require any modifications to the Java compiler or to the Java virtual machine. Any standard compiler and JVM can be used. User code is written in standard Java, with calls to our runtime library. The user must also write small interface files (CORBA-IDL style, but much simpler), translated by an interface translator provided with the system.

In the next section, we describe the parallel extensions from the user's point of view and illustrate with a few examples. The Converse system used in the implementation is described next in Section 3, along with a description of Java facilities used in our implementation and a discussion of the actual implementation and issues of interoperability. This is a preliminary implementation of parallel Java, which we plan to enhance in the future. Potential enhancements are discussed in section 4 followed by our conclusions.

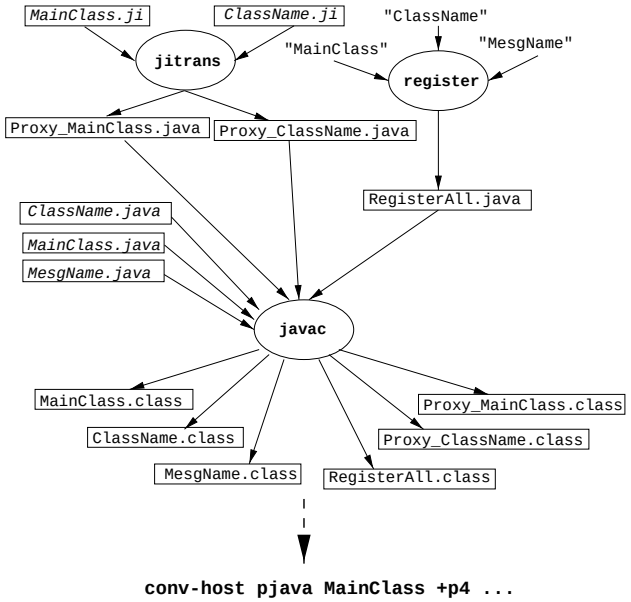

Figure 1: Structure of a simple Parallel Java application: Programmer codes main and remote classes, provides interface files (`*.ji`) for each of them, generates registration and proxy classes, and then compiles all Java files.

2 Design of Parallel Java

We have extended Java with two new constructs based on the Charm++ model. *Remote objects*, similar to *chares* in Charm++, are objects that can be created on remote processors, and are accessed through proxy objects. *Object groups*, like *branch-office chares* in Charm++, have an instantiation of an object, i.e. a branch, on every processor. Before we describe these extensions, we provide an overview of the structure of a simple Parallel Java application.

2.1 Application Structure Overview

There are various components to a Parallel Java application. The programmer must designate a main class, an instance of which is automatically created on processor 0. Arguments (`String argv[]`) are passed to the main object's static method `main`. After `main` exits, the system invokes the scheduler and waits for messages in the scheduler loop. Other processors go into the scheduler loop right from the beginning.

Remote classes are defined similarly to regular Java classes, with a few exceptions. All remote objects are created and accessed through the use of *proxies*. A proxy class implements a representation of a remote object providing access to the remote object through similar *entry methods*. Entry methods are methods that can be invoked remotely. Proxy classes are generated with the `jitrans` utility, as shown in Figure 1, which uses interface files provided by the programmer to generate proxy class definitions.

Message classes are programmer-defined classes that must be defined for every type of message that a constructor or entry method can receive. Messages are serializable objects that use Java's serializable interface to convert themselves to and from byte arrays. This serialization and deserialization is taken care of by the runtime system.

Names of the main, remote and message classes are passed through a `register` utility that creates `RegisterAll.java`. All the parallel classes, messages and entry methods (including constructors for parallel objects) have to be registered with the runtime system, so that it knows which methods to invoke, how to unpack a message and which object a method is intended for.

The complete structure of files making up a basic parallel application is shown in Figure 1.

2.2 Remote Objects

A class definition file for remote objects should import the `parallel` package. A remote class is a class that can be instantiated remotely. It has the same structure as a Java class except for some syntactic restrictions on methods that can be invoked remotely. These methods, called *entry methods*, are also declared in the interface files. All entry methods of a remote class are asynchronously invoked, should take a single parameter of a user-defined message class and should not return any value. Constructors may optionally have a RemoteObjectHandle as a first parameter. This handle is a global reference to the object itself. Thus if a remote object needs to send its own reference to other objects in messages, it has to store this handle locally (see Figure 2). Remote objects are accessed through proxies, and references to remote objects can be transmitted in messages using handles.

Remote objects are declared by creating an instance of a proxy object. For example, to create a remote object of class `ClassName`, declare

```
Proxy_ClassName myobj;
```

To actually create an object remotely, we must send a new object message that calls a constructor for that class:

```
myobj = new Proxy_ClassName(pe, msg);
```

Here, `pe` is the processor on which the object is to be created. If we leave out `pe`, a processor will be chosen by the dynamic load balancer. Finally, to send a method invocation message to a remote object, we use the proxy as follows:

```
myobj.MethodName(msg);
```

This call is asynchronous; the caller does not wait for the call to complete.

To send a reference to a remote object in a message, we must obtain the object's handle. A proxy acts as a local representation of a remote object, while a handle is a global reference to the object. Consider the following example:

```
msg.anobj = myobj.globalRef;
someobj.SomeMethod(msg);
```

Here, a handle on `myobj` is placed in msg, and then transmitted to `SomeMethod` of `someobj`. To extract `myobj` on the receiving side, we would do the following:

```
Proxy_ClassName myobj2 =
  new Proxy_ClassName(msg.anobj);
```

This creates a new proxy that controls access to a pre-existing remote object.

Interface Files: The programmer defining a remote class must also provide an interface file for that class. An interface file for a remote class consists of a listing of all the entry points and message types for that class with the following syntax:

```
class ClassName {
  entry Constructor1(MessageType0);
  entry EntryMethod1(MessageType1);
    ⋮
  entry EntryMethodN(MessageTypeN);
}
```

Note that non-entry-methods need not be listed in the interface file.

Figure 2 shows a sample program implementing a remote object to illustrate the creation and method invocation concepts described above.

2.3 Object Groups

Object groups are defined in the same way as remote objects, and created much the same way as well. Since an instance of the object is created on all processors, no processor number is specified to the proxy constructor.

```
Proxy_GroupClassName mygroup =
  new Proxy_GroupClassName(msg);
```

To send a message to a single branch of the object, we use the syntax

```
mygroup.SomeMethod(pe, msg);
```

where pe is the number of the processor to send the message to. To send the message to all instances of the object, simply omit the pe parameter.

The interface file for an object group class is similar to that of the remote class, with the exception that the `class` keyword is replaced by `group`.

```
group ClassName {
  entry Constructor1(MessageType0);
  entry EntryMethod1(MessageType1);
    ⋮
  entry EntryMethodN(MessageTypeN);
}
```

Figure 3 shows a sample program implementing an object group. The object group created is a `Ring`, and so its creation effectively creates one ring node on each processor.

2.4 Running Parallel Java programs

Once all the Java files have been compiled using the Java Compiler, the Parallel Java application can be run on a network of workstations using the `conv-host` utility supplied with Converse and a Parallel Java virtual machine `pjava`:

```
conv-host pjava MainClass +p4 -classpath=...
```

`conv-host` reads the IP addresses of the individual workstations from a local file

```
public class Main{
  public static void main(String argv[]){
    Proxy_A a = new Proxy_A(1,new InitMsg());
    a.DoWork(new WorkMsg());
  }
  public void ExitApp(ExitMsg emsg){
    PRuntime.exit(0);
  }
}

public class A{
  private RemoteObjectHandle globalRef;
  private RemoteObjectHandle mainhandle;

  public A(RemoteObjectHandle h,
           InitMsg imsg){
    // do initializations
    globalRef = h;
    mainhandle = PRuntime.mainhandle;
  }
  public void DoWork(WorkMsg wmsg){
    // do work
    Proxy_Main m;
    m = new Proxy_Main(mainhandle);
    m.ExitApp(new ExitMsg());
  }
}
```

Figure 2: Example of RemoteObject Creation and Method Invocation

and spawns the Parallel Java virtual machines `pjava` on those workstations, passing `MainClass` and other parameters to it. `pjava` uses the Java Invocation API to load the Parallel Java Runtime class, and registers the remote objects using the `RegisterAll` class. It then invokes method `main` of class `MainClass` on processor 0, and invokes the scheduler.

3 Implementation

Our parallel implementation of Java is based on the Converse interoperability framework and several facilities provided in Java (JDK 1.1) itself. Before describing our implementation we first briefly review Converse and the relevant Java features.

3.1 Converse

Converse [2] is a runtime framework that supports multilingual parallel programming. It

```
public class Main {
  private static final int NITER = 100;

  public static void main(String argv[]) {
    Proxy_Ring ring =
        new Proxy_ring(new InitMsg(NITER));
    ring.StartRing(0, new StartMsg(0));
  }
}

public class Ring{
  // private instance variables
  ...
  public Ring(RemoteObjectHandle h,
              InitMsg imsg) {
    globalRef = h;
    niters = imsg.iterations;
    mype = PRuntime.MyPe();
    numpes = PRuntime.NumPes();
    ring = new Proxy_Ring(globalRef);
  }
  public void StartRing(StartMsg s) {
    if(mype==(numpes-1)) {
      if(s.iter == niters) {
        PRuntime.exit(0);
      } else {
        s.iter++;
        ring.StartRing(0,s);
      }
    } else {
      ring.StartRing((mype+1)%numpes, s);
    }
  }
}
```

Figure 3: A Ring Program in Parallel Java

provides an infrastructure for building new parallel programming languages. Languages implemented on top of Converse can interoperate with parallel modules written using other languages. Thus a parallel application can consist of multiple modules written using different languages built on top of Converse.

Converse makes this possible by abstracting the notion of computation assignment into objects called generalized messages and using a common scheduler for all generalized messages irrespective of the language runtime that generated them. A generalized message consists of two parts. The message header specifies the handler for that message, whereas the message body corresponds to the language specific information about the data enclosed in that message. The Converse scheduler waits for messages to arrive for either the local processor or remote processor, and executes the appropriate handler based on the message header.

The scheduler is exposed to the language implementor. Indeed, one can even substitute a different scheduler for the default one. Exposing the scheduler to the language runtime is necessary because different languages embody radically different control regimes. Languages such as MPI have a single thread of control. If the MPI module blocks for the message, they hold up the entire processor. Message-driven languages such as Charm++ invoke methods on objects upon arrival of messages and never block. Thus, an application consisting of concurrent modules written in Charm++ and MPI should not block the ready computations in Charm++ just because the MPI module is blocked on a specific tagged message. When the scheduler is exposed to the MPI developer, MPI can transfer the control to scheduler for performing other computations while it waits for the specific message.

3.2 The Java Language

Much of our implementation makes use of available Java [3] facilities. We discuss the Java Remote Method Invocation API and some of the drawbacks of its use. We also discuss features of the Java Native Interface, the Invocation API and the Reflection API, as they are used in our implementation of parallel Java.

3.2.1 Java Remote Method Invocation

The Java Remote Method Invocation API (RMI) was provided to enable writing distributed applications in Java. RMI architecture is based on a client-server model. Every remote object acts as a server and can offer multiple interfaces to clients. Clients of the remote objects invoke methods on the stub for that remote object. However, the clients never interact with the implementation of the remote object but rather with the remote interface that object offers. Although this provides more

security and flexibility for the remote object developer, it is inefficient and often not needed in a parallel application, where the remote objects are trusted. Use of TCP as the underlying transport layer in RMI restricts the number of hosts in a distributed program to a system specific maximum (such as 64). Though this is not a very serious shortcoming for distributed applications, parallel applications will encounter this limit because they may wish to use hundreds of processors. Java RMI uses dynamic class loading to create stubs to remote objects at runtime. This provides enormous flexibility for dynamic environments such as WWW, but is often inefficient and unnecessary in parallel programs, where the security checks associated with the classloader are not needed. Java RMI is synchronous. The calling thread waits for the result of the remote method execution. Implementing asynchronous RMI on top of synchronous RMI implies providing a callback method on the client object. Therefore, one has to implement the client object also as a remote object and pay the unnecessary overhead of the return message. All the remotely accessible objects in RMI have to extend UnicastRemoteObject. Since Java does not permit multiple inheritance, one cannot use classes derived from any other class as classes for remote objects.

For these reasons, we chose not to make use of the Java RMI facility. Instead, we opted to use the communication facilities of Converse.

3.2.2 Java Native Interface

Java Native Interface(JNI) allows applications written in Java to interoperate with code written in other programming languages such as C and C++. JNI is typically used for utilizing certain platform-dependent features, for reusing legacy code, and for speeding up time-critical portions of applications. Java classes are augmented with "native" methods that are written in C or C++ and linked dynamically through DLL's on Windows platform or through shared objects on Unix. The Java Native Interface provides functions to create, access and modify Java objects, invoke Java methods, and to raise and catch exceptions. We use JNI to interface with the Converse messaging layer, and the scheduler.

3.2.3 Invocation API

Invocation API is a set of library functions that allow embedding the Java Virtual Machine in a native application. Such applications create an instance of Java VM, and attach the current thread to it. Once the virtual machine is initialized, it is instructed to carry out Java-related tasks such as loading classes, instantiating them and invoking methods on the objects. Our parallel Java implementation uses the Invocation API in the application wrapper program `pjava`. This wrapper program provides a gateway between the Converse Host program and the parallel Java application on every processor. Specifically, it loads the parallel runtime system on every processor, and passes control to the Converse scheduler.

3.2.4 Reflection API

The Java Reflection API supports introspection about loaded classes to applications. In particular, it allows applications to construct new classes and arrays, access and modify fields of classes and elements of arrays, and to invoke methods on objects. Object brokers for Java-based components rely on the Reflection API to discover information about objects and to customize objects at runtime. In addition, Java Reflection enables writing tools such as debuggers and profilers in Java portably. The parallel Java runtime system class makes use of the Reflection API to invoke methods on objects when corresponding messages arrive.

3.3 Implementation of Parallel Java

We were guided by the following principles in implementing Parallel Java:

- Use minimum native code in order to achieve portability. We use only six native methods in the parallel runtime class.

- Use existing Java facilities whenever possible. An example of this is the use of the serializable interface for messages.
- Avoid copying as much as possible. An important consequence of this decision was to use handles as a global reference instead of proxies. Also, if the message is intended for an object on the local processor, a reference is passed to a message instead of cloning it.

When a new remote object is created by calling the proxy's constructor method, it calls the `CreateRemoteObject` method in the parallel runtime class. This method allocates a *virtual handle*, a special handle indicating that the remote object that the handle refers to is not yet created. It then sends a message to the processor on which the remote object is to be created, specifying the class to be instantiated, the constructor method to be called and the message that is to be passed as an argument to the method. This *send* is asynchronous, so the runtime system does not wait for the object to be actually created before returning to the application. When the `CREATE` message reaches its destination processor, the remote processor creates an object, and calls its constructor method with the message argument. It then sends a message back to the source processor of that message indicating that the remote object has been created. Upon receipt of this `CREATED` message, the original proxy reference gets updated to point to the actual object, and no longer remains virtual. If in the meanwhile, i.e. after creation of proxy and before creation of the remote object, some object tries to invoke a method on the remote object, the call is queued inside the runtime system of the processor which created the first proxy.

For method invocation, the wrapper method in the proxy invokes the `InvokeMethod` method of the runtime class, passing it the reference to the remote object, the ID for the method (created by `RegisterAll` to be invoked and the message argument. If the handle is not virtual, i.e. the remote object has already been created, then the runtime system sends an `INVOKE` message to the home processor of the remote object. If a virtual handle is specified, then the runtime system forwards this message to the creator of the first proxy of that object, which maintains a queue of methods to be invoked on the remote object once it is created.

A message object is serialized into a byte array only if the method is to be invoked on an object residing in a remote Java virtual machine. Otherwise, its reference is included in a machine-level message passed to the destination object. This means that once the message has been passed as a parameter to a remote method, its ownership should be relinquished by the caller. Modification to the message object after it has been sent can result in unspecified behavior and can result in the most obscure timing-dependent bugs.

The registration mechanism we use offers a simple and efficient mechanism for locating remotely instantiatable classes, methods, and messages. This is one place where, for the sake of efficiency, we have deviated from the facilities for dynamic class loading provided in Java and implemented our own static registration of all the entities involved in remote method invocation. We use the Java Reflection API to access information about classes, methods and messages when they are registered and store them in the runtime system for efficient access.

In order to exit the parallel application, one needs to call the `exit` method provided by the parallel runtime, and not the method provided by the `System` class. `PRuntime.exit` sends a message to processor 0 declaring the caller's intention to exit. Processor 0 then broadcasts this message to all the processors and waits for the replies. When other processors receive this exit message, they acknowledge to processor 0 and exit the virtual machine. Processor 0 exits after it receives all the acknowledgements.

Since the remote processors typically do not have direct access to the standard input and output of the host (the machine from which the application is started), we use the facilities provided by Converse to print to `stdout` and `stderr` of host and to read from `stdin` of host. We have embedded this facility inside the `out`,

`err` and `in` objects inside the runtime class, so that methods such as `println` are supported.

Currently, the Java threads are not mapped to the native threads in Sun's Java implementation. This has an unpleasant side effect that thread context switches cannot occur once the JVM is inside a native method. Since we need to invoke the native Converse scheduler for processing Parallel Java messages, other threads cannot run even if there are no messages to be processed. However, this will go away with the availability of native threaded version of the Java Implementation. An alternative solution is to run the scheduler in a polling mode as a Java thread.

3.3.1 Interoperability

`pjava` is essentially a C program that uses Converse libraries and embeds the Java Virtual Machine using the Invocation API. It initializes Converse using `ConverseInit`, starts JVM, loads `PRuntime` from a well-known classpath, and starts the Converse scheduler. The facility to embed a Java virtual machine inside any native application also makes it possible for other parallel languages implemented on top of Converse to interoperate with modules written in Parallel Java. For example, suppose a Charm++ application needs to invoke some computation in a Parallel Java module, it can invoke the JVM and take the steps taken by `pjava` to start computation in Parallel Java. Similarly, classes developed in Parallel Java can include native methods that trigger Charm++ computations. The Converse scheduler will automatically interleave the execution of Parallel Java objects, Charm++ objects, and other entities in a non-preemptive fashion.

3.4 Current Status

Parallel Java will run on any machine where there exists both Converse and Java installations. Since ports for Converse exist to most platforms (but not yet on Windows), availability largely depends on that of Java. We use several features supported in JDK 1.1. At the time of this implementation they were available only on the SUN platforms running Solaris which is where our implementation was tested.

Preliminary performance results indicate that remote method invocation in our Parallel Java implementation is substantially faster than the Java RMI facility. More extensive performance analysis is planned for the future.

4 Future Work

In the future, our approach to further development of the parallel Java system will be along two lines: first, making parallel Java feature-rich by including features supported by other message-driven languages such as Charm++. These include more extensive load-balancing support, object arrays, specifically shared variables, and message prioritization. Second, we wish to take advantage of the dynamic nature of Java to enable the construction of customizable parallel components.

Currently our parallel Java implementation uses a quasi-dynamic random load-balancing strategy. If the processor number is not specified for the remote object at creation time, the runtime system creates it on a randomly generated processor. One can provide a load-balancing strategy for object-creation in the parallel Java runtime. This approach has some drawbacks in the case of applications using modules written in multiple languages. Using this approach, every individual module may be load-balanced, but the application can still exhibit severe load-imbalance. In order to provide a common load-balancing strategy across multiple modules, Converse provides support for languages that use dynamic load-balancing. In the future, we would use this Converse facility to create new objects.

In future versions of Parallel Java, we plan to implement object arrays, new information sharing abstractions, and message prioritization. Object groups (Branch-Office chares) are a restricted form of the more general *object array* abstraction. This abstraction supports

multidimensional arrays of remote objects. These arrays are range-addressable, support static mapping and object migration. Object arrays have been implemented in Charm++ [4] and have been found useful in several scientific applications.

In our current implementation, the only mode of inter-object communication is through remote method invocation. However for many programming tasks, better mechanisms of sharing information between objects can be used to make the structure of the program simple to understand and modify. In Charm++, we have already shown the utility of such abstractions called specifically shared variables[5].

Message prioritization provides an elegant way to optimize a parallel message-driven program[6]. This optimization is achieved by associating priorities with computations specified by messages in order to speed up processing of tasks on the critical path of the program.

One important feature of our design is that the remotely instantiatable classes need not be derived from a specific class such as `UnicastRemoteObject` as in Java RMI. This allows the users to create their own hierarchies of parallel objects. This also allows them to remotely instantiate arbitrary classes originally written in a sequential context. However, in order to achieve this, we need to remove the restriction that the constructors and entry-methods of these classes can take exactly one argument, i.e. the message. We plan to remove this restriction using automatic parameter marshalling. The interface translator will be extended to include the automatic packing and the runtime will be extended to do unpacking of the arguments.

A long-term research goal of our parallel Java work is to demonstrate the applicability of message-driven object-based programming to build customizable parallel components that can be reused for rapid parallel application development. The JavaBeans component architecture developed at Sun Microsystems already employs Java to build customizable sequential components. Recent developments in JavaBeans include a bridge with ActiveX technology from Microsoft and integration with CORBA. The concept of sequential components in JavaBeans can be extended to parallel components, thus providing a bridge to the fast method invocation mechanism of Converse. This capability will be demonstrated in NAMD, a molecular dynamics simulation program we have been developing at University of Illinois [7]. We will build customizable parallel components for various entities in the simulations. It will then be possible to tie these components together using a scripting language in a customized molecular dynamics application.

5 Conclusion

We described the design and implementation of a parallel Java extension. The parallel constructs distinguish between parallel, remotely invokable objects, and normal sequential objects. The extension supports the dynamic creation of remote objects and efficient asynchronous method invocation. Object groups, which are essential for efficient and modular programming for parallel applications, are also supported. The implementation uses several new features of Java, such as the invocation API and the reflection API. No change to the Java compiler or JVM is required. Instead an interface compiler is used to generate wrapper code for each parallel class. The user only has to specify the names of parallel classes, and the signatures of their remotely invokable methods to the interface translator.

Our parallel Java extension is implemented on top of the Converse interoperability framework. This simplified the implementation of the parallel runtime, and makes it automatically portable to a variety of machines supported by Converse, as long as Java is available on them. More importantly, using Converse also means that parallel Java modules can coexist with parallel modules in other languages, such as Charm++, MPI, or HPF. As a result, one can put together an application in Java relatively quickly, and reimplement indi-

vidual modules in more efficient languages such as Charm++ or reimplement the critical sequential components in C or Fortran. This can provide a boost in programmer productivity.

A few other approaches to parallel Java have been made recently. JavaParty [8] supports synchronous and asynchronous method invocation of remote objects. It requires modification to the Java compiler. Some other approaches, [9], [10] provide CSP like channels for communicating Java processes. Our approach differs from them primarily in our support for interoperability with other parallel languages, and in our parallel object primitives which we believe are a better match with both parallel programming requirements, and the Java programming model.

Several issues have been identified for future work, and significant effort must be devoted to them. However, we believe that this approach will eventually lead to a productive new tool for parallel and distributed programming.

References

[1] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[2] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[4] Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proc. Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[5] A. Sinha and L.V. Kalé. Information Sharing Mechanisms in Parallel Programs. In H.J. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 461–468, April 1994.

[6] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[7] L. V. Kale, Milind Bhandarkar, Robert Brunner, Neal Krawetz, James Phillips, and Aritomo Shinozaki. A case study in multilingual parallel programming. Technical report, Theoretical Biophysics Group, Beckman Institute, University of Illinois, Urbana, June 1997.

[8] Michael Philippsen and Matthias Zenger. JavaParty–Transparent Remote Objects in Java. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation (To Appear)*, Las Vegas, Nevada, June 1997.

[9] Erik D. Demaine. Higher-Order Concurrency in Java. In *Proceedings of the Parallel Programming and Java Conference (WoTUG-20)*, pages 34–47, April 1997.

[10] Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In *WoTUG-20: Parallel Programming and Java Conference (To Appear)*, April 1997.