Java's Role in Distributed Collaboration

Marina Chen James Cowie

Cooperating Systems Corporation
Chestnut Hill, MA 02167

Abstract

In this white paper, we sketch some techniques for using Java to improve the state-of-the-art in scalable collaboration management for scientific and engineering applications. We argue that flexibility, cost containment, and support for large-scale collaboration must join raw performance as metrics for successful, scalable HPC software. Finally, we summarize our experience with large-scale Web-based collaboration, and describe some plans for future work.

1 Introduction

The last two years have witnessed a significant realignment in the market for hardware and software support for High Performance Computing (HPC), as distributed problem-solving environments have emerged to supplement more centralized approaches. In large part, this transition was an inevitable consequence of the demand for more mature HPC technologies to support commercial ventures.

As HPC left the laboratory in search of sustainable business applications, it shed some of the vestigial characteristics of a fetal software process (emphasis on weak prototypes, poor development tools, and special-purpose languages and programming models). By contrast, marketable HPC solves problems by maximizing the throughput of a distributed collaboration.

The new goal of scalable software is to contain the development cost of scientific and commercial collaborations as they grow in size, degree of distribution, and complexity. Containing the magnitude of these costs is not as important as the ability to predict their trend and factor HPC costs into long-range business plans.

From this perspective, "high performance" hardware and software can either reduce costs (by resolving computational bottlenecks in distributed collaborations) or simply increase them. Depending on how they are integrated, investment in special-purpose tools and technologies can restrict the flexibility of collaborative teams to adapt to new challenges.

1.1 Java and HPC: Opportunity

The Java standards (language definition, virtual machine, and standard library implementations) have arrived at the right time to help HPC practitioners bridge the collaboration gap. By themselves, none of the claimed advantages of Java are radical or unique. Object-orientation didn't help C++ conquer the distributed computing world, the capability to construct rapid prototypes didn't propel Smalltalk or Tcl/Tk or Perl 5 to the forefront, and neither strong typing, nor type-safe exceptions, nor garbage-collection could entice Standard ML out of the laboratory.

Java offers all these features, but adds standard packages for multithreading, network access, and user interface construction. Even more important, Java programs execute atop a standard virtual machine using bytecode interpretation, and can be guaranteed to be safely portable across a wide space of platforms without recompilation.

1.2 Java and HPC: Shortcomings

> *"Giving up on assembly language was the apple in our Garden of Eden: Languages whose use squanders machine cycles are sinful. The LISP machine now permits LISP programmers to abandon bra and fig-leaf."* (Epigrams in Programming, ACM SIGPLAN Sept 1982)

Performance puritans have been raising this alarm for many years. There's a reason, however, why hardware and software development and maintenance costs are on radically different curves. Humans simply aren't as good at managing generational transitions in software products as they are in hardware products, and as a result, rational developers should be willing to pay in short-term performance to improve the real long-term bottom line: predictability of project costs through improved interoperability, portability, and support for distribution and maintenance.

Why, then, hasn't the HPC community leapt aboard the Java bandwagon? Perhaps HPC practitioners are so intimately familiar with painful tradeoffs to achieve raw performance that they are naturally suspicious of solutions that give back any of those cycles. In Java's case, let's examine the reasons for their suspicions in more detail.

Performance Obstacles The obvious reason to adopt a "wait-and-see" approach to Java has been its lack of traditional scalability: less-than-stellar speed on a single processor, and no inherent support for either task-parallelism across multiple address spaces or data-parallelism. In terms of base performance, Java's virtual machine approach cannot compete head-to-head with either C++/f77/MPI (for high-performance scalar and NOW environments) or F95/HPF (for SMP and special-purpose MPP environments). Java suffers in the comparison, not only in terms of the raw speed of bytecode interpretation, but also because the virtual machine lives in a single address space, requiring additional overhead of a thick library layer (JIDL, HORB, etc.) to exchange data among multiple VMs.

In the short term, these are valid criticisms. But our concern lies in the broader definition of scalability: controlling acquisition and maintenance costs for scalable collaborative HPC efforts. If we can determine which performance criticisms are long-term, and which are short-term, we can determine whether Java's strengths in rapid prototyping and portability can outweigh temporary performance glitches. We shall differentiate among three classes of performance obstacles: engineering challenges for Java implementators, availability problems for Java developers (missing software components), and fundamental flaws in the design of the Java standards, in order to estimate Java's long-term impact on the costs of developing HPC software.

1.2.1 Engineering Challenges.

Reducing the overhead of bytecode interpretation is an obvious place to start improving Java's performance, and there's no shortage of commercial and academic efforts targeted at techniques for Just In Time (JIT) compilation. These techniques will be most successful for intensive regular iterations over statically allocated memory, and fairly successful at intraprocedural/peephole optimizations within general-purpose scalar code. Without application-specific compilation assistance, JIT techniques probably won't give much help on convoluted, irregular computations over adaptive data structures. (This roughly parallels the history of "traditional" compiler technology for traditional HPC languages such as HPF and HPC++.)

But it's safe to predict that Java's inner-loop performance over statically allocated memory (the same codes that HPC languages handle best) will soon fall within 2 or 3 times that of C. Other languages are now being compiled to Java VM bytecodes (for example, Ada 95); this extra attention will help further improve the performance of interpretation.

Java Memory Management. Java's dynamic memory management is another sore spot which is amenable to careful engineering. With time, the (currently significant) penalties for allocating, deallocating, and garbage-collecting at runtime will also help close the scalar performance gap with C. Java's emphasis on designing and reusing standard packages will help as well. For example, matrix packages typically perform intensive allocation and deallocation of temporary arrays. The designer of such a package may pay special attention to recycling, rather than abandoning and reallocating, large temporaries. Java's garbage collection allows the designer to concentrate on managing references to recyclable temporaries during their lifetimes — preventing them from escaping the package and ceasing to be safely recyclable — rather than the mechanics of allocation and deallocation (where most C and C++ codes go down in flames).

Java Threads and SMP. Along with scalar speedups, Java implementators are making good progress on two other fronts: the use of native OS threading packages to implement Java threads, and implementation of the Java VM atop shared memory multiprocessors. The primary application programming challenges for such systems have always involved trading off multithread safety against performance (by minimizing the size of the critical sections of code which must be locked to manipulate shared resources safely).

What might seem at first glance like a strike against Java (increased programming complexity, and performance hits for sloppy use of threading) actually applies more strongly to languages like C whose thread support is patched on as a runtime library afterthought. Since threading is central to Java, and standardized within the language and its VM, its performance impacts are subject to engineering improvement. Careful Java VM implementors can minimize the overhead of these locking constructs and Java's thread policy mechanisms. As a result, we expect that Java will emerge as one of the easiest, cheapest ways to take advantage of the parallelism of large SMP servers that will form the backbone of distributed HPC collaborations.

1.2.2 Missing Components.

As Java implementors deliver on these engineering challenges, they will pass on transparent speedups to the application developers. To lock in these performance improvements for endusers, these application developers must then supply some critical software links which are not provided by the Java VM or standard packages. Globally shared namespace support (providing Java clients with access to CORBA objects, and allowing Java servers to provide access to their own objects) is one such missing link; application-dependent communication protocols (for on-the-fly compression, synchronization, and recovery of multimedia data, for example) are another.

These are specific cases of a general problem which has not been adequately addressed by any proposed HPC language (up to and including Java). High performance in a distributed collaboration can rely as much on application-dependent strategies for communication, and on management of shared names and resources, as on the performance of each individual component. We will return to this problem in our description of a Java-based distributed HPC simulation environment.

1.2.3 Fundamental Difficulties.

Not all of Java's performance difficulties can be solved by hacking improvements into the VM implementation or extra packages. Some of Java's weaknesses are fundamental to its design, although none are insurmountable. Most visibly, Java lacks C++'s operator overloading and templates. Without operator overloading, it's difficult to implement syntactically clean support for data-parallel array constructs in Java, as the "obvious" array arithmetic operators cannot be rebound. Similarly, Java's lack of templates makes it harder to implement many fruitful task-parallel techniques based on iteration over collective container classes. (Approaches which cast all objects in a collective class to and from Java's generic **Object** class can restore some of this functionality, but incur significant overhead from two-way casting on every store to or retrieval from such a container.) Neither of these criticisms is a show-stopper, and solutions have been proposed[8]; they only make it marginally more difficult to construct data- and regular task-parallel codes in the same syntactic style as HPF or HPC++.

Other Java features may impose fundamental difficulties on some specific HPC application classes. Designers of realtime HPC codes will be frustrated by Java's reliance on garbage-collected memory, for example. The ability to omit C++'s class destructors, or C's `free` statement, is convenient (and prevents many bugs, thereby containing maintenance costs). However, garbage collection is costly enough (at least tens of milliseconds) that realtime HPC applications will need the option to manually deactivate asynchronous garbage collection, schedule it synchronously, and bound its timing in advance. Because of the relatively small size of the realtime HPC market, these goals may not be satisfied immediately.

Java also deliberately omits function pointers, which form the backbone of many prominent C and C++ idioms for low-latency active messaging, discrete event simulation, and callbacks within a framework.

Technically, of course, Java programmers can (and should) replace idioms that use function pointers with cleaner, safer object-oriented solutions with little loss in performance. Supporting function pointers in Java would have been practically impossible without opening the door to other pointer mischief, and wouldn't have been portable across platforms, or even between two Java VMs on the same platform. Still, it's a significant language restriction that singlehandedly wipes out some favorite tools from the HPC programmer's arsenal.

Summary With a few exceptions, then, we can assert that Java's performance problems vis-a-vis C and C++ are (1) short-term, easily fixed by the enormous commercial market which has sprung up on the strength of industry hype, and (2) more than compensated for by Java's strengths in rapid prototyping and cross-platform portability. In the following sections, we propose some specific sources of HPC development cost that Java can help control. We then describe some ongoing experience with Java as a basis for building distributed high performance collaborative tools.

2 Putting Java to Work in Distributed Collaboration

We have already discussed HPC's ongoing realignment from centralized solutions toward distributed environments. At the same time, the "space of expertise" among producers and consumers of computing technology has become intricately segmented. Hardware, operating systems, compilers, and runtime libraries used to cover known computational space. In recent years, entirely new categories of expertise have arisen: computational scientists, financial engineers, database technicians, and decision support systems specialists are just a few of the new microdomains represented in the classified ads.

A large part of the difficulty in constructing collaborative high performance computing applications lies in resolving the "impedance mismatches" between these different groups of experts. By default, each team brings a different set of skills to a collaboration — chained to a different set of prejudices about how software should be built, and swearing allegance to a different set of underlying design assumptions and tools.

Collaborative software development cost overruns arise because of the complexity of resolving these differences to reach common goals. This resolution can be visualized as taking place in three continuous collaborative processes: (1) interface prototyping, (2) dataflow simulation and task mapping, and (3) software integration. These processes are interdependent and continuous; breakdowns in any one can contribute to cost overruns in the others. We are studying how Java-based development environments, in particular, can help scientific and engineering application developers contain costs and promote scalability on all three fronts.

Rapid prototyping. Complex collaborative efforts require quick proof-of-concept demonstrations and validations of basic assumptions about the shared design. Building scientific and engineering codes, like other collaborative processes, relies in large part on the ability to construct mockups of user interfaces and the interactions between components long before the details of their implementation have been filled in. Java's portable graphics support allows a prototype graphical user interface to execute in all participants' home environments from day one.

Such a GUI might model each top-level task (or "role") within a collaboration as an onscreen entity, supporting a specified set of symbolic actions or operations (control, visualization, input consumed, output produced). Each team would then independently create Java code that simulates, schematically represents, or actually implements the functionality of each of their roles in the collaboration.

All other teams can load this code over the network, linking it with the GUI classes to create a complete application at each site, or each team can publish their code as a local service running atop appropriate local resources. As the fidelity of the simulation provided by each component increases, what started as a simple graphical mockup becomes a useful tool for participants to monitor their own development progress relative to others, and to identify and resolve specific problems with component interoperability. The net effect is that of a distributed visual editor for building collaborations, in which all participants can view the progress of the work as a whole, while providing incrementally improved content for their own components.

Simulation of adaptive dataflow. By providing, in essence, a coarse-grained simulation of distributed application dataflow, this prototyping process helps to establish lower bounds on the required bandwidth and computational throughput of each participant in the distributed application. It also helps map application tasks appropriately onto heterogeneous resources and collaborators' specific expertise. Java can help promote top-down design of this process, starting with the partition of the collaboration into tasks, specification of the public interfaces of solutions for those tasks, and initial binding of those interfaces to a publication "home" at one of the collaborative institutions. Those tasks can then be partitioned into subtasks and mapped to specific Java packages, and local resources identified to serve as hosts for the implementations of those packages.

Integration of software. As collaboration partners use their specific skills to solve their subset of the collective tasks, they then need to publish code and documentation for those solutions in a standard format. Web technologies (HTML and its standard browsers) make hyperlinked code and documentation available to all participants; Java completes the picture by allowing publication of packages of executable code along with their APIs. The test and validation role forms another primary task within the collaboration; success or failure are measured by the ability of the integrated collection of components to respond appropriately to a real or synthetic workload supplied by the testers.

At all times, control over source and object code remains with the original developers, because their remote partners can load the Java code over the network each time it is used. This allows implementation improvements to be transparently propagated as they are filled in by the primary developers of the package.

3 Initial Experiences with Distributed Collaboration

To gain some perspective on why these elaborate design techniques are warranted, and to understand why Java in particular seems to offer great promise as a portable tool for implementing them, it always helps to look backward.

We have been studying the emergent problems of scalable distributed collaboration in earnest for several years. Over that period, we moved through a sequence of prototype projects and thought experiments (WebHPL in early 1995, giving rise to WebWork/WWVM in late 1995 and WebFlow in 1996, all in partnership with the Northeast Parallel Architecture Center at Syracuse University). We started by focusing on a model (WebHPL) in which standard HPC codes were simply wrapped in a veneer of Web technology (at the time, primitive Perl 4 scripts via the CGI interface of HTTP-based Web servers).

This didn't provide the necessary control over shared namespace management, or flow control for links between components. It sidestepped collaboration design issues, and didn't really answer questions about security and resource management; it simply assumed that coarse-grain pipelining of data between monolithic Fortran kernels would suffice. For some sets of assumptions (high performance Intranets, and a close-knit workgroup composed of people with similar expertise), we expect that this approach is still valid.

Later projects (WebWork, segueing into WebFlow) shifted emphasis away from the HPC kernels *per se*, toward design of the collaborative infrastructure — the "glue" that supports shared namespaces among HPC applications, and provides dataflow mechanisms for timing progress among coupled components. WebWork specified a model of computation in which a pool of HPC codes, wrapped as network services, acted as each others' clients in a *compute web*[6]. The Responsive Web Computing project began to explore ways in which Web-integrated applications could provide and receive quality-of-service guarantees, by implementing appropriate realtime resource management techniques and protocols[1]. Meanwhile, the WebFlow design added explicit dataflow to manage geographically distributed data-intensive applications.

3.1 RSA130: Factoring By Web

In 1995 we constructed FAFNER, a proof-of-concept prototype of a distributed collaborative environment for factoring RSA-130, the 130-digit composite challenge number[3, 4]. FAFNER's job was automation and coordination of the flow of tasks and subsolutions within a globally distributed network of anonymous sieving clients. The NFS sieving code itself was crafted at Bellcore[7], starting with a pair of irreducible polynomials derived during extensive sieving experiments at the University of Saarland, and ultimately feeding a large sparse matrix solver running on a Cray-C90 at the SARA Computer Center in Amsterdam.

The project attracted the interest of browsers from over 500 different Internet hosts; 20 percent of those hosts stayed to participate in the sieving stage. These ranged from SLIP-connected home computers to high-performance corporate workstation clusters, and literally covered the globe. Browsers from 28 countries, from AT (Austria) to ZA (South Africa), left their prints on the project. In all, FAFNER clients donated over 17% of the cycles used to crack RSA130, whose prime factors were finally revealed to the world in April 1996[5].

The FAFNER Web-factoring package received the "most geographically distributed/most heterogeneous" award in the High Performance Computing Challenge at Supercomputing '95[2]. Despite all that, it was a fairly primitive aggregation of Perl 5 and C code, with limited support for reconfiguring the computation as it proceeded, and was very much a special-purpose problem solving environment for performing the Number Field Sieve.

Clients in the factoring-by-Web collaboration could get project documentation and register themselves anonymously through a large collection of Web pages generated by CGI scripts. These pages generated a mix of canned and custom Perl and C code for each user to download and build, drawn from multiple sources (some developed and resident at CSC, the rest at Bellcore). Clients could elect to become FAFNER servers (installing a personalized package of Perl code on their own Web server) or just act as sieving clients. Sieving contributors were required to build and execute the (very large) C codes we provided them, at their own risk, using their own C compiler. Every time a bug was found in the client code, we were forced to reinstall it as the standard version on the FTP server, and contact each sieving client to alert them to the availability of the "improved" code.

These factors combined to convince us that the state of the art in Web technology circa 1995 (e.g., Perl CGI scripts for server-side code, and downloadable C source code for client-side code) was woefully insufficient. We were exposed to serious security violations due to latent bugs in the server-side Perl scripts we executed on behalf of clients. We had no real control over client-side code once we had shipped it and clients had put it to work. We had no way to revise the structure of the collaboration (for example, by introducing new roles for third-party archivists, or offering multiple variants of sieving code) without shutting down the entire project, redesigning and rebuilding, and starting back up. Worst of all, we required clients to take unnecessary security risks to perform what was, after all, an act of altruism (donating cycles to a worthy cause).

4 Future Directions

Just as we were closing down the RSA130 effort, Java began to emerge as a potential solution to many of these problems. If we could provide at least the network client code in Java, sieving volunteers would be able to restrict the applet's access to their local filespace. Some parts of the code (the large numeric codes which carry out the sieving operation) would still be distributed as C code, more because of the impossibility of rewriting than for performance's sake. Still, any step which reduces the size and complexity of the code that users have to trust is a major improvement.

4.1 Java-based factoring collaboration.

We have now started designing FAFNER II to support a future factoring project which will tackle a 512-bit encryption modulus. The Web server that provides documentation and registration will be written in Java, as will the client graphical interfaces, and the wrapper that makes the legacy numeric code (itself still in C) network-capable and more fault-tolerant. As client sieving applets periodically consult the central server for new tasks to perform, they can also process notifications of software updates, dynamically downloading authenticated revisions to their own code.

Incremental Protocol Extension. We have also constructed a Java-based Web server that supports incremental protocol extension via dynamic class loading; this allows the Web server to "branch out" to temporarily offer new services on an allocated port drawn from an auxilliary pool. For example, after initial negotiation via HTTP, a network client and task service may elect to communicate on a new port using a private protocol optimized for their application-specific content. The Web server manages dynamic loading of the server-side class files that implement the service, manages the pool of extra server ports, and allows the server administrator to monitor resources (CPU utilization and network bandwidth) expended by each secondary service.

4.2 Distributed Simulation of Telecommunications Networks

We have also begun to study how Java-based collaboration tools can be applied to parallel discrete-event simulation (PDES). Even compared to more traditional HPC software domains, PDES systems remain heavily dependent on application- and platform-specific techniques for performance improvements[9]. Again, a conflict of emphasis arises between two definitions of scalability: industry research pursues cross-platform integration of multiple scalar simulators, while academic research focuses on speedup within a single parallel simulator.

To build a scalable telecommunications simulation framework, we'd like to determine ways to map sub-simulations (that is, simulations of subnetworks, or even individual pieces of telecommunications equipment) onto distributed resources (individual simulators), and use a Web-based dataflow model to propagate events between them. Experience has shown that it can be extremely difficult to get good performance out of such a dataflow scheme, because of the requirement that subsimulations never process events out of time-order. This can result in very fine-grained synchronization requirements to prevent causality violations, which kill any performance improvements won through distribution.

Java's role in PDES. It seems clear that the key is not to use Java to construct generic parallel simulation engines, but to provide more sophisticated tools for describing and partitioning the simulated telecommunications architecture, and map its subsimulations onto simulating resources in such a way that long-distance synchronizations are minimized. These tools will need to capture a wealth of application-specific information about the event patterns emitted by each component within a simulated architecture, so that distributed components can make proper lookahead promises to each other. Unlike C, or C++, or HPF, Java actually provides the proper combination of object-oriented features, dynamic class loading, and cross-platform portability via bytecode interpretation required for this task.

A challenging distributed simulation might use Java (or C codes wrapped as native methods in Java) to simulate a global-scale telecommunications network, combining wireless (cellular), wireline (ATM), and low-earth-orbit satellite links, driven by a call placement model synthesized from a profile of civilian and military traffic. The domain decomposition and mapping process will take advantage of structural information about this simulated domain at all levels.

At the network level, we can block-partition the physical environment to which wireless callers are aligned (geographic distribution, or radio spectrum utilization) to attempt to block subsimulations with interdependencies onto the same simulating resource. At the network architecture level, we can map subnetworks with the highest degree of internal connectivity to the same simulating resource, and abstract discrete traffic to a flow-based model between subsimulations. At the functional level, we can further decompose the simulated civilian or military activity which provides the telecommunications network stress (Mother's Day, or a missile launch).

4.3 Three Concurrent Collaborative Processes for PDES

This collaborative process brings together telecommunications domain specialists, scalar and parallel network simulation experts, user interface designers, and application specialists. The challenges can be analyzed in terms of our three concurrent interdependent tasks described in section 2.

1. Interface prototyping. A complex simulation can be viewed from several perspectives, each corresponding to a particular level of abstraction. One view shows the collection of application functions (military or civilian) that generate telecommunications traffic, as they place and receive calls according to their program of simulated activity. Another view details the events generated by call placement and routing, as they flow through the simulated network of networks. A third view details the inner state of each subsimulation in more detail, providing more information about its accumulated state, number of calls dropped or rejected, and resource utilization. We are using Java's standard graphics, threading, and networking packages to construct multiperspective visualization tools for each of these views, plus a standard "simulation browser" framework to link them together and provide realtime updates to all participants.

2. Dataflow simulation and task mapping. In addition to the direct views of simulated activity, other views provide "metainformation" about the availability of software components from each team for each region of the simulation, and about the health of each simulation component and its resource utilization. Components which have no implementation at all will simply "swallow" event dataflow; these will quickly be replaced by schematic flow-based simulations, to give a first approximation of the behavior of each region of the global network, and then by more detailed event-based simulations which will gradually improve in fidelity. This incremental process allows collaboration partners to prioritize components for code development, and identify bottlenecks in the mapping of components to simulating resources. Java's OO design and dynamic class loading allow us to start with skeletal implementations of this dataflow web, and fill in details and improve fidelity over time. The goal is to minimize idle time — either on the part of developers waiting for each other, or on the part of invididual simulation components waiting for events from their overloaded neighbors.

3. Software integration. The final goal of a Java framework for distributed simulation is to provide the pluggable infrastructure into which these simulation kernels can be embedded, and help those kernels negotiate ways to exchange dataflow information and thus minimize the time spent in fine-grained synchronization. We also hope to use Java to integrate two pieces traditionally missing from telecommunications simulation: network management and visualization capabilities. Finally, the ability to dynamically load and instantiate new Java classes gives us some leverage to support on-the-fly application-specific synchronization protocols between subsimulations. The recipient of events can ask the provider of those events to install an *event notification filter* on its behalf, so that it receives more precise early warning about the lower bound on the next arrival time of a relevant event. This improved guarantee then helps to maxmize the time spent on internal simulation without fear of an out-of-order event arriving.

4.4 Conclusion

Raw performance, once the holy grail of scientific computing, actually plays a very small part in achieving the goals of large-scale scientific collaboration. Designers of distributed collaborations should instead optimize the throughput of the entire organizational process in order to minimize the variability of software development costs.

Java doesn't offer any truly new features, and has some short-term performance drawbacks. Nonetheless, HPC developers should start to take advantage of its nearly universal availability, support for cross-platform portability, and combination of object-orientation with standard packages for graphics, multithreading, and networking.

Our experience to date suggests that initial Web technologies by themselves (e.g., HTML, HTTP, CGI) were insufficient to support the software engineering requirements of significant distributed scientific collaborations. Java arrived just in time to fill in the remaining gaps: tools to construct user interface prototypes, interactive dataflow analysis and task mapping, and integration of software from diverse teams of experts.

References

[1] A. Bestavros, M. Chen, M. Crovella, A. Heddaya, S. Sclaroff, and J. Cowie. Responsive Web Computing: resource management, protocol techniques, and applications. Technical Report BU TR96-008, Boston University and Cooperating Systems Corporation, 1996.

[2] S. Bhatt, M. Chen, J. Cowie, G. Fox, W. Furmanski, and A. Lenstra. Factoring on the World-Wide Computer (WWC). *Supercomputing '95 3rd Annual High-Performance Computing Challenge*, December 1995.

[3] J. Cowie. FAFNER: Web server support for factoring RSA130. Technical Report CSC TR-049601, Cooperating Systems Corporation, April 1996.

[4] J. Cowie. GNFSD: the general number field sieve daemon. Technical Report CSC TR-049602, Cooperating Systems Corporation, April 1996.

[5] J. Cowie, B. Dodson, M. Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery, and J. Zayer. A world wide number field sieve factoring record: On to 512 bits. *ASIACRYPT '96*, November 1996.

[6] G.C. Fox, W. Furmanski, M. Chen, C. Rebbi, and J. Cowie. Webwork: Integrated programming environment tools for national and grand challenges. Technical Report NPAC SCCS-715, Northeast Parallel Architecture Center, June 1995.

[7] A.K. Lenstra and H.W. Lenstra Jr. The development of the number field sieve. In *Lecture Notes in Math*, volume 1554. Springer-Verlag, Berlin, 1993.

[8] A. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[9] D. M. Nicol and P. Heidelberger. Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.