A. Summary

This proposal is concerned with enabling parallel computation in a world full of Internet technologies. On the hardware side these will include massively parallel engines designed and deployed as Internet servers. On the software side, they will include software developed in network-aware programming languages like Java—software engineered to survive in heterogeneous and very dynamic environments. We want to refine MPI-like programming models and APIs for high performance programming in these environments—researching ways to get the fastest possible message passing from languages like Java, and ways to exploit novel associated technologies like Jini to produce richer message-passing environments.

The work proposed here would follow on naturally from the *HPJava* project, ongoing formerly at Syracuse and now at Florida State University. In that project we are developing a parallel programming model that adds certain extensions to the base language for handling distributed data. While the original HPJava project emphasized issues relating to translation of extended languages, the work proposed here will emphasize underlying run-time technologies for parallel programming in Java-centric environments.

Java introduces implementation issues for message-passing APIs that do not occur in conventional scientific programming languages. We will investigate how to apply ideas from projects such as Jaguar and JaVIA to MPI-like APIs, to reduce the overheads of the Java Native Interface. We will integrate ideas on efficient object serialization from projects like KaRMI with MPI-specific methods for handling complex datatypes. Simply providing efficient MPI-like APIs for Java is not enough. The programming model must address features specific to distributed computing. To better support computing in volatile Internet environments, we will need features like dynamic spawning of process groups, parallel client/server interfaces, and new mechanisms for handling failures. Jini provides a natural framework for implementing these features. An important emphasis will be on researching synergies between parallel message-passing programming and Jini-like systems. By combining ideas from MPI with ideas from Jini we aim to create an environment that encourages scalable, fault-tolerant parallel computing. Finally, we will explore uses of Jini in a middle tier for initiating parallel MPI jobs, where the parallel programs may be written in languages other than Java.

In the end, Java is one language amongst several contenders. To focus our activities we have concentrated on Java, because we believe it encompasses much of what is happening in the Internet today. Of course we hope and believe that the lessons learnt will have relevance beyond any single language.

B. Table of Contents

Replace with NSF Form 1359.

C. Project Description

C.1 Introduction

This proposal is concerned with enabling parallel, high-performance computation in a world rife with Internet technologies. Parallel computations may or may not be distributed *across* the Internet, but in any case we assume the commodity hardware and software technologies that will be most important in the immediate future will be fine-tuned for the Internet environment. These technologies will include massively parallel engines designed and deployed as Internet servers, and software developed in network-aware programming languages like Java—software engineered to survive in heterogeneous and very dynamic environments.

Partly through the activities of the *Java Grande Forum* [29] (and, we expect, the newly formed *Jini Activity Group* of the Global Grid Forum) the prospect of using Java for essentially "scientific" computing has has become increasingly realistic. Work in industrial and academic sectors on optimizing compilers, JITs, language enhancements and libraries have made it increasingly likely that future Java environments will meet the performance constraints for large-scale computations and simulations. The work on improving the performance of Java is driven largely by its industrial application as a programming language for high-performance Internet servers, but we assume scientific programmers may also reap the benefits.

The proposed work will further develop ongoing research by the current authors into application of Java for parallel computing. Our *HPJava* system is based around a small set of language extensions designed to support parallel computation with distributed arrays. This work received funding from an earlier NSF grant (see Section C.4.5). While the ongoing work emphasizes use of language extensions and translators to provide a high-level programming environment—assuming a native interface to a underlying, conventional parallel programming environment—the proposed work will particularly address uses of Java in the implementation of the *underlying* environment.

An influential development in parallel computing was the publication in 1994 of the Message Passing Interface (MPI) standard [32]. MPI supports the Single Program Multiple Data (SPMD) model of parallel computing, providing many modes of reliable point-to-point communication, and a library of collective operations. The MPI standards specify bindings for Fortran, C and C++. But none of these languages is especially adapted to the Internet, where code is often loaded dynamically across the network, resources are frequently discovered and lost spontaneously, and fault tolerance is a crucial issue.

In the proposed work we will be concerned with using network-oriented languages and in particular Java for high-performance computing. One preoccupation is with refinement of MPI-like programming models and APIs for high performance programming in Java—researching ways to get the fastest message passing from Java, and ways to exploit novel Java technologies like Jini (a Java architecture for making services available over a network) to produce richer message-passing environments. A complementary concern is with use of Jini in a middle tier between client and MPI-based parallel services.

Java introduces implementation issues for message-passing APIs that do not occur in conventional programming languages. One area of research is how to transfer data between the Java program and the network while reducing overheads of the Java Native Interface. We will investigate how to apply ideas from projects like Jaguar [44] and JaVIA [12] to MPI-like APIs. Another important issue is how to minimize the overheads of serialization in communicating Java objects and multidimensional arrays. We hope to integrate ideas on efficient object serialization from the KaRMI project [39], for example, with MPI-specific ideas we started to explore in [9]. We will be especially interested in supporting efficient communication of scientific array objects like those supported by the Java Grande Numerics Working Group.

The programming model must address features specific to distributed computing. To support computing in volatile Internet-oriented environments, we will need features like dynamic harnessing of process groups and parallel client/server interfaces. These are features that the MPI Forum started to address in the MPI 2 standard, but implementations in conventional parallel programming environments have been slow to arrive. A natural framework for dynamically discovering new compute resources and establishing connections between running programs already exists in Sun's Jini project.

In the proposed work, an important emphasis will be on researching synergies between parallel message-passing programming and Jini-like systems. One defining characteristic of distributed computing is the presence of *partial failures*. By combining ideas from MPI with ideas from Jini we hope to facilitate an environment that encourages scalable and fault-tolerant parallel computing.

C.2 Background and Motivations

C.2.1 High-Performance computing and Java

We believe that Parallel computing must adapt itself to the Internet environment, and embrace current Internet technologies. Many people accept that the Java language is likely to continue its important role in Internet software, but the idea that it should also be adopted as an important language for large-scale technical computations still meets some resistance. Over the last few years supporters of the *Java Grande Forum* have been working actively to address some of the difficulties. The goal of the forum has been to develop consensus on enhancements to the Java language and platform to support large-scale ("Grande") applications. Through a series of workshops and conferences [1, 2, 18–20] the forum has helped stimulate research on Java systems and applications.

Associated developments have helped establish the case that Java can meet the vital performance constraints for numerically intensive computing. A series of papers from IBM [34, 35, 45], for example, demonstrated how to apply aggressive optimizations in Java compilers to obtain performance competitive with Fortran. In a recent paper [36] they described a case study involving a data mining application that used the Java Array package supported by the Java Grande Numerics Working Group. Using the experimental IBM HPCJ Java compiler they reported obtaining over 90% of the performance of Fortran.

The Java Grande Forum has a Concurrency and Applications Working Group. This group has been looking directly at uses of Java in parallel and distributed computing. We will refer to some of this work below.

C.2.2 Niches for parallel Java programs

Computers that host major Web sites will either be multiprocessors or clusters of workstations. We already see this today, and the trend will presumably continue. Since Java and parallel computers will coexist in Internet servers, this is one place we expect to see roles for Java-based parallel computation emerging. We may, for example, see compute-intensive services appearing on the Web—perhaps supporting data-mining queries that use parallel algorithms or financial analysis programs involving complex simulations.

Truly scalable servers are likely to be clusters rather than symmetric multiprocessors. As a specific example, consider the *Ninja* vision of the future of the Internet elaborated by researchers at UC Berkeley [41]. In their view a service should be *scalable* (able to support thousands of concurrent users), *fault-tolerant*, and highly-available. While a major concern is with mobile code for service deployment, eventually services must maintain persistent state. In the Ninja model this is held in a controlled environment engineered to provide high availability and scalability—namely a cluster of workstations [23]. This *Base* is not necessarily homogeneous and it is not completely "reliable", so it is not exactly a conventional parallel computer. But this is an environment where we might expect message-passing parallel programs written in Java to find a natural home.

A completely different place where we might see early uptake of Java-based parallel computing is in the classroom. Java has become an important teaching language in Universities. For teaching parallel computing principles to students, Java is likely to be a more attractive language than Fortran. For example, our *mpiJava* software [37] has been downloaded by about 500 people. On the basis of project descriptions given when people download the software we estimated that perhaps 10% of potential users are teachers looking for classroom software. This is a not a dominant proportion, but it is an influential one so far as future uptake is concerned. In this context highly tuned implementations are not essential. A pure Java MPI-like package that is portable and can be installed easily on available networks of PCs is probably ideal.

Finally, because of its platform independence, mobility, and other associations with the Internet, Java is a natural candidate as a language for *metacomputing*. We interpret this to mean computation by parallel programs distributed across the Internet itself. Within the MPI community there is an ongoing effort to extend MPI specifications and implementations to support metacomputing, by allowing logical process groups to span geographically separated clusters and supercomputers [28]. Java-based metacomputing can exploit and supplement these ongoing MPI activities in natural ways. Many authors have discussed Java approaches to metacomputing. Charlotte [7, 8] and Javelin [13, 38] concentrate on harvesting cycles of computers running Web browsers by downloading *applets* to them. JavaParty [39, 40] and Manta [42] support an interacting SPMD style of distributed programming, but emphasize communication through *remote method invocation* (RMI). This work is clearly important, but our interest leans towards more tightly coupled parallel environments, and in fact metacomputing will not be a *primary* focus of the work proposed here. In more tightly cooupled environments it is more questionable whether remote method invocation is the best model of communication within a running parallel program.

The message-passing model of synchronization seems a better fit to the requirements.

While remote method invocation may not be the best model for communication within an *executing* parallel program, we assume that RMI-based technologies can play a natural role at the level of *coordinating* computational resources for execution—discovering, monitoring and recovering those resources. The harmonious integration of message-passing models for parallel computing with remote object models at some outer level is an important theme in the proposed work.

C.2.3 Jini

Jini is Sun's Java architecture for making services available over a network. It is built on top of the Java Remote Method Invocation (RMI) mechanism. The main additional features are a set of protocols and basic services for "spontaneous" discovery of new services, and a framework for detecting and handling *partial failures* in the distributed environment.

A Jini lookup service is typically discovered through multicast on a well-known port. The discovered registry is a unified first point of contact for all kinds of device, service, and client on the network. This model of discovery and lookup is somewhat distinct from the more global concept of discovery in, say, the CORBA trading services or HP's e-speak [26]. The Jini version is a lightweight protocol, especially suitable for initial binding of clients and services within multicast range. In the Ninja outlook, for example, Jini technology might fit comfortably at the periphery, near the end-user devices, or *within* the Base, addressing initial federation of nodes, crashes of individual nodes, etc. The latter the kind of environment that especially interests us.

The ideas of Jini run deeper than the lookup services. Jini completes a vision of *distributed programming* started by RMI. In this vision *partial failure* is a defining characteristic, distinguishing distributed programming from the textbook discipline of concurrent programming [43]. So in Jini remote objects and RMI replace ordinary Java objects and methods; garbage collection for recovery of memory is replaced by a *leasing* model for recovery of distributed resources; the events of, say, AWT or JavaBeans are replaced by the distributed events of Jini. Finally, the synchronized methods of Java are mirrored in the nested transactions of the Jini model. Concurrent programming is not an identical discipline to scalable parallel programming, but we need analogous sets of abstractions for the parallel case.

We are not alone in recognizing the potential of Jini for coordinating resources in the context of large-scale computations. Very recently a Jini Activity Group has been formed under the umbrella of the Global Grid Forum [24]. This group will investigate ways in which the Jini philosophy and technology can be used as an infrastructure with Grid environments [25], especially for supporting resource and service discovery. We plan to be actively involved in the workings of that group.

While a proposal of this nature should not focus exclusively on a proprietary (though open) technology like Jini, it seems likely that we can exploit Jini and related systems to create improved parallel computing environments, especially for Java.

C.2.4 Lessons

To support the parallel programmers of the future we believe we need Java implementations of lightweight messaging systems akin to MPI—perhaps the most successful platform developed for parallel computing. A likely physical setting is in the more or less tightly coupled (but probably heterogeneous, multi-user) clusters of trusted workstations that will host the Web services of the future. While models of distributed programming other than message-passing (including Linda-based models like JavaSpaces or JavaNOW) certainly have a role, experience with earlier generations of parallel computer suggests that the low-latency message-passing model is a good fit to requirements.

But these are likely to be volatile environments that demand the reliability provided by foundations like Java and Jini. The software must be adaptive. Availability changes as workloads and network traffic fluctuates—old nodes crash, new ones are attached and discovered on the fly. Jini is a leading candidate for dealing with these situations, and we expect it can play an important role in implementing parallel-programming systems for these new environments.

C.3 Goals of the Proposed Work

There are three related strands in the proposed work. One strand will investigate use of Java and especially *Jini* technologies in a middle tier for initiating parallel MPI jobs. For definiteness we refer to the architecture as *JiniMPI*. So far as the architecture itself is concerned the parallel program could be written in Java or some other language that invokes an MPI-style message layer. A second activity will investigate implementation-level issues related to developing high-performance message-passing systems *in* Java (and potentially other object-oriented network programming languages). We need to know how to efficiently implement associated APIs in dynamic environments like Internet servers and networks. The specific APIs will be derivatives of the *MPJ* specifications from the Java Grande Message-Passing Working Group (section C.4.3). A third strand will continue development of our *HPJava* system. HPJava is a relatively high-level environment for parallel computing in Java. In the new work, we would plan, in particular, to adapt the run-time system of the existing HPJava software to work in the Java environments discussed above. Besides its intrinsic value, this exercise will stress-test the new, underlying, pure-Java layers that we propose.

In principle the three levels, *JiniMPI*, *MPJ*, and *HPJava*, are independent, but they complement each other and are expected to be especially powerful when used together.

C.3.1 Jini-based middle tiers for parallel programming

The envisaged JiniMPI architecture is illustrated in Figure C.1. The architecture supports MPI-based parallel computing. It also includes ideas from systems like Condor and Javelin. Our diagram only shows the server layer (bottom) and the service layer (top). There would also a client layer that communicates directly with the "Control and Services" module.

Each workstation has a *Jini Parallel Computing Embryo*—a Jini service that registers the availability of the workstation to run applications. The Jini embryo is the represen-

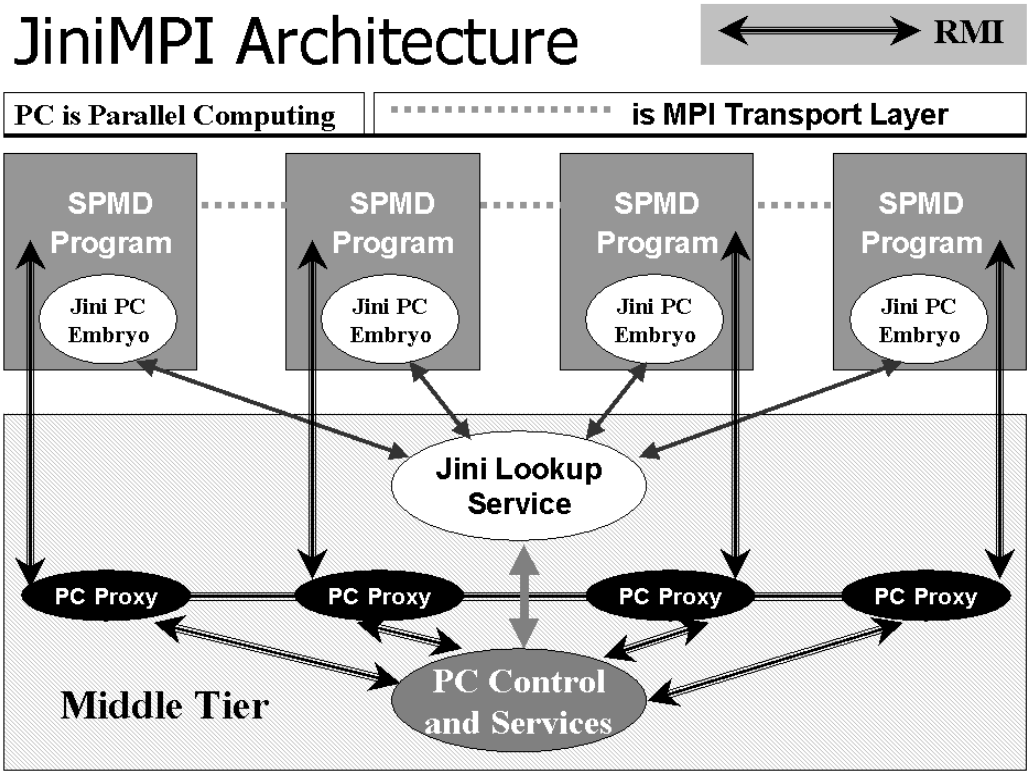

Figure C.1: JiniMPI Architecture

tative of the machine—advertising its availability to run general applications or particular software. The *Gateway*, or *Control and Services module* [3], queries the Jini lookup services to find appropriate computers to run a particular MPI job. The mechanism could be used just to be run a single job, or to set up a farm of independent workers

The Gateway receives a proxy for each embryo it selects from the lookup services in the usual Jini way. Once an initial RMI link is established from the Gateway to each SPMD node, Java proxies are created in turn for the individual node programs. The node programs themselves can be written any language (Java, Fortran, C++ etc). Acting on behalf of the client, the Gateway-Embryo negotiations will also supply the Gateway with any additional data needed (e.g., specification of required parameters for the job and how to input them). An important theme is the separation of *control* from *data transfer*. In the "control layer" we have Jini services (registration, lookup and invocation), other services such as load balancing, and fault recovery. In a "fast transport layer" we have MPI style data messages. The Jini embryo is used to initiate (and perhaps monitor) the process. It takes only a backseat role in the actual execution of the parallel program.

We will also investigate the implications of using a JavaSpace in the control layer as the basis for a management environment. Note this is very different from using Linda or JavaSpaces at the execution level, and performance problems are not likely to be a primary concern.

The proposals here would build on research in the ongoing *Gateway project* [21]. This is an effort to build computational web portals that allow users to access high performance computing facilities via web browsers like Netscape and Internet Explorer. The goal of Gateway is to provide a high level user interface that simplifies access to various computing resources maintained by computing centers with varying access and security policies. Gateway provides a commodity-based solution to these problems, taking advantage of the investment of the commercial sector into such technologies and standards as CORBA, XML, and Java.

C.3.2 Research on high-performance message passing models for Java

In the early stages of the project we will complete a reference implementation of the *MPJ* specification, an MPI-inspired Java API from the Java Grande Message-Passing Working Group (section C.4.3). Our existing *mpiJava* software (section C.4.2) has been one basis for this work, but for further research a portable, *pure Java* version will be needed. Section C.4.4 describes a design. One of the conclusions of the design study was that difficult issues of reliability (and useability) in a network environment are naturally addressed in the framework of the Jini programming model. An initial reference implementation is likely to make extensive use of Jini for job initiation and handling failures. This implementation will be a foundation for subsequent research in the project.

The current MPJ specification supports essentially MPI-1.0 functionality, with some extensions specific to object-oriented languages (for example it has the facility to send and receive arbitrary serializable objects). With the reference implementation in place, the project will follow two directions

- Research into optimizations specific to the language and network context, to improve bandwidth and latency.
- Design and pilot implementation of extensions to the basic message-passing model, improving support for highly dynamic environments.

Associated tasks are detailed in the following two subsections.

Fast message-passing for Java

An initial reference implementation will probably use Java sockets as the basic transport. But research is needed into different approaches to low-level transport—for example calling native MPI by standard JNI or other methods, or using Java bindings to lower-level interfaces like VIA. It is also likely to involve work on improving the efficiency of object serialization, or exploiting the research of other groups on efficient object serialization.

Our earlier work on mpiJava already exploited the Java Native Interface (JNI) to call native MPI implementations. But there are concerns with the efficiency of crossing the JNI barrier. Detailed criticisms of JNI can be found in [44] and [12]. The two groups involved have described ways to go beyond standard JNI, specifically to support efficient Java interfaces to VIA. Our research would attempt to leverage this work, either by adapting similar techniques to make low-overhead interfaces to native MPI, or (for suitable platforms) adopting Java VIA interfaces as a low-level transport in our Java implementation of the message-passing API (along lines comparable with [14]). Note that the new approaches often assume limited changes to compiler or JVM, at least in the garbage collector.

Another area that needs further research is specific to object-oriented languages. To allow fast communication of *object graphs* between processors we need very efficient object serialization. Important work on improving Java serialization has been described in [39]. The authors report that their UKA-Serialization can save 76% to 96% of the time needed to serialize objects. Their work was done in the context of an optimized reimplementation of RMI, but we can use the same software in fast MPJ implementations. We may hope to combine ideas from UKA-Serialization with the MPI-specific ideas we started to explore in [9], to facilitate fast communication of objects in MPJ. For technical computation an especially pressing concern is the case of multidimensional arrays (*one*-dimensional arrays can often be communicated without expensive serialization).

Extending the MPJ message-passing model

As noted above, an initial MPJ draft specifies functionality similar to MPI 1.1, complemented by some object-oriented features.

One set of extensions to this draft will be inspired by features of the MPI 2 standard. This standard is not yet widely implemented even for traditional languages, but it includes features that are likely to be important in the volatile Java environments we target. Dynamic process creation was not part of MPI 1 but it is undoubtedly important for our environments. An early addition to the baseline MPJ model will be operations similar to those in MPI 2 start new groups of processes and return intercommunicators connecting them to the inital group. A new feature in our research will be adoption of Jini (or similar technologies) for discovery of the required computational resources. Another relevant feature of the MPI 2 specification is its introduction of a parallel client/server model, by which two running *parallel* programs (client and server) can establish a connection, and thus operate collectively for some period. MPI 2 proposes fairly limited functionality here, and it seems that a Java environment ought to be able to offer more flexible mechanisms modelled on Jini lookup.

Dynamic discovery of compute resources is one area where Jini-like ideas can help us. A more difficult issue for scalable computing in networked environments is how to deal with dynamic *loss* of resources, due to a failure somewhere in the distributed platform. As emphasized in Section C.2.3, the general Jini framework incorporates various mechanisms designed to support programming in the presence of partial failure. One goal of our research will be to explore ways to incorporate similar concepts in the parallel message-passing context, encouraging programs that are truly scalable even in the presence of partial failure. For example, it may be that a scalable version of the Jini *transaction* model is ideal to support checkpointing of running parallel programs. This is a more speculative area, but the goal would be isolate a Jini-like kernel of key concepts for fault recovery in SPMD parallel programs. Most likely these would be applied at the level of large, collective operations, such as global checkpointing, forking groups of slave processes for some subtask, and so on.

C.3.3 Continued development of the *HPJava* environment

A starting point for much of the work proposed here is our ongoing HPJava project [27]. The HPJava system involves a translator for an extended Java language with support for parallel programming with distributed arrays, and a "run-time system" which is a library of collective operations implemented on top of MPI.

The proposed work would include some further development of the HPJava translation system; we would be particularly interested in moving the underlying run-time system to operate on top of the portable pure-Java environments described in the previous subsections.

One reason this is important is that supporting the higher level HPJava distributed array communication model will give us a concrete application of the lower-level APIs, and in particular it will exercise the parts of the message-passing API that are concerned with communication of *arrays*. We consider this to be a basic issue for technical computing, and one that has not been addressed in a fully satisfactory way by some earlier proposals.

C.3.4 A three year workplan

Year one: In the first year we will complete ongoing work on a Jini-based pure Java implementation of MPJ—a message-passing environment with functionality similar to MPI 1.1. This implementation will be an important foundation for the subsequent research in the project. To support the later research, the transport layer must be easily replacable (similar to KaRMI, for example). This implies a layer analogous to the MPICH device level, but we will need new features to support Java and objects. Jini will be used for discovering compute hosts and (importantly) to ensure clean global termination in the event of failures. To prove the design, a native MPI-based implementation of the transport layer will also be implemented. This will complement the initial socket-based implementation, and will use standard JNI. (Our existing *mpiJava* software will probably be phased out.) In this year we will also be studying issues relating to concrete design of the JiniMPI Architecture, using Jini technologies as a gateway to parallel computing resources.

Year two: The basic message passing model will be extended with a version of dynamic process spawning using Jini to find resources, and a parallel client/server model, using Jini to establish connections between running parallel programs. Suitable Java-centric APIs will be designed. Relevant application codes from areas like parallel data-mining will be produced. We will study ideas following on from the Jaguar and JAVIA work at Berkeley and Cornell, and try to make use of those ideas to optimize our initial MPJ implementation, the goal being to develop genuine high-bandwidth, low-latency message-passing in Java. We will integrate UKA-serialization or successors, and further study MPI-specific improvements for important cases such as multidimensional arrays. Extensions needed to support efficient communication of the scientific Array classes supported by Java Grande will be a prioriy. Pilot implementations of JiniMPI architecture will be developed. We will investigate reliable checkpointing primitives for parallel programming, using Jini transactions, or related mechanisms.

Year three: The experience of the first two years will feed into a practical model of scalable Internet computing, combining parallel-programming ideas from MPI with Jini ideas about fault tolerance. This model will be integrated with the three-tier JiniMPI model.

C.4 Related Work

C.4.1 The *HPJava* parallel programming environment

We have been investigating a language model that combines characteristic data-parallel features from the HPF (High Performance Fortran) language with an explicitly SPMD programming style. A goal of this model (the *HPspmd model*) is to facilitate calls to libraries for parallel programming with distributed data from within an explicitly SPMD-parallel program.

In particular we have been implementing and testing these ideas within an environment called HPJava, built around and implementation of the HPspmd language extensions with Java as the base language [27].

The current HPJava compiler is a source-to-source translation tool taking a parallel program written in the HPspmd-extended Java language as input, and generating a standard Java program with classes describing the HPspmd distributed arrays. At the time of writing an initial version of the HPJava compiler is operational. Recently we successfully compiled and ran an 800-line multigrid benchmark, transcribed from an existing Fortran 90 program. The explicitly parallelized HPJava program is about the same length as the original (at best implicitly parallel) Fortran, encouraging us in the belief that we have a good model for parallel programming.

The current translator uses a relatively naive translation scheme. This will be replaced by a higher quality scheme over the next several months (within the original HPspmd project). Still the underlying communication libraries are based on Java interface to native MPI software—we will not have a fully portable pure-Java environment. One future task in the work proposed here would address this shortcoming.

C.4.2 Experience with *mpiJava*

mpiJava [6, 9, 37] is our object-oriented Java interface to MPI. The system provides a fully-featured Java binding of MPI 1.1 standard. The object-oriented API is modelled largely on the C++ binding that appeared in the MPI 2 standard. The implementation of mpiJava is through JNI (Java Native Interface) wrappers to a suitable native implementation of MPI. The software comes with a comprehensive test-suite translated from the IBM test-suite for the C version of MPI. Platforms currently supported include Solaris using MPICH or SunHPC-MPI, Linux using MPICH, and Windows NT using WMPI 1.1.

The MPI standard is explicitly object-based. The C and Fortran bindings rely on "opaque objects" that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The mpiJava

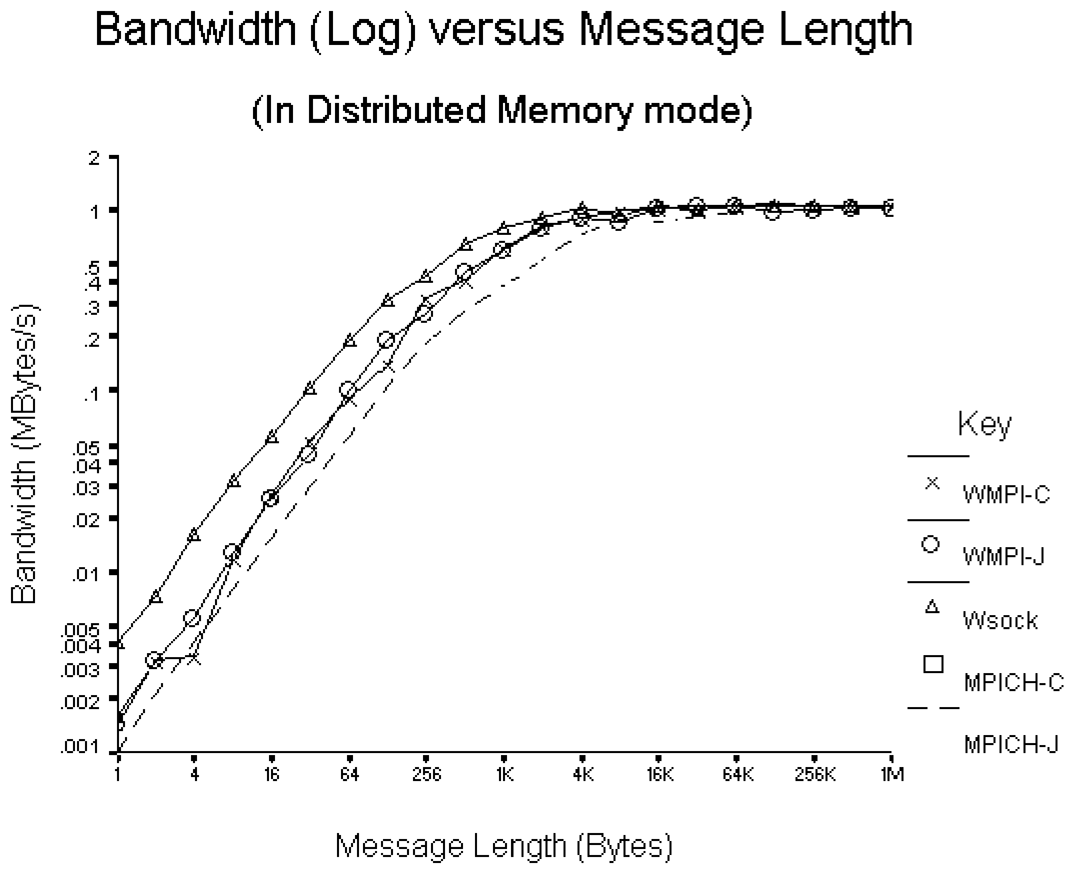

Figure C.2: PingPong Results in Distributed Memory mode

API follows this model, lifting the structure of its class hierarchy directly from the C++ binding.

The benchmarks in Figure C.2 compare mpiJava ("J") timings with native C timings for communication between a pair of PCs. The timings represent two different native MPI implementations (MPICH and WMPI), and also compare with with raw Windows sockets. We see that the mpiJava JNI wrappers introduce a modest extra latency relative to native MPI, but for large messages bandwidth is not compromised. Although these results are encouraging, we should remark that these benchmarks were run using the Classic JVM. Current JIT compilers will degrade bandwidth because Java arrays are usually copied when they are passed to native methods. How best to avoid such overheads is an important research question [12, 44].

mpiJava is part of the *HPJava* environment [27]. We are actively developing and supporting the software. Downloads run at about 30 per month.

C.4.3 Java Grande Message-passing Working group

Java bindings to MPI were developed independently by several teams. One Java MPI interface was produced by Getov and Mintchev [22, 33]. In their work Java wrappers were automatically generated from the C MPI header. This eased the implementation work, but did not lead to a fully object-oriented API. A subset of MPI was implemented in the

DOGMA system for Java-based parallel programming [31]. Dincer and Kadriy described an instrumented Java interface to MPI called *jmpi* [15]. Java implementations of the related PVM message-passing environment have been reported in [46] and [17].

The Message-Passing Working Group of the Java Grande Forum was formed as a response to the appearance of these diverse APIs. An immediate goal was to discuss a common API for MPI-like Java libraries. An initial draft for a common API specification was distributed at Supercomputing '98 [11]. Since then the working group has met in San Francisco and Syracuse, and a Birds of a Feather meeting was held at Supercomputing '99. Minutes of meetings were published on the *java-mpi* mailing list [30]. To avoid confusion with standards published by the original MPI Forum (which is not presently convening) the nascent API is now called *MPJ*. A version of the specification was published last year [10].

C.4.4 Case study: reference implementation of MPJ

Presently there is no complete implementation of the draft MPJ specification. Our own Java message-passing interface, *mpiJava*, is moving towards the "standard". The new version 1.2 of the software supports direct communication of objects via object serialization, which is an important step towards implementing the specification in [11].

The mpiJava wrappers rely on the availability of a platform-specific native MPI implementation for the target computer. While this is a reasonable basis in many cases, the approach has some disadvantages. For one thing the two-stage installation procedure—get and build a native MPI then install and match Java wrappers—can be tedious and discouraging to potential users. Secondly, in the development of mpiJava we sometimes saw conflicts between the JVM environment and the native MPI runtime behaviour. The situation has improved, and mpiJava now runs with several combinations of JVM and MPI implementation, but some problems remain. Finally, this strategy simply conflicts with the ethos of Java, where pure-Java, write-once-run-anywhere software is the order of the day.

Ideally, the first two problems would be addressed by the providers of the original native MPI package. We envisage that they could provide a Java interface bundled with their C and Fortran bindings. Ultimately, such packages would presumably be the best, industrial-strength implementations of systems like MPJ. Meanwhile, to address the last shortcoming listed above, we have outlined in [5] a design for a *pure-Java* reference implementation for MPJ. Design goals were that the system should be as easy to install on distributed systems as we can reasonably make it, and that it be sufficiently robust to be useable in an Internet environment. A particularly strong requirement is that in no circumstances should the software leave resource-wasting orphan processes lingering after an abrupt termination.

We are by no means the first people to consider implementing MPI-like functionality in pure Java. Working systems have already been reported in [15, 31], for example. Our goal was to build on some lessons learnt in those earlier systems, and produce software that is standalone, easy-to-use, robust, and fully implements the specification of [11].

We wish to simplify installation of message-passing software to a bare minimum. A user should download a jar-file of MPJ library classes to machines that may host parallel jobs, and run a parameterless installation script on each. Thereafter parallel java codes can be compiled on any host in the LAN (or subnet). An `mpjrun` program invoked on the development host transparently loads all the user's class files to available compute hosts,

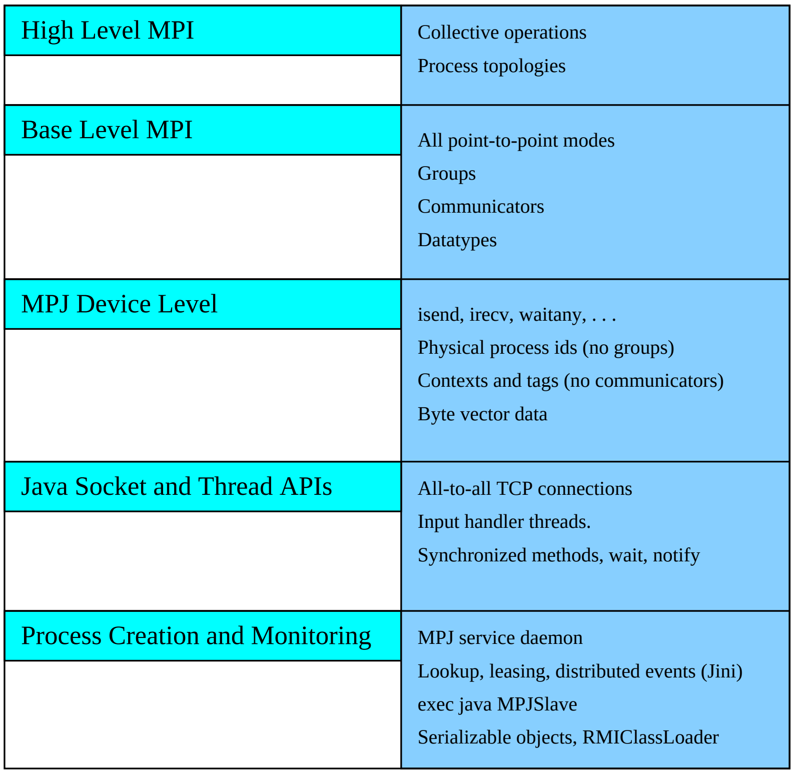

Figure C.3: Layers of proposed MPJ reference implementation

and the parallel job starts. The only *required* parameters for the `mpjrun` program should be the class name for the application's main program and the number of processors the application is to run on.

To be usable, an MPJ implementation should be fault-tolerent in at least the following senses. If a remote host is lost during execution, either because a network connection breaks or the host system goes down, or for some other reason, *all* processes associated with affected MPJ jobs must shut down within some short interval of time. On the other hand, unless it is explicitly killed or its host system goes down altogether, the MPJ *daemon* on a remote host should survive unexpected termination of any particular MPJ job. Concurrent tasks associated with other MPJ jobs should be unaffected, even if they were initiated by the same daemon.

The paper design suggests that Jini is a natural foundation for meeting these requirements. The installation script can start a daemon on the local machine by registering a persistent activatable object with the `rmid` daemon. The MPJ daemons automatically advertise their presence through the Jini lookup services. The Jini paradigms of leasing and distributed events are used to detect failures and reclaim resources in the event of failure. These observations lead us to believe that an initial reference implementation of MPJ should probably use Jini technology [4, 16] to facilitate location of remote MPJ daemons and to provide a framework for the required fault-tolerance.

A possible architecture is sketched in Figure C.3. The base layer—process creation and monitoring—incorporates initial negotiation with the MPJ daemon, and low-level services provided by this daemon, including clean termination and routing of output streams (Figure

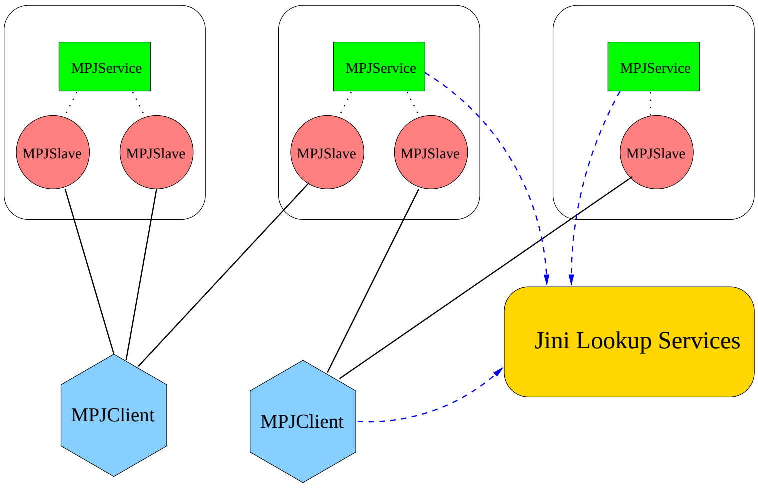

Figure C.4: Independent clients may find `MPJService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

C.4). The daemon invokes the `MPJSlave` class in a new JVM. `MPJSlave` is responsible for downloading the user's application and starting that application. It may also directly invoke routines to initialize the message-passing layer. Overall, what this bottom layer provides to the next layer is a reliable group of processes with user code installed. It may also provide some mechanisms—presumably RMI-based (we assume that the whole of the bottom layer is built on RMI)—for global synchronization and broadcasting simple information like server port numbers.

Higher layers use Java sockets directly for efficient communication. The first manages low-level socket connections, establishing all-to-all TCP socket connections between the hosts. The idea of an "MPJ device" layer is inspired by the abstract device interface of MPICH. A minimal API includes non-blocking standard-mode send and receive operations. Other point-to-point communication modes are implemented with reasonable efficiency on top of this minimal set. The device level itself is meant to be implemented on socket `send` and `recv` operations, using standard Java threads and synchronization methods to achieve its richer semantics. The next layer above this is base-level MPJ, which includes point-to-point communications, communicators, groups, datatypes and environmental management. On top of this are higher-level MPJ operations including the collective operations. We anticipate that much of this code can be implemented by fairly direct transcription of the `src` subdirectories in the MPICH release—the parts of the MPICH implementation above the abstract device level.

C.4.5 Results from prior NSF support

Work on this proposal is related to previous NSF awards including the research grant described below from the Division of Advanced Computational Infrastructure and Research. This ongoing project has transferred from Syracuse to Florida State University. Fox is PI.

Grant: 9872125, total award $346,827 over period 09/01/98-08/31/01, "Data Parallel SPMD Programming Models from Fortran to Java".

This involves senior personnel Fox and Carpenter, who with two students on the project have recently moved from Syracuse to Florida State. The project focuses on the use of Java for *data parallel* programming but the methods are applicable to other languages. Collaborator Professor Xiaoming Li from Peking University is investigating applications to traditional scientific languages—especially Fortran. We have published several papers on this subject where the details are described. The HPJava model is less ambitious than systems like High Performance Fortran (HPF) and aims to support an SPMD model intermediate between basic message passing (MPI) and HPF. One can incorporate pure MPI code but also array based computation with automatic decomposition with a user specified mapping in the spirit of HPF. An essential capability is unified support of successful data parallel libraries like ScaLAPACK, PetSC, Kelp, Global Array Toolkit, PARTI/CHAOS and Adlib. So far we have developed an operational HPJava translator and linked to Global Arrays and Adlib. As part of our collaboration we have prepared and given in China a tutorial on HPJava and related approaches (http://www.npac.syr.edu/projects/pcpc/HPJava/beijing.html). To support MPI work from within HPJava programs we developed the MPI Java binding *mpiJava* which is available for download from (http:/www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html), and will be form one of the foundations of the work described in the current proposal. It is a reference implementation used by the Java Grande Message Passing Working Group. Fox organized the Java Grande Forum (http://www.javagrande.org) to address all the issues connected with the use of Java in scientific computing and both the working groups and associated conferences (now sponsored by ACM) have been quite successful. HPJava ideas have greately benefitted from contacts in this arena. As well as publications given below, Sung Hoon Ko completed his Ph.D. in this area last year.

Select publications

B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li and Y. Wen "Towards a Java environment for SPMD programming", *4th International Europar Conference*, Springer, 1998.

G. Zhang, B. Carpenter, G. Fox, X. Li and Y. Wen, "The HPspmd Model and its Java Binding.", chapter in book, R. Buyya ed, *High Performance Cluster Computing*, Vol 2, Prentice Hall 1999.

M. Baker, B. Carpenter, G. Fox, S.H. Ko and S. Lim, "mpiJava: An Object-oriented Java Interface to MPI", *Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP '99*, San Juan, Puerto Rico, April 1999.

B. Carpenter, V. Getov, G. Judd, A. Skjellum and G. Fox "MPJ: MPI-like message passing for Java", Concurrency: Practice and Experience, 12(11), 2000.

D. Bibliography

[1] *ACM 1999 Java Grande Conference*. ACM Press, 1999. San Francisco, California, June 12-14.

[2] *ACM Java Grande 2000 Conference*. ACM Press, 2000. San Francisco, California, June 3-5.

[3] Erol Akarsu, Geoffrey Fox, Tomasz Haupt, Alexey Kalinichenko, Kang-Seok Kim, Praveen Sheethaalnath, and Choon-Han Youn. Using the Gateway system to provide desktop access to high performance computational resources. In *HPDC-8*. IEEE Press, August 1999.

[4] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[5] Mark Baker and Bryan Carpenter. MPJ: A proposed Java message-passing API and environment for high performance computing. In *International Workshop on Java for Parallel and Distributed Computing*, Cancun, Mexico, May 2000. To be presented.

[6] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[7] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with Java applets. *Concurrency: Practice and Experience*, 10(11-13), 1998.

[8] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *9th International Conference on Parallel and Distributed Computing Systems*, Dijon, France, September 1998.

[9] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[10] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11)—special issue on MPI-based parallel programming with Java, 2000.

[11] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998. http://www.javagrande.org/.

[12] Chi-Chao Chang and Thorsten von Eiken. Interfacing Java to the Virtual Interface Architecture. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[13] B.O. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu. Javelin: Internet-base parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 1997.

[14] Rossen Dimitrov and Anthony Skjellum. An efficient MPI implementation for Virtual Interface (VI) Architecture-enabled cluster computing. In *Third MPI Developer's and User's Conference*. MPI Software Technology, Inc., March 1999.

[15] Kivanc Dincer. *jmpi* and a performance instrumentation analysis and visualization tool for *jmpi*. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[16] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

[17] Adam J. Ferrari. JPVM: Network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11-13), 1998.

[18] Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling*, volume 9(6) of *Concurrency: Practice and Experience*, June 1997.

[19] Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling II*, volume 9(11) of *Concurrency: Practice and Experience*, November 1997.

[20] Geoffrey C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing*, volume 10(11/13) of *Concurrency: Practice and Experience*, 1998. Palo Alto, California, February 28 and March 1.

[21] The *Gateway* home page. http://www.gatewayportal.org.

[22] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in Java: Exploiting native libraries. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[23] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an evolutionary platform for infrastructural service. In *1999 Usenix Annual Technical Conference*, Monterey, June 1999.

[24] Global Grid Forum home page. http://www.gridforum.org.

[25] Grid Forum Jini activity home page. http://www.cs.cf.ac.uk/User/O.F.Rana/jini/.

[26] Hewlett Packard Company. *E-speak Architecture Specification*, September 1999. http://www.e-speak.hp.com/.

[27] HPJava project home page. http://aspen.csit.fsu.edu/pss/HPJava/.

[28] IMPI Steering Commitee. *IMPI—Interoperable Message-Passing Interface*, January 2000. http://impi.nist.gov/IMPI/.

[29] Java Grande Forum. Java Grande home page. http://www.javagrande.org/.

[30] The *java-mpi* mailing list. http://mailer.csit.fsu.edu/mailman/listinfo/java-mpi/.

[31] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed object group management architecture. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

[32] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[33] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. Technical Report TR-CSPE-07, University of Westminster, School of Computer Science, Harrow Campus, July 1997.

[34] J. E. Moreira. Closing the performance gap between Java and Fortran in technical computing. In *First UK Workshop on Java for High Performance Network Computing*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[35] J. E. Moreira, S.P. Midkiff, and M. Gupta. From flop to MegaFlops: Java for technical computing. In *Languages and Compilers for Parallel Computing*, volume 1656 of *Lecture Notes in Computer Science*. Springer, 1998.

[36] J. E. Moreira, S.P. Midkiff, M. Gupta, and R. Lawrence. High performance computing with the array package for Java: A case study using data mining. In *Supercomputing 99*, November 1999.

[37] mpiJava home page. http://aspen.csit.fsu.edu/pss/HPJava/mpiJava.html.

[38] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: Scalability issues in global computing. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[39] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[40] M. Philippsen and M. Zenger. JavaParty—transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[41] UC Berkeley Computer Science Division. Ninja project home page. http://ninja.cs.berkeley.edu/.

[42] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[43] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 1994. Reprinted in [4].

[44] Matt Welsh. Using Java to make servers scream. Invited talk at ACM 1999 Java Grande Conference, San Francisco, CA, June, 1999.

[45] Peng Wu, Sam Midkiff, Jose Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[46] Narendar Yalamanchilli and William Cohen. Communication performance of Java based Parallel Virtual Machines. *Concurrency: Practice and Experience*, 10(11-13), 1998.

E. Biographical Sketches

David Bryan Carpenter

Computational Science and Information Technology	Phone: (850) 644-0180
400 Dirac Science Library	Fax: (850) 644-0098
Florida State University	E-mail: dbc@csit.fsu.edu
Tallahassee, FL 32306-4120	URL: http://www.csit.fsu.edu/ dbc/

Education

B.Sc. in Physics, University of London (1976-1979).
Ph D. in Physics, University of London (1979-1983).

Professional Experience

2000–present Visiting Scholar/Scientist, CSIT, Florida State University **1996–2000** Research Scientist, NPAC, Syracuse University **1994–1995** Programmer, High Performance Computing Centre, Southampton, UK. **1989–93** Research Fellow in Department of Electronics and Computer Science and Department of Physics, Southampton, UK. **1989** Employed at "Transputer Technology Solutions", Southampton, UK. **1985–1988** Research Fellow in Department of Physics, Southampton, UK. **1985** Royal Society Overseas Fellowship at DESY, Hamburg. **1983–1984** Research Fellow in Theoretical Physics Department, Edinburgh University.

Select Publications:

[1] Bryan Carpenter, Vladimir Getov, Glen Judd, Anthony Skjellum and Geoffrey Fox "MPJ: MPI-like message passing for Java", *Concurrency: Practice and Experience*, Vol 12, num 11 (2000).

[2] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim "Object Serialization for Marshalling Data in a Java Interface to MPI", *ACM 1999 Java Grande Conference*, ACM Press 1999.

[3] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim, "mpiJava: An Object-oriented Java Interface to MPI", *Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP '99*, San Juan, Puerto Rico, April 1999.

[4] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xiaoming Li, Xinying Li and Yuhong Wen, "Towards a Java environment for SPMD programming", *4th International Europar Conference*, Springer, 1998.

[5] Guansong Zhang, Bryan Carpenter and Geoffrey Fox and Xinying Li and Yuhong Wen, "The HPspmd Model and its Java Binding.", chapter in book, R. Buyya ed, *High Performance Cluster Computing, Vol 2*, Prentice Hall 1999.

[6] G. Zhang, B. Carpenter, G. Fox, X. Li, X. Li and Y. Wen, "PCRC-based HPF Compilation", *10th Int'l Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, 1336, Springer 1997.

[7] B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw and X. Li, "Experiments with *HP Java*", *Concurrency: Practice and Experience*, Vol 9, num 9 (1997), p633.

[8] J. Merlin, B. Carpenter and Tony Hey, “shpf: a Subset High Performance Fortran compilation system”, *Fortran Journal*, (1996), pp 2-6.
[9] D.B. Carpenter and H. Glaser, “Some Lattice-Based Scientific Problems, expressed in Haskell”, *Journal of Functional Programming*, 6(3):419-443, May 1996.
[10] As. Abada, C.R. Allton, Ph. Boucard, D.B. Carpenter, M. Crisafulli, S. Güsken, P. Hernadez, V. Lubicz, G. Martinelli, O. Pène, C.T. Sachrajda, K. Schilling, G. Siegert and R. Sommer, “Semi-leptonic Decays of Heavy Flavours on a Fine-grained Lattice”, *Nuclear Physics* B416 (1994) p675.

Summary of Interests:

Carpenter has worked in parallel computing since 1985, when he was one of the first scientists to exploit the Inmos transputer for simulations of physical systems. Working at the University of Southampton, UK he became involved with design of libraries to support parallel computing. In a fruitful collaboration with John Merlin, then also at Southampton, he was codeveloper of the *shpf* system, an early implementation of subset HPF. This work produced the first implementation of the Adlib runtime library. After moving to Syracuse in 1996, Carpenter worked in the PCRC project. As part of this work the Adlib library was substantially redesigned to meet the requirements of a new HPF compiler, and reimplemented. Eventually it was delivered as the NPAC PCRC runtime kernel. Work on C++ interfaces to Adlib was a formative influence in the ongoing work on HPJava and HPspmd. Carpenter's recent research has focussed around the HPspmd project, which is developing translators and other support for library-based data-parallel programming, using Java and other languages. The mpiJava software for message-passing parallel programming was a spin-off from this project.

Synergistic Activities

Last year Carpenter gave a series of lectures on his research activities in China:
http://aspen.csit.fsu.edu/pss/HPJava/beijing.html
He also moved to Florida State University, where he has been teaching a graduate course on Internet technologies:
http://aspen.csit.fsu.edu/it1fall00

Collaborators

Mark Baker, University of Portsmouth; **Geoffrey Fox**, Florida State University; **Vladimir Getov**, University of Westminster; **Sung Hoon Ko**, Florida State University; **Han-Ku Lee**, Florida State University; **Xiaoming Li**, Beijing University; **Sang Boem Lim**, Florida State University; **Jarek Neiplocha**, Pacific NorthWest National Laboratory; **Guansong Zhang**, IBM.

Ph.D. Advisor

Elliot Leader

F. Summary Proposal Budget

Replace with NSF Form 1030s.

Budget Justification

Personnel

Salary costs are for 4 months per annum of the Principal Investigator, Carpenter, 2 Graduate Research Assistants, and 2 Undergraduate Research Students, including summer research.

Rates are based on actual rates where there are incumbents and approved rates for unfilled positions. Rates for senior staff and graduate research assistants are increased by 3% in year two and in year three.

There is no salary requested for Dr Fox.

Fringe Benefits

Fringe Benefits are 18.4% for senior staff and 0.6% for students.

Travel

Funds requested are for attending conferences to help keep abreast of the latest developments in the field. Estimates are based on previous conference attending costs.

Tuition

Tuition of $2000 per year for each graduate student is recorded in item G6.

Indirect costs

Florida State University's indirect cost is 46.5% $\times$ MTDC (total direct costs minus tuition).

G. Current and Pending Support

H. Facilities, Equipment, and Other Resources

H.1 Local Computer Facilities

The School of Computational Science and Information Technology (CSIT) at FSU has a substantial computer infrastructure, including 5 four-processor Origin 200s, a cluster of 30 Pentium Pros with 100 Mbit/sec ethernet network, a 16 processor IBM SP2, two 4 processor ES40 Compaq machines, and an assortment of older heterogeneous clusters. These will be supplemented by about 10 modern Sun servers from NPAC. Florida State University has committed to the allocation of 5 to 8 million dollars to the purchase of a massively parallel supercomputer that will achieve over a Teraflop of processing power over the next two years.

CSIT presently has 3.5 system administrators who are responsible for the daily upkeep of CSIT resources, software upgrades, troubleshooting, helping users, and performing backups.

H.2 Office

CSIT provides adequate office facilities for all participants.

H.3 Major Equipment

The computer systems described above constitute the major equipment for this effort.

H.4 Other Resources

CSIT provides basic secretarial services, duplicating facilities, and the like.

I. Letters of Cooperation and Support