A. Project Description

A.1 Motivation of the proposed work

Until quite recently it was generally assumed that computers would be used mainly for doing computations. Today an equally widespread idea is that a computer is primarily a device for *communication*. Of course there is no reason to think the value of the computer as, say, a simulation device for science, engineering, commerce, or even entertainment, has suddenly diminished. But lately growth in this arena has been eclipsed by the runaway growth in Internet applications.

The pivotal importance of communication is no surprise those in the high-performance computing community. For example, the 80s and 90s witnessed a remarkable diversity of hardware and software architectures for linking individual computational nodes through very high performance communication devices, specifically to support *parallel computation*. Perhaps one should not try to push the analogy between the Internet and the parallel computer too far—it may well be true that the overall communication bandwidth of a parallel computer is a better characterization of its effective power than the aggregate processor speed, but in this context communication is ultimately a means of computation, not an end in itself. In fact the specialized communication technologies that evolved for parallel computing do not seem to have had a major influence on the recent history of the Internet. Meanwhile it is abundantly clear that if parallel computing is to continue to thrive, and gain wider acceptance, it must adapt itself to the changeing environment of the Net. Parallel computing must find niches it can exploit in this larger context.

This proposal is especially concerned with enabling parallel computation, and the associated communications, in a world dominated by Internet technologies. We will not assume that parallel computation is necessarily done *across* the Internet, although of course this is one possibility that has been widely discussed, especially under the name of metacomputing. We *do* assume that the software and hardware technologies that will be readily available in the immediate future—the commodity technologies—will be fine-tuned for the Internet environment. On the hardware side these will include parallel—perhaps 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.

One of the most influential developments in parallel computing over the last decade was the publication in 1994 of the Message Passing Interface (MPI) standard [2]. The idea of an agreed, standard API for communication in parallel programs now seems obvious. In practise it was relatively slow in coming. As a result it benefitted from a great deal of accumulated experience from application developers using earlier, proprietary APIs for message passing. MPI supports the Single Program Multiple Data (SPMD) model of parallel computing. A group of processes cooperate by executing identical program images, manipulating distinct local data values. MPI provides reliable point-to-point communication with several blocking and non-blocking communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended

standard, MPI 2, allows for dynamic process creation and one-sided access to memory in remote processes.

The MPI standards specify language bindings for Fortran, C and C++. None of these languages is specifically adapted to the heterogeneous environment of the Internet, where downloadable and mobile code are norms, resources (including computational resources) may be discovered and lost spontaneously, unreliable networks and the associated need for fault tolerance are defining issues. In the proposed work we will be especially concerned to support use of network-oriented languages for high-performance computing. For now this means Java, although we cannot rule out the emergence of other comparably important languages in the course of the proposed work. Hence one immediate preoccupation is with refinement of MPI-like APIs for Java.

The validity of using Java for essentially "scientific" computing has been quite controversial in the past, but over the last few years the prospect has become increasingly realistic. Ongoing activities in the *Java Grande Forum*, complemented by work in academic and industrial sectors on optimizing compilers, JITs, language enhancements and libraries, have helped to close the credibility gap. By now it is widely assumed that future Java environments will meet the vital performance constraints needed to support 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. Scientific programmers will also reap the benefits.

As a part of the Java Grande process, a Message-Passing Working Group has been considering MPI-like APIs for Java over the last two years. Draft specifications and prototype implementations have been produced, but work remains to be done on the final specification, and robust reference implementations are urgently needed.

In the end, just providing MPI-like APIs for Java not enough. We must address the specific features of distributed computing. MPI was designed for a *reliable* environment. According to the philosophy of Sun's Jini project, for example, a defining characteristic of distributed computing versus concurrent programming (or in our case, versus parallel programming) is the presence of *partial failures*. We intend to exploit ideas similar to those developed in the Jini project. We anticipate such ideas will facilitate construction of scalable and fault-tolerant platforms for parallel computing, harvesting spontaneously discovered computational resources. The technology adds new job initiation capabilities, beyond those found in traditional high-performance computing environments.

An early deliverable will be a robust pure-Java implementation of the message-passing API, *MPJ*, specified by the Java Grande Message-Passing Working Group. This implementation will internally adopt Jini protocols and software components to manage distributed resources and detect failures, while providing a traditional SPMD programming environment at the user level. Later research will investigate how to bring low-level techniques for dealing with partial failures into the realm of control of the parallel programmer. An overall research goal is to combine successful technologies for High Performance computing with state-of-the-art object-oriented technologies for network programming.

A.2 Background

A.2.1 Why parallel computing and Java?

We can assume that the computers that host major Web sites will be either multiprocessors or clusters of workstations. Many are now, and this trend will presumably continue. Increasingly these servers are programmed in Java. The simple fact that the computers are parallel obviously does *not* imply that people will necessarily write parallel programs for them. Most of the time they clearly will not—individual nodes will simply handle independent transactions. But since these two technologies—Java and parallel computers—will coexist in Internet servers, this is a *[fertile]* place where we may expect to see roles for Java-based parallel computation emerging. In one scenario we can imagine that in the near future compute-intensive commercial services start to make their appearance on the Web. The exact nature of these services is still uncertain. Perhaps they will be data-mining queries using parallel algorithms, or financial analysis programs based on physical optimization processes, or perhaps people will simply want to play chess against a parallel computer.

[Need to make a case that multi-threaded Java programming is not enough.]

Relate the JGF-related efforts. Most important lesson is that (on reputable authority) Java can be made efficient.

Reiterate that many of the platforms most readily available *for parallel computing in the future will be machines deployed primarily as massively parallel Internet servers. These are likely to be volatile environments that demand the reliability provided by foundations like Java and Jini.*

A.2.2 Where will parallel programs live?

For the sake of a concrete example, consider the *Ninja* vision of the future of the Internet elaborated by researchers at UC Berkely. In their view a service—an Internet-accessible application or set of applications—should be *scalable* (able to support thousands of concurrent users), *fault-tolerant* (able to mask faults in the underlying server hardware), and highly-available. An important goal is to enable Internet-based services that are accessible globally from any user device, from PCs and workstations down to cellphones and Personal Digital Assistants. So one major concern is with mobile code for service deployment—specialized active proxies that migrate out across the Internet to position themselves close to the client devices, whatever they may be. But services must maintain persistent state, and the architects of Ninja conclude that distributed, wide-area management of this state is generally intractable. So, while "soft" state may be distributed across proxies, hard, persistent state is maintained in a carefully-controlled environment—the *Base*—engineered to provide high availability and scalability.

The Base is assumed to be a cluster of workstations with fast, local communication, a controlled environment, and a single administrative domain. The Base hosts the backend of services. The Base may be constructed from a heterogeneous set of nodes, and individual nodes may fail under unpredictable loads, and so on; but the cluster is strongly coupled and essentially trustworthy. It isn't necessarily homogeneous and it isn't completely reliable, so it is not quite a conventional parallel computer. However this is one environment where

we might expect parallel programs written in Java to thrive. Partly (as suggested *[above]*) this could happen because massively parallel programming will be needed to implement the individual Internet services of the future; partly it may be because the commodity parallel computers of the future will be designed primarily as Internet servers, because this is where the demand will be. Scientific programmers may exploit these resources to run their programs simply because they are readily available.

A completely different place where we might see early uptake Java-based parallel computing is in the classroom *[...]*

The last niche for parallel Java we will discuss here is in a sense the most obvious. 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. *[Cite some relevant papers from the JG workshops, if poss.]* Within the MPI community there is an ongoing effort to extend MPI specifications and implementations to support metacomputing, by allowing logical process groups (the basic cooperative units of parallel computation) to span geographically separated clusters and supercomputers. For example, an MPI interoperability standardization effort led by the National Institute of Standards and Technology [1] proposes a cross-implementation protocol for MPI that will enable heterogeneous computing. MPI implementations that support the *Interoperable MPI* (IMPI) protocol will allow parallel message passing computations to span systems, using the native vendor message passing library within each system.

Java-based metacomputing could exploit and supplement these ongoing MPI activities in various ways. Suppose, for example, that only a parallel *sub*-component of a distributed application is particularly suited to implementation in Java. If the Java part is programmed in the essentially MPI-like paradigm we espouse, it is an easy step to suppose that the Java component could interact with the non-Java, MPI-based part through the inherently parallel IMPI protocols (rather than, say, through a serial, performance-limiting CORBA or RMI gateway). In another scenario, a parallel program may be written uniformly in Java, using our MPI-like API. An optimized implementation of the communication class library is made available at each site that hosts distributed Java jobs. Internally these implementations can use a native, vendor-supplied MPI, with IMPI protocols between sites. The platform-independent, compiled byte-code for the user's parallel program is uploaded to host sites at run-time and dynamically linked to the local message-passing stubs.

The three application areas described here are suggestive only. This is essentially a research proposal, and we cannot predict with certainty how the results might be used.

A.2.3 Where does Jini fit in?

Short section of background on Jini—what it is; where it fits in the scheme of things; the role of it's lookup services; spontaneous federation of services; fault-tolerance and self-healing in communities of services; the Jini take on "distributed programming". What Jini is not*, really: not a global infrastructure—it isn't Ninja or E'speak. In the Ninja vision, for example, Jini is a technology that might fit comfortably (like MPI) within the "Base" (addresses initial federation of nodes, crashes of individual nodes, etc) or at the periphery,*

near the end-user devices (deals with attachment of devices to the Net).

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.

[Contrast Jini discovery with discovery in E'speak, Ninja, CORBA trading.]

[Talk about fault-tolerance features: leasing, distributed events, transactions.]

A.2.4 Bringing these things together

To support the parallel programmers of the future we will need Java implementations of lightweight messaging systems akin to MPI—the single most successful platform 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 we expect to host the Web services of the future. We need to address the fact that these environments will be highly dynamic. Any software must be adaptive: availability changes dramatically as workloads and network traffic fluctuates; nodes crash, new ones are attached and discovered on the fly, old ones are removed. Jini is the Java technology for dealing with these situations.

Reiterate: this proposal will address design issues for new message-passing APIs for Java and potentially other object-oriented network programming languages. It will address the principles of reliably implementating these APIs in the dynamic environments of Internet servers and networks. In particular implementations on top of Jini and emerging successors will be developed. The proposal will consider how MPI-like parallel programming APIs may best be enhanced to make Jini-like techniques for fault-tolerant programming available directly at the parallel application level.

Realistic assessment of current level of demand for MPI + Java (figures on mpiJava *downloads,* java-mpi *membership). Not yet large enough to support commercial exploitation, but enough to encourage research.*

A.3 Approach

Get back to concrete stuff. Sketch the proposed implementation of MPJ, using Jini.

Improving the API: talk about channels, MPI-RT.

Hand waving about how we will import Jini-like fault-tolerance into MPJ. (Don't really have anything concrete to say yet.)

Things to emphasize...

- *Java Grande*
- *Jini*
- *JG Message-passing Working group—MPJ*
- *mpiJava*
- Proposal for Jini-based implementation of MPJ

- *MPI-2 (dynamic process creation).*

Things to mention...

- *Java interfaces to VIA*
- *Ninja*
- *E-speak*
- *IMPI (Standard for interoperable MPI)*
- *MPI-RT (Skjellum's work)*

[Use the word "Landscape" somewhere.]

B. Bibliography

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

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