Jade: A Parallel Message-Driven Java

Jayant DeSouza and Laxmikant V. Kalé

University of Illinois, Urbana IL 61801, USA
jdesouza@uiuc.edu, kale@cs.uiuc.edu

Abstract. We present Jade, a Java-like[1] language with parallel message-driven features. The parallel constructs include parallel classes called *Chares* and parallel arrays of objects called *ChareArrays*. Communication with a parallel object occurs through asynchronous method invocation. In the message-driven paradigm, when a method of a Chare or ChareArray is invoked, it continues to completion before any other method of the same parallel object can run. In contrast to Java's run-time compilation, the Jade source code is translated to Charm++ source code, which is then *compiled* and executed on the target machine. The resulting code supports object migration and load-balancing and scales well to large number of processors. Java's standard libraries are not supported.

1 Introduction

A promising approach to parallel programming that we have been exploring for the past decade is to effect an "optimal" division of labor between the application programmer and the "programming system". Specifically, this approach requires the programmer to decompose their application into a large number of interacting objects, and passes to the runtime system the responsibility of mapping the objects to processors. The "objects" may be true C++ objects, or they may be user-level threads running MPI[3]. In either case, the model requires encapsulation: each object is allowed to access its variables, along with read-only data structures.

This approach leads to several benefits: powerful scalability, efficient execution on both shared and distributed architectures, a simple programmer cost model for the various features of the language, object migration and automatic dynamic measurement-based load balancing [8, 4], automatic checkpointing, support for out-of-core execution, the ability to dynamically shrink or expand the set of processors used by the job.[12]

We have implemented this approach in C++, in the form of the Charm++ system.[6, 9, 11] Charm++ has been very successful in parallelizing several applications effectively. As an example, NAMD[7], a molecular dynamics application written in Charm++ is very scalable and runs on thousands of processors.[21] Recently, NAMD won the Gordon Bell award at SC2002.[22] Another example

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

is the FEM framework[2] written in Charm++ and AMPI, which is being used to develop very large applications at the Center for the Simulation of Advanced Rockets.

The work reported here describes our preliminary attempts at bringing the benefits of the Charm++ approach to Java. Some of the benefits of combining Java's serial programming model with the parallel model of Charm++ are:

- The absence of pointers and pointer arithmetic in Java enables better compiler analysis because the aliasing issue is removed. We plan to do compiler analyses that will take advantage of this feature.
- Java's memory model and garbage collection make memory management easier and this removes a major source of difficult bugs, thereby improving programmer productivity.
- C++ is a hard language to parse. In fact, Charm++ does not parse user C++ code, but requires the user to write an interface file to identify parallel objects and other special Charm++ entities. Java is a simpler language than C++, thus enabling a Jade parser.
- In Charm++ when parameters are packed for transmission, the user needs to make some simple but essential modifications when passing an array as a parameter. The user also has to write a pack-unpack (PUP) method containing pointer-following codes if the object being passed contains dynamically allocated data and/or has pointers. All this is done automatically in Jade. This is not done automatically in Charm++ because of the lack of a parser and due to the aliasing issues that arises when working with pointers.
- To support migration and load-balancing in Charm++, the user needs to write pack-unpack (PUP) methods for all classes that might be migrated, whether or not the class has pointers. Without this step, which some users of Charm++ find cumbersome, the advantages of automatic load-balancing will not be realized. Since these methods can be auto-generated based on the data members of the class and since Java does not have pointers like Charm++, this is done automatically for the user in Jade.
- Java has limited support for parallel programming. It is run-time compiled and usually slower than compiled code. Java provides threads and simple synchronization, but assumes a shared-memory model in which threads run, making it hard to run Java codes in parallel on multiple nodes. Distributed parallelism could be achieved using RMI, but Java RMI is very inefficient and puts the burden of writing message-passing code on the user. There are no facilities for global synchronization and communication such as reductions and broadcasts. Java lacks higher level parallel constructs such as parallel arrays. And so on. By introducing message-driven features into Java, adding global communication and synchronization, providing higher-level language constructs for parallel objects, and making it compiled rather than run-time compiled, Jade offers a sturdy base for parallel programming in Java.

Our approach uses Java's syntax and basic semantics, extending them with Charm++ parallel constructs, and translating Jade to C++/Charm++. This retains the performance benefits of C++ while providing the user with a clean programming model.

The paper surveys some of the past work on parallel Java in Section 2. Section 3 describes the execution model, language constructs and other features of Jade, and then compares and contrasts Jade with both Charm++ and Java. We then discuss some implementation details and translation issues in Section 4. Finally, we summarize the work presented and, keeping in mind that this is just a start to the work we plan to achieve with Jade, we discuss our future plans.

2 Related Work

Java has built-in features to support parallelism: typically, Java threads are used for shared memory applications and either Java RMI or the standard socket interface is used for distributed applications. Java RMI is slow, interpreted, requires explicit name binding, and provides no optimization for local objects.

The JavaParty[20] project takes an approach to parallel Java that is based on improving Java RMI. The execution time of a Java RMI call is analyzed and divided into time for low-level communication, serialization of parameters, and RMI overhead. Each of these issues is addressed by them with the ParaStation, UKA-Serialization[19], and KaRMI[17] projects, respectively. JavaParty code is interpreted, whereas Jade is compiled.

The Manta[1] project also works by improving Java RMI. It uses a compiler approach to generate specialized serializers, implements its own runtime and protocol to speed up RMI, and can use a fast low-level communication layer (Panda/LFC) to improve speed. Manta supports runtime generation of marshaller code, distributed garbage collection, and compiler optimizations such as execution of non-blocking remote methods without a thread context switch. (This last is an automatic feature of Jade since blocking methods are not supported in the parallel constructs.) Manta interoperates with standard JVM and supports the Sun RMI protocol.[14, 13]

Titanium[23] translates Java into C. It adds a global address space and multidimensional titanium arrays to Java. However, on distributed memory architectures, the authors admit that accessing global data is significantly slower than accessing local data. Several compiler analyses are performed. Barriers and data exchange operations are used for global synchronization.

Our approach differs from JavaParty, Manta, and Titanium in that we introduce a different parallel programming style, the message-driven style, into Java. We also support object migration and load-balancing.

One of the authors has also implemented a previous version of parallel Java.[10] In that version, parallel constructs such as *chares* were added to Java. A JVM was executed on each processor of a parallel machine, and messages were injected into them using JNI and Java Reflection, i.e. the Java code was interpreted on a JVM. Parallel arrays, migration of objects, and global communication and synchronization were not supported.

```
 1 /** Simple parallel hello program using ChareArrays.
 2  main sends a message to hi[0], which sends it on to the
 3    next element, and so on.*/
 4 package Hello;
 5
 6 public synchronized class TheMain extends Chare {
 7     public int main(String []args){
 8         ChareArray hi1 = new Hello[5];
 9         hi1[0].sayHello(21);
10     }
11
12     public void memberMethod(int parameter) {...}
13 }
14
15 public synchronized class Hello extends ChareArray1D {
16     public Hello(){}
17
18     public void sayHello(int number) {
19         if (thisIndex < 4)
20             thisProxy[thisIndex+1].sayHello(number+1);
21         else
22             CkExit();
23     }
24 }
```

Fig. 1. Example Jade program: Array Hello.

3 Jade Parallel Constructs

We first describe the programming model and parallel constructs of Jade and then compare it with Charm++ and Java.

3.1 Jade

A Jade program consists of a number of parallel objects distributed across the available processors. The Jade runtime system manages the location, migration, message-forwarding, and load-balancing of these objects. Sequential objects in the system are owned by parallel objects and are tightly encapsulated within them, as explained below.

Figure 1 shows an example Jade program. Line 4 declares the Hello package which contains the subsequent entities. If no package is declared, the entities belong to a default null package.

Chares: A *chare* (line 6) is essentially a Java object without an explicit thread of control. When a method of the chare is invoked, it executes to completion without interruption. This implies that only one method of a chare can execute at a time, i.e. in the Java sense it is as if all methods are `synchronized`. A chare is declared by specifying the `synchronized` attribute for the class, and by extending from the `Chare` class, which implements Serializable.

Jade does not assume a shared memory model. A chare can have member variables, which can be instances of sequential classes or other chares. However, because a chare is a parallel object, its member variables are not permitted to be public. Member functions (which, of course, can include access methods) can be public. They need not be `void`, but when invoked asynchronously, the return value is discarded.

Jade execution model: Execution of a Jade program begins by instantiating one copy of every chare that has a `main` (see line 7 of Figure 1), and then invoking the `main` function of the chare on processor 0. This is different from the standard Java behavior, in which `main` is `static`, and only one `main` is invoked, without instantiating the class containing it. Typically, the `main` functions create parallel objects and invoke methods on them to get the work started. The reason we instantiate the class containing `main` is because most programs need an object to send back results to, print them out, etc. Usually one `main` is sufficient for a program, but multiple `main`'s are supported for Jade module initialization. Note that `ChareArrays` (described below) cannot have `main` functions.

ChareArrays: Jade supports 1D, 2D, and 3D arrays of parallel objects as its primary parallel construct. Support for user-defined index types is part of the language, but is not yet implemented. The arrays are sparse and created lazily, i.e. each element of an array is only created when first accessed. Each element of the array is essentially a chare. The elements of the array are mapped by the system to the available processors.

A parallel array is declared by making the class `synchronized` and extending `ChareArray1D`, `2D` or `3D`. Line 15 of Figure 1 shows the declaration of a 1D array.

Line 8 of Figure 1 shows the instantiation of a ChareArray containing five elements. (The language supports ChareArray constructors with arguments but the discussion of the syntax is omitted due to space constraints.) The next line invokes the sayHello method of the first element (index value 0) of the ChareArray. The `int thisIndex` member of ChareArray1D (line 19) returns the index of the current element, and the `CProxy thisProxy` member returns a proxy to the array itself. To broadcast a method invocation to all elements of an array, leave out the index, e.g.`A.sayHello()`.

Asynchronous method invocation(AMI): Invocation of methods on chares and ChareArray's is usually asynchronous, i.e. the caller of such a method does not block and wait for a response, e.g. see line 9 of Figure 1. A feature of asynchronous invocation is that it makes (invocation-related) deadlock impossible. AMI is just like sending a message; but the associated features of the message-passing programming style popularized by MPI are not present, e.g. in Jade there is no corresponding blocking receive for messages.

Note that AMI does not directly permit traditional blocking function calls (possibly with return values). This can be achieved by splitting up the calling method into two methods, the first of which contains the code up to and including the AMI call. The called function must then send back a message to the second method to resume the computation.

Parameter passing: Method invocation can be either local or remote, depending on the location of the called chare (i.e. on which processor of the parallel system it is running). Parameters are passed between parallel objects by copying them over via a message: this involves packing and unpacking of parameters. Parameters are automatically packed and unpacked without user intervention by the Jade translator-generated code. Chares and ChareArrays can also be sent as parameters; however, they are passed by reference.

Since a Jade parallel object cannot have public data members, and since parameters are passed by copy between parallel objects, we can conclude that the address space of a parallel object is private to it and cannot be directly accessed by any other parallel object. This tight encapsulation of data facilitates migration of objects.

Unlike traditional languages, (but similar to Java with threads) `main` can terminate without causing the program to end. Thus, on line 9 of Figure 1, `main` does not block waiting for the call to `sayHello()` to return. Control moves on to the next statement, and, in this case, since there is none, `main` terminates. Therefore, in Jade the end of computation is signaled by calling `CkExit()`, as shown in line 22 of the example program. CkExit gracefully terminates the program on all processors, ensuring that all messages have been delivered, flushing buffers, shutting down processes, etc.

3.2 Comparison with Charm++

The current implementation of Jade supports only asynchronous method invocation. Charm++ also supports *Synchronous method invocation(SMI)* through `threaded` and `sync` methods. `threaded` methods have an explicit thread of control and can therefore block. The Charm++ system still ensures that only one method of the chare containing the `threaded` method executes at a time, i.e. if a thread is blocked, no other method of the chare is invoked. This protects data accesses, i.e. no locking is required when accessing member variables. Calls to `sync` methods will block, and so they can only be called from `threaded` methods. There is no fundamental obstacle to implementing SMI in Jade and we plan to do this in the next version.

Charm++ only automates packing and unpacking of basic parameter types, but requires user intervention for passing arrays, nested objects, and pointers. Jade completely automates parameter passing.

3.3 Comparison with Java

Given the large amount of research into translation of (sequential) Java to C, we decided to focus our efforts on the parallel features rather than on implementing all Java features.

Java import statements are translated into #include's with a hierarchy of directories. Java `packages` and package-private classes are supported.

Differences: As mentioned above, parameter passing between parallel objects in Jade is by copy (except that parallel objects are passed by reference), whereas Java parameter passing (which is retained in Jade for non-parallel objects) is by copy for primitive types and by reference for objects.

Java supports dynamic loading of classes based on the `CLASSPATH`. However, our compilation product is a single executable and dynamic loading is not supported. It is possible to implement this capability in C++, and if applications require it, we will add this feature to Jade.

Multiple classes can be defined in one Jade source file, unlike Java which has a very tight tying of classes to the naming structure to simplify dynamic class loading.

Jade treats `main` differently from Java as described in Section 3.1.

Java bytecode is portable across heterogeneous systems, while Jade code is compiled to a particular architecture. Java primitive types are standard across all platforms, but Jade types depend on the machine and C++ compiler used.

Restrictions: Java names can be Unicode, whereas we support only ASCII. We do not currently support the standard Java runtime and libraries, nor do we support exceptions. We plan to implement non-preemptive threads in the near future, as well as garbage collection.[2]

4 Translation Issues

Our translator is implemented using the ANTLR parsing system which supports Java as an implementation language. For our grammar, we used a public domain ANTLR Java grammar.[16]

The steps and byproducts involved in compiling a Jade source file are shown in Figure 2. The Jade source file (`.pjs`) is translated into four files: the Charm++ interface file (`.ci`), a `.h` header file and a `.C` C++ source file

The `.ci` file is then fed to the Charm++ translator. It contains information about the parallel constructs in the program. The Charm++ translator generates `.decl.h` and `.def.h` files, which are `#include`'d in the `.h` and `.C` files respectively.

All the above generated files apart from the `.ci` file are then compiled with a C++ compiler and linked with the Charm++ and the Jade libraries to generate a single executable `a.out` file.

5 Implementation

It is important to remember that the objective of Jade is to retain the performance advantages of Charm++ while gaining the simplicity and ease of debugging of Java. It is not intended to outperform Charm++, but to bring the benefits of the message-driven programming style to Java.

[2] `delete` is supported in the interim.

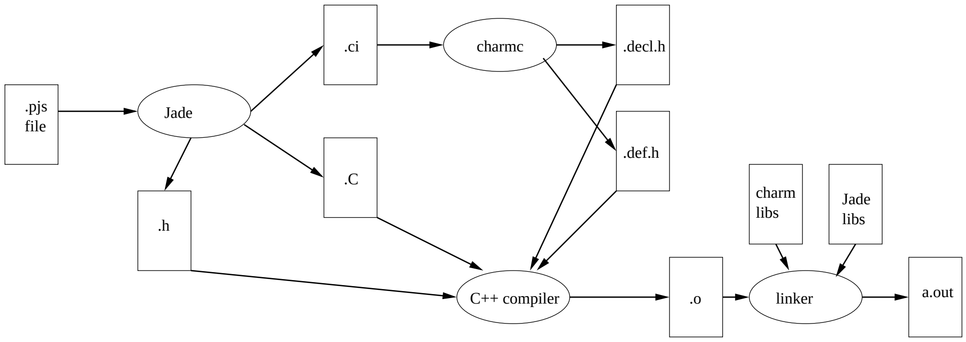

Fig. 2. Compilation of a Jade source file.

We implemented a Jacobi 2D decomposition example in Jade. The resulting translated code runs on all the platforms supported by Charm++, such as the Intel ASCI-Red, SGI Origin2000, IBM SP-2, Cray T3E, Convex Exemplar, clusters of Unix (Linux, Solaris, etc) workstations, and even single-processor UNIX machines. Fig. 3 shows the speedup on the LeMieux cluster at Pittsburgh Supercomputing Center. (Each node of LeMieux contains four 1 GHz Compaq Alpha CPU's; the nodes are connected by a high-speed Quadrix Elan interconnect.)

The implementation has demonstrated automatic parameter marshalling, automatic object migration and measurement-based load-balancing, efficient execution on both shared and distributed architectures, the ability to shrink and expand the number of processors used on-demand at runtime, and automatic checkpointing of the data.

The performance is essentially the same as Charm++. No additional overhead is introduced in sequential or parallel constructs, messaging, etc.

6 Summary and Future Work

We have presented Jade, a native compiled parallel Java using the message-driven parallel programming style. Jade parallel constructs include parallel objects called *Chares* and parallel multi-dimensional arrays called *ChareArrays*. Parallel objects can migrate across processors, and automatic measurement-based load balancing ensures even load distribution. Communication is through asynchronous method invocation and includes automatic parameter marshalling. Messages are forwarded to the target object even if it has migrated. Barriers and reductions are supported.

Our plans for further research include the enhancement of Jade with parallel garbage collection, threads and synchronous method invocation, and the addition of compiler optimizations in the translation process.

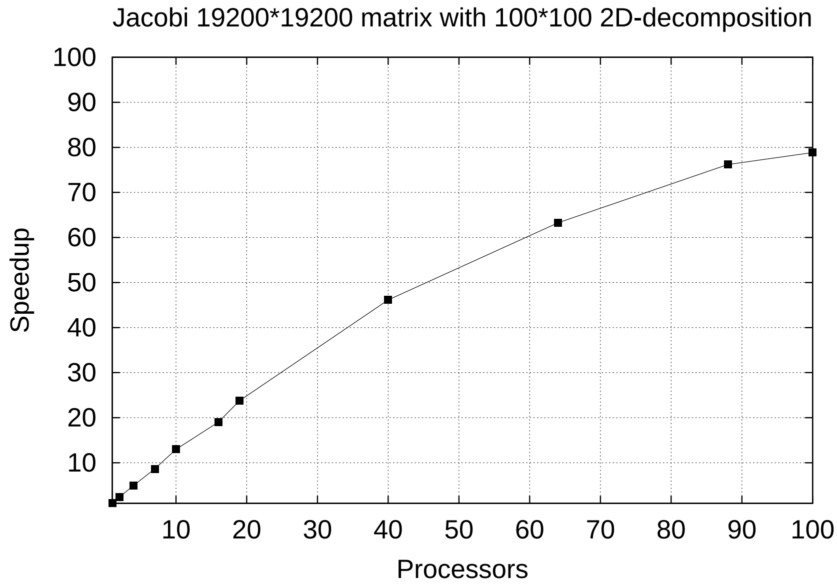

Fig. 3. Performance of Jacobi 2D written in Jade.

References

1. H. Bal et al. Manta: Fast parallel Java. URL: http://www.cs.vu.nl/manta/.
2. M. Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.
3. M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.
4. R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999. Submitted for publication.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
6. L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.
7. L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.
8. L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of*

4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico, volume 1800, pages 1152–1159, March 2000.
9. L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.
10. L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel java with a global object space. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, pages 235–244, Las Vegas, Nevada, July 1997.
11. L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
12. L. V. Kalé, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.
13. J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.
14. J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proc. Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.
15. M. W. Macbeth, K. A. McGuigan, , and P. J. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *Proc. CASCON 98*, pages 40–54, Missisauga, ON, Canada, 1998.
16. J. Mitchell, T. Parr, et al. Java grammar for ANTLR. URL: http://www.antlr.org/grammars/java.
17. C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*, San Francisco, CA, June 1999.
18. T. Parr et al. Website for the ANTLR translator generator. URL: http://www.antlr.org/.
19. M. Philippsen and B. Haumacher. More efficient object serialization. In *Parallel and Distributed Processing, LNCS, International Workshop on Java for Parallel and Distributed Computing*, volume 1586, pages 718–732, San Juan, Puerto Rico, April 1999.
20. M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.
21. J. Phillips, G. Zheng, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.
22. J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.
23. K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13), September – November 1998.