Dagstuhl Seminar

on

High Performance Computing and Java

Organized by

Susan Flynn-Hummel	(IBM Thomas J. Watson Research Center)
Vladimir Getov	(University of Westminster)
François Irigoin	(École de Mines de Paris)
Christian Lengauer	(Universität Passau)

Schloß Dagstuhl, 28. – 31.08.2000

Contents

1 Preface

The object-oriented programming language Java is being viewed as the modern alternative for C++ and has rapidly captured people's attention, largely because of its features for interactive and Internet programming. One advantage of Java over C++ is that it includes mechanisms for parallelism and coordination, which makes it a natural language for distributed computing.
Java is still commonly perceived as execution-inefficient. It is not being realized widely enough that this inefficiency is a property of the language *implementation*, not of the language *per se*. Initial implementations interpreted relatively unoptimized bytecode via a relatively unsophisticated Java Virtual Machine (JVM). Recent developments in compilation technology –for instance, increased static analysis and just-in-time (JIT) compilation– and extensions of the JVM with instruction-level optimizations have done away with many of the initial sources of Java's execution inefficiency. In certain applications, Java is these days competitive with C or C++, but a lot safer to execute and easier to program.
In the face of these developments, interest in Java has grown also in the field of high performance computing. Obviously, high performance applications place extreme demands on execution efficiency.

The first question which is immediately being asked when Java is being proposed as a vehicle for high performance computing is: why?
The most frequent answer is: to get access to the incredible amount of resources that goes into program development in Java (this includes class libraries, compilation technology, programmer education, etc.). Without any other answer, one would mimic high performance programming in Fortran, C or C++ when programming in Java, and one would expect the Java compiler to produce essentially the target code which a Fortran, C or C++ compiler would produce. This approach is actually being pursued in some research and is legitimate as an immediate aid to a new generation of programmers, who make Java their preferred choice.
An answer that is much more ambitious to implement but that does Java more justice is: to develop a new technology in high performance computing that allows us to do things that cannot easily be done with Fortran, C or C++. For example, one distinct advantage of Java over Fortran, C, and C++ is its support of secure, portable and dynamic target code. This makes it a promising vehicle for irregular applications, such as those with thread-

based parallelism, or for run-time compilation and optimization – if Java can deliver the expected execution efficiency.

So, the next question is: can Java deliver? The immediate answer seems to be: in its present form to a limited extent. Many obstacles to high performance are due to current implementations of the language. However, some are part of its design and others are imposed by the unusually restrictive semantics of Java.
How can Java be made more suitable for high performance computing? Should high performance applications be adapted to the present limitations of the language, or should the language be extended (moderately) to become better suited? If the former, how can we tune Java implementations to exploit the performance potential of the language to the fullest? If the latter, what form should these extensions take? Is it permitted to add/modify language features or can/should one stick to class library extensions?
These and similar questions were explored by 35 researchers at Schloß Dagstuhl. The abstracts in this report summarize the presentations. Three full papers of participants in the October 2001 issue of the *Communications of the ACM* provide an overview of the current state of the art in getting high performance from Java.

The outcome of the seminar has been: Java has a lot more potential for high-performance computing than is commonly believed. Yes, if implemented naïvely, there are problems with scientific programming – no, they are largely not inherent but can be overcome. Yes, exploiting the full potential of Java's parallelism requires efficient virtual machines and high-performance networking software – which is in the process of being developed.

We are grateful to the IBM Thomas J. Watson Research Center for financial support for the seminar.

Susan Flynn-Hummel, Vladimir Getov, François Irigoin, Christian Lengauer

2 Abstracts

Java Communications for Grande Applications

Vladimir Getov

University of Westminster, UK

Java is receiving increasing attention as the most popular platform for distributed computing. However, it is still subject to significant performance drawbacks and lack of support for parallel message-passing computing. In this talk, we present part of the results and proposed solutions to these problems. In particular, we report about the current status of the organized collaborations within the Java Grande Forum in the area of Message Passing for Java (MPJ). An outline of the current MPJ specification is given along with a discussion of several open issues and performance results on different platforms - Linux cluster, IBM SP-2, and Sun E4000. These "proof-of-concept" Java and message-passing results are quite encouraging for future developments and efforts in this area. We also demonstrate that a much faster drop-in RMI and an efficient serialization can be designed and implemented in pure Java. Our benchmark results show that our better serialization and improved RMI save more than 50% of the runtime in comparison to the standard implementations available at the moment. Our results confirm that fast parallel and distributed computing in Java is indeed possible.

Invoke Interface Bytecode

Bowen Alpern

IBM Thomas J. Watson Research Center, USA

Like Java itself, Java interfaces have a reputation for inefficiency. This talk reports on a small case study to see if Java features can be supported with low run time overhead. The results are mixed. An efficient technique for interface-method dispatch is presented. Unfortunately, since Java interfaces

are not loaded with the classes that purport to implement them, dynamic type checking is required before each interface-method call. Thus, while Java-style interfaces are not inherently inefficient, Java's scheme for loading them at the last possible moment may be.

Distributed Execution of Java Bytecode Implementing Java Consistency Using a Generic, Multithreaded DSM Runtime System

Luc Bougé

ENS Lyon, France

(joint work with Gabriel Antoniu, Philip Hatcher, Mark McBeth, Keith McGuigan, Raymond Namyst)

This talk describes the implementation of Hyperion, an environment for executing pure Java programs on clusters of computers. To provide high performance, the environment compiles the original Java bytecode to native code and supports the concurrent execution of Java threads on multiple nodes of a cluster. The implementation uses the PM2 distributed, multithreaded runtime system. PM2 provides lightweight threads and efficient inter-node communication. It also includes a generic, distributed shared memory layer (DSM-PM2) which allows the efficient and flexible implementation of the Java memory consistency model. We provide preliminary performance figures for our implementation of Hyperion/PM2 on clusters of Linux machines connected by SCI and Myrinet.

Cluster Computing with Java Threads

Philip J. Hatcher

University of New Hampshire at Durham, USA

Our work combines Java compilation to native code with a run-time library that executes Java threads in a distributed-memory environment. This allows a Java programmer to view a cluster of processors as executing a *single* Java virtual machine. The separate processors are simply resources for executing Java threads with true concurrency and the run-time system provides the illusion of a shared memory on top of the private memories of the processors. The environment we present is available on top of several UNIX systems and can use a large variety of network protocols thanks to the high portability of its run-time system. To evaluate our approach, we compare serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. All measurements have been carried out on two platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

Java-Based Code Mobility

Omer F. Rana

University of Wales, UK

Two themes are investigated, the first explores the importance of mobile computing with reference to "high performance computing". Generally, the scientific computing community has been viewed as a synonym for high performance computing. However, we suggest that scientific computing should be considered as more than improvements in performance, and should also take into consideration ease of use, and large scale, high throughout applications. The emergence of mobile and nomadic computing requires interaction

between embedded and mobile devices connected over low bandwidth links, and with the capability to support millions of devices and users. In such systems, mobility can support (1) User Virtual Environments, whereby mobile users are given a uniform view of their prefered working environment, as they migrate from one domain to another, and (2) Mobile Virtual Terminals, where devices can move and connect to different points of attachment, and continue to access and receive the same services. Mobile computing necessites the provisioning of geographically transparent services, and requires novel ways to manage devices and resources to provide such services.
The second theme suggests that Java provides the best implementation medium for achieving mobile software services, and various aspects of Java for achieving mobile services are explored. The Java class loading mechanism is first explored, followed by the Jini API for uploading dynamic code to implement software services. A systems architecture based on Java services and Jini is defined, and extended with the use of code distribution based on the mobile-agent paradigm. We suggest that Java based implementation gives us a good abstraction for implementing mobile services.

Manta: Efficient Communication for Java

Thilo Kielmann

Free University of Amsterdam, The Netherlands

Manta is a high-performance Java platform for parallel programming on distributed-memory systems. Manta is based on a static compiler, translating Java source code directly into executable programs. Here, the focus is on Manta's communication mechanisms that aim to achieve high efficiency while fitting into Java's object-oriented model.
In Java, objects communicate by invoking methods on each other. For distributed-memory platforms, Java provides the Remote Method Invocation mechanism (RMI) accordingly. Unfortunately, Java RMI has been designed for Internet-based client-server applications; slow execution is the well-known consequence. For Manta, we re-implemented the RMI mechanism from scratch. Manta RMI relies on compiler-generated serialization routines, on an efficient (compact) RMI protocol, and on fast user-level communication. On our platform (using 200MHz Pentium Pro's and Myrinet),

a parameterless RMI completes in just 37 microseconds, only 3 microseconds slower than the underlying, C-based RPC call. Compared to the JDK implementation of RMI, Manta RMI is at least a factor of 10 faster. For interoperability with JVM's, Manta also implements the (slow) Sun protocol, including dynamic loading and compilation of byte codes.
For parallel applications, remote objects are shared between threads in different address spaces. Unfortunately, such shared objects constitute performance bottlenecks when accessed frequently within a parallel application. For shared objects with a high read/write ratio, replication can significantly improve performance. Manta implements Replicated Method Invocation (RepMI) for achieving this goal. RepMI resembles RMI as much as possible: replicated objects are identified by implementing special interfaces which are recognized by the Manta compiler. Write methods are simultaneously shipped to all replicas, much like a RMI is shipped to the single, remote object. For replica consistency, write operations are executed in a globally ordered sequence. On our platform, a read method (on the local object replica) completes in less than one microsecond, while writing 64 replicas takes just 120 microseconds.
RepMI provides efficient access to shared objects with a simple and object-oriented programming interface. For some uses of shared objects, however, its consistency model is too strict, thus degrading performance. Examples of such uses are collective operations among all parallel threads of an application, like reductions or all-to-all exchanges. The integration of such collectively used objects into Manta is subject to ongoing work.

Java Numerics: Prospects and Technology

Ronald F. Boisvert

National Institute of Standards and Technology, U.S.A.

In this talk, I survey the activities of the Numerics Working Group of the Java Grande Forum. The Working Group is an open association of researchers from industry, academia, and government seeking to improve the Java language and its environment for numeric-intensive computing. From a basic set of requirements for numerical computing, the group has identified several critical areas where improvements in Java are necessary: floating-point

performance, complex arithmetic, multidimensional arrays, elementary functions, access to IEEE arithmetic features, and a lack of standardized class libraries for core numerical tasks. I discuss the current progress and future plans in each of these areas.

Performance Evaluation of Java for Numerical Computing

Roldan Pozo

National Institute of Standards and Technology, U.S.A.

Among the many features of Java as program development platform, one of the most commonly cited shortcomings is its performance, particularly for computational-intensive codes. In this presentation we will take a close look at how Java works and how it can be made to execute faster for scientific simulations and modeling. We will examine optimization strategies and bytecode transformations that generate 100% pure Java with 2-10x performance improvement, with speeds competitive with optimized C/C++ and Fortran.

High-Performance Java Codes for Computational Fluid Dynamics and Sparse Matrix Computations

Christopher Riley

University of North Carolina at Chapel Hill, U.S.A.

Are Java's object-oriented features compatible with high performance computing? To help answer this question, Java implementations of a Computational Fluid Dynamic (CFD) code and a sparse matrix package have been developed as part of the HARPOON project at UNC-CH. The CFD code is a finite-volume, structured grid solver used to model high-speed, high-temperature flows. Originally written in Fortran, the current Java version

employs an object-oriented design that uses abstract classes to model the various chemistry options, boundary conditions, and grid-related concepts. The Java version of the sparse matrix package is based on the C implementation of the sparse matrix routines in MATLAB.
Test cases to benchmark performance were run on three platforms (Sun, Intel, SGI) using different versions of the JVM (Java 1.1.7, Java 1.2, Java 1.3). The measured execution times of the Java sparse matrix routines are less than 2 times slower than the MATLAB version. The Java CFD code runs between 2 and 10 times slower than the original Fortran version depending on the platform. The best relative performance was measured on an Intel Pentium II. Because these Java codes are derived from realistic applications, they can make excellent benchmarks.

Problem-Solving Environments for Scientific Computing

David Walker

Oak Ridge National Laboratory, U.S.A.

(presented by Omer F. Rana)

Problem Solving Environments (PSEs) provide an automated compiling, composition and execution environment for scientific application. PCEs are, by definition, problem specific. Each PSE contains software components for a specific application, maintained in a repository. Each component contains XML interface, which contains a definition of the I/O) parts, execution constraints and links to executable codes, etc. A scientist can construct apps by connecting components (which can wrap sequential or parallel codes) into a data flow graph. This is then passed to a resource management program execution.
Although a PSE is problem specific the infrastructure for a PSE is not. We describe PSE infrastructure, and discuss common theme & lessons learned.

On Structuring Scientific Java Systems

Judith Bishop

University of Pretoria, South Africa

While high performance of computers is essential, high performance of people is often forgotten in the scientific milieu. Good programming practices such as separation of concerns, re-use, maintainability and correct concurrrent execution need attention alongside performance. Often projects that switch from Fortran or C to Java do not take the step of ratcheting up on the new programming paradigm Java offers. The same is true for many courses taught to engineering and science students. This talk addresses the dual issues of how Java can best supply everything that the older languages do, and then what it can meaningfully give in added value, especially in the parallel and scientific area. Experience with developing solutions in Java to some fifty typical numerical problems has led to a coherent object-oriented approach and a couple of essential support classes for teaching and production work.
The novel Java features that can best be employed on scientific programming are objects, packages (libraries), abstract classes, interfaces, serialization, threads and socket connections (for messaging). A typical solution to a numerical proble is structured as a class with two objects. The first instantiates a Worker class which inherits from an abstract library class supplying a solver method and which provides the concrete equations to work on. The second object instantiates a Datahandler class into which all input-output is relegated. A Display class and a Graph class augment the novice user's experience of input-output in Java. If more equations are to be solved, the library class is re-inherited with the new information.
For trainee programmers, early experience with concurrency, distribution and parallelism is essential. An approach which concentrates on threads that are fired up on different computers and communicate via input-output statements on socket connections is both simple and safe. At a later stage, the programmers can move on to RMI, Corba or MPJ. Our figures show that both RMI and CORBA have reasonable performance for distributed programs, both in Java and with interfaces to other languages.
Further information on the approach and all the examples can be found on http://www.cs.up.ac.za/javagently under the "Java for Engineers and Scientists" site, and papers related to the approach and to the figures for RMI and CORBA are on http://www.cs.up.ac.za/~jbishop under publications.

Java Programming for Numerical High Performance Computing: Language, Libraries, and Compiler Issues

Jose E. Moreira

IBM Thomas J. Watson Research Center, U.S.A.

The performance of Java programs has improved significantly since the introduction of the language a few years ago. However, some characteristics of the language still make optimization a difficult chore for Java compilers and virtual machines. In particular, the structure of multidimensional arrays, which in Java are organized as arrays of arrays, make the optimization of run-time tests and alias disambiguation a very hard problem. Our approach to improving Java performance starts with the introduction of true multidimensional arrays, through a fully Java-compliant class library (package). Using the properties of these multidimensional arrays, our prototype compiler builds safe and alias-free regions of code that are guaranteed to be free of exceptions and aliasing between arrays. Extensive loop transformations and automatic loop parallelization can be applied to these regions. Our results show that, for many benchmarks, Java can be very competitive in performance, delivering between 80-100% of the performance of the best Fortran compilers. For some cases, we also observe close to linear speedup on a modest sized (4-processor) SMP.

Software Components

Steven Newhouse

Imperial College of Science, Technology & Medicine, UK

(joint work with Anthony Mayer and John Darlington)

We introduce a component software architecture designed for demanding grid computing environments. That allows the optimal performance of the component based applications to be achieved. Performance over the assembled component application is maintained through inter-component static and dynamic optimisation techniques. Having defined an application through both its component task and data flow graphs we are able to use the associated performance models to support application level scheduling. By building grid aware applications through reusable interchangeable software components with integrated performance models, we enable the automatic and optimal partitioning of an application across distributed computational resources.

Using Java within the Grid

Gregor von Laszewski

Argonne National Laboratory, U.S.A.

Emerging national-scale "Computational Grid" infrastructures are deploying advanced services beyond those taken for granted in today's Internet: for example, authentication, remote access to computers, resource management, and directory services. The availability of these services represents both an opportunity and a challenge for the application developer: an opportunity because they enable access to remote resources in new ways, a challenge because these services may not be compatible with the commodity distributed-computing technologies used for application development. The Commodity Grid project is working to overcome this difficulty by creating what we call

Commodity Grid Toolkits (CoG Kits) that define mappings and interfaces between Grid and particular commodity frameworks. In this paper, we explain why CoG Kits are important, describe the design and implementation of a Java CoG Kit, and use examples to illustrate how CoG Kits can enable new approaches to application development based on the integrated use of commodity and Grid technologies.
More information is available at http://www.globus.org/cog.

Actors and High-Performance Java

Gul Agha

The University of Illinois at Urbana-Champaign, U.S.A.

The goal of a high-level programming language is to provide an easy to understand and correct representation which can be nevertheless executed efficiently. In other words, the level at which programs are written should be as close as is feasible to a mental model for the computation. Actors provide a natural model for concurrent activity extending objects to encapsulate threads and support mobility. Compilers and run-time systems developed for actors show that it is possible to obtain the same performance using high-level actor based languages as it is with the fastest lower-level langaguages such as Split-C. By contrast, Java is a difficult language to program parallel and distributed systems. Threads can interfere with each other, and excessive synchronization can cause deadlocks. Moreover, migration is excessively expensive as is communication (serialization and synchronous communication). We have also studied supporting actors in Java and experimented with an actor-based dialect of Java. Our conclusion is that actors can simplify programming while improving execution efficiency.

Synchronous Java Event Spaces

Alexander Knapp

Ludwig-Maximilians-Universität München, Germany

The Java Language Specification defines an asynchronous interaction between the main memory and the threads' working memories by pairings of *read* and *write* actions on behalf of the main memory and *load* and *store* actions on behalf of the threads. We demonstrate that this asynchronous behaviour indeed is observable. However, for the special class of properly synchronised Java programs, limiting access to shared variables to synchronised regions, we prove that the Java memory model may be simplified by identifying *read–load* and *store–write* pairs. The proof is based on a straightforward formalisation of the Java memory model as partial orders, called event spaces.

Reference Analysis in Java

Paul A. Feautrier

Université de Versailles, France

Java inefficiency is the price we pay for Java safety and portability. One way of recovering performance without sacrificing safety is hoisting as much as possible the run-time checks to compile time. This can be done only if we have a way of precisely analysing the relation between references and objects (the "points-to" relation). I propose such an analysis method, in which "reference equations" are solved by a process akin to Gaussian elimination. At present, the solution process is incomplete and applies only to static control programs, but extensions are contemplated. Applications include removing array bounds checks, flattening arrays, helping the garbage detector and computing dependencies.

Shape Analysis

Reinhard Wilhelm

University of Saarland, Germany

(joint work with Mooly Sagiv and Thomas Reps)

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The results can be used to understand and verify programs. They also contain information valuable for debugging, compile-time garbage collection, instruction scheduling, and parallelization.
Our approach is parametric, i.e., it can be tailored to a particular application by defining the means of observation through which to look at the contents of the heap.

Making Java Unexceptionally Fast

Christian Probst

University of Saarland, Germany

Static analysis for imperative languages is a topic quite well understood. Having tools like the program analyzer generator PAG at hand allows the compiler write to concentrate on specifying his analysis instead of on implementing all kind of administrative stuff. The generated analyzers then can easily be integrated into existing compilers and optimizers.
Those analysers work on the static interprocedural control flow graph that must be constructed before starting the analysis. While this construction is easily done for imperative languages things get more complicated for object oriented ones. Here one may not know at compile time for a variable to which class' objects it may point to at run time. As this classes are used to determine possible targets for method calls this also hinders construction of

a precise control flow graph, introducing superfluous call edges and thereby sources for data dependencies.
We present a mixture of flow-insensitive and flow-sensitive approaches, where the first ones are used to construct a conservative approximation of the control flow graph. The second ones work on this graph in order to refine it by narrowing the set of classes that a variable may point to.
Those analysers work on the static interprocedural control flow graph that must be constructed before starting the analysis. While this construction is easily done for imperative languages things get more complicated for object oriented ones. Here one may not know at compile time for a variable to which class' objects it may point to at run time. As this classes are used to determine possible targets for method calls this also hinders construction of a precise control flow graph, introducing superfluous call edges and thereby sources for data dependencies.
We present a mixture of flow-insensitive and flow-sensitive approaches, where the first ones are used to construct a conservative approximation of the control flow graph. The second ones work on this graph in order to refine it by narrowing the set of classes that a variable may point to.
In addition, we have developed an analysis allowing to identify superfluous checks for null pointers and classes not yet initialized. These work by keeping track of objects already checked (classes already initialized) by partially evaluating if-statements. First benchmarks counting the static number of removed checks look rather promising.

Object Inlining

Peeter Laud

University of Saarland, Germany

The semantics of objects in Java requires in general, that each object is represented as a pointer to the area in the heap, which contains the actual data of the object. This will result in fragmentation of the heap, thereby causing extra effort in dereferencing the pointers, making heap management (garbage collection) more time-consuming and also having negative effect on the locality of the data, which may result in decreased cache performance.

In contrast to that, in C++ (and also in other languages) one also has the possibility to declare the fields of classes to be "aggregated", i.e., their content is kept together with the object, instead of being made to follow a pointer to get it. Object inlining means changing the layout of objects by aggregating subobjects into their containers. Of course, one has to be careful not to change the semantics of the program that way.
In our talk, we describe an analysis of Java, which conservatively estimates, which fields of which objects may be aggregated. Our analysis is based on estimating the sharing patterns, but we try to avoid doing a full-blown alias analysis, and instead try to detect only patterns having relevance to deciding, whether inlining is allowed.

Compilation Techniques for Explicitly Parallel Programs

Jaejin Lee

Michigan State University, U.S.A.

A problem faced by compilers of explicitly parallel languages, such as Java, OpenMP, and Pthreads, the solution of which is the focus of this talk, is that data races and synchronization make it impossible to apply classical optimization and analysis techniques directly to shared memory parallel programs because the classical methods do not account for updates to variables in threads other than the one being analyzed.
Many current multiprocessor architectures follow relaxed memory consistency models. However, this makes programming and porting more difficult. Moreover, sequential consistency, which is not a relaxed consistency model, is what most programmers assume when they program shared memory multiprocessors, even if they do not know exactly what sequential consistency is.
In this talk, we present analysis and optimization techniques for an optimizing compiler for shared memory explicitly parallel programs. The compiler presents programmers with a sequentially consistent view of the underlying machine irrespective of whether it follows a sequentially consistent model or a relaxed model. Furthermore, the compiler allows optimization techniques to

be applied correctly to parallel programs that conventional compilers cannot handle.
To hide the underlying relaxed memory consistency model (including the Java memory model) and to guarantee sequential consistency, our algorithm inserts fence instructions. We reduce the number of fence instructions by exploiting the ordering constraints of the underlying memory consistency model and the property of the fence instruction. To do so, we introduce a new concept called dominance with respect to a node in a control flow graph. We also show that reducing the number of fences by minimizing the number of nodes is NP-hard.
We introduce two intermediate representations: the concurrent control flow graph, and the concurrent static single assignment form. Based on these representations, we develop an analysis technique, called concurrent global value numbering, by extending classical value partitioning and global value numbering. We also extend commonly used classical compiler optimization techniques to parallel programs using those intermediate representations. By doing this, we guarantee the correctness (sequential consistency) of the optimized program and maintain single processor performance in a multiprocessor environment. We also describe a parallel loop overhead reduction technique.

Java Compilation for Embedded Systems: Current and Next Generation

Christian Fabré

Silicomp Research Institute, U.S.A.

The Silicomp Research Institute is developing Java solutions for the embedded world. One of our belief is that, due to the scarce CPU and RAM resources available in embedded systems, there is no room for a compiler within the VM. Therefore, compilation must be done outside the JVM.
We have developed two such bytecodes to native compilers dedicated to the embedded world, and which can be used with off-the-shelf VMs.
Turbo is precompiling a whole application, with the intent of burning the resulting native code into the embedded device's ROM. Besides traditional

intra-method optimisations, Turbo can make a number of global optimisations due to its knowledge of the structure of the whole application.
Fajita can compile method per method and aims at low lattency compilation amd easy retargetability. It can be used in two scenarios: on-demand-compilation, where a small (approx. 10K) profiler sitting within the VM can ask a remote server for the compilation of a "hot" method; and compile-on-the-way, where the compiler is used within a class proxy and the native code is propagated within an extension of the class file format. Both scenarios preserve the full Java dynamicity.
In this presentation, we present the details of Turbo and Fajita, as well as our view of where the embedded world is going in terms of solutions and technologies for Java performance.

The Spar/Java Programming Language

Henk J. Sips

Delft University of Technology, The Netherlands

(joint work with & C. van Reeuwijk)

In recent years, embedded systems with multiple processors have become increasingly important. These systems often consist of a general-purpose processor and one, or even several, digital signal processors (DSPs). For portability, flexibility, and robustness it is often useful to regard such a cluster as a single, parallel system. Since such a system contains several types of processors, it is said to be heterogeneous. In its simplest form, parallelization of applications can be done using an explicit parallel programming interface, such as provided by libraries like MPI, or Java threads. Unfortunately, programming using this model requires a detailed understanding of the complex interplay between program and machine, which is extremely laborious and error-prone. Moreover, this approach is often not portable. Instead, in the Spar/Java language, we follow a more implicit approach, where parallelism is generated by the compiler, supported by user annotations to the program. We have designed the Spar/Java **[something funny here...]** supports Java, augmented with the each and foreach constructs for

semi-implicit parallelization. The each and foreach expose opportunities for parallelization to the compiler, but the details of the parallelization are left to the compiler. This allows for more robust and portable parallelization. The each and foreach statements, together with with language extensions for multi- dimensional arrays and support for more specialized arrays, form the 'Spar' language extensions. Spar/Java includes an annotation language that allows the description of explicit or partially explicit placement of data and tasks on processors in a homogeneous or heterogeneous parallel system. In Spar/Java, all placements are static at run time; no object migration between processors is done. The annotations do not alter the semantics of the program, they just serve as hints for the scheduler in the compiler. The annotations allow the placement of both tasks and data to be specified in a uniform manner.

Java-Based Parallel Computing on the Internet

Peter Cappello

University of California at Santa Barbara, U.S.A.

Javelin is a Java-based system for parallel computation on the Internet. We present Javelin 2.0, focusing on enhancements that facilitate aggregating larger sets of host processors, a branch-and-bound computational model and its supporting architecture, a scalable task scheduler using distributed work stealing, an eager scheduler implementing fault tolerance, and the results of performance experiments. Javelin 2.0 frees application developers from concerns about interprocess or communication and fault tolerance among Internetworked hosts. When all or part of their application can be cast as a piecework or a branch-and-bound computation, Javelin 2.0 allows developers to focus on the underlying application, separating logic for communication and fault tolerance from application logic.
Future work, based on Jini and Javaspaces technology, is discussed briefly.

OpenMP and Java

Barbara M. Chapman

University of Houston, U.S.A.

Java provides threads for share memory parallel programming. However, they are defined in a manner that facilitates the coordination of a number of different tasks, and do not offer the functionality used to develop parallel scientific applications. Although it is possible to write data parallel code for shared memory systems using Java threads, the resulting code will be very low-level and may differ significantly from the original sequential code. OpenMP is a recent de facto standard for shared memory parallel programming that has bindings for Fortran, C, and C++. It provides a traditional APT for scientific and technical compiling; but relying largely on directions, the base language does not need to be extended. We introduce the OpneMP features, and discuss OpenMP language issues, its application and limitations. OpenMP was quickly adopted by the user community, and it is proposed that a Java binding be made available, so that Java users may also take advantage of this higher-level programming interface.

Tool Requirements for High-Performance Java

Cherri M. Pancake

Oregon State University, U.S.A.

A variety of tools are available to assist in the development of high- performance Java (HPJ) applications. They are primarily directed at three aspects of development: creating and launching parallel or distributed programs, debugging HPJ, and tuning HPJ performance. This presentation gives a brief survey of HPJ tools, showing how they have evolved from two sources, serial Java IDEs and parallel tools for other languages. Comparisons with the requirements of typical HPJ users demonstrate that today's tools are lacking many basic capabilities that are critical for developing effective programs. It is unlikely that significant numbers of new users will be attracted to HPJ until those gaps can be addressed.

Portability of Parallel and Distributed Applications

Ami Marowka

The Hebrew University of Jerusalem, Israel

Portability has become an important consideration in parallel application design.

The word portable, or portability, has been widely and often used in the parallel processing community. However, there is no adequate, commonly accepted definition of portability evaluation available. Portability evaluation of parallel application is difficult to quantify, evaluate, and compare. In this talk, I will introduce a framework called Scalable Portability Evaluation Methodology (SPEM) for analyzing the parallel portability.

The novelty of SPEM methodology is in its ability to compare, evaluate and analyze the parallel portability over different applications and architecture. Portable speedup and efficiency metrics are defined and function as the basis for the definition of a new parallel portability metric called Portability Degree (PD). Based on the new metric the Scalable Portability (SP) approach to analyze portability of a parallel system is formally defined.

The empirical nature of the problem calls for experimental comparisons across parallel machines. Then an abstraction of a parallel system from the viewpoint of a single process called Parallel Processes System (PPS) is defined. The PPS abstraction serves as a base model for the development of a proof-of-concept prototype of a scalable and portable parallel system called Scalable-PVM (S-PVM).

The effectiveness of the Portability Degree metric was tested using two sets of experimental studies. The first set, consisting of two scientific applications that were developed by S-PVM prototype, was tested on three different homogeneous parallel machines architecture. The second set, consisting of five kernels from the NAS Parallel Benchmark (NPB) suit, was studied in a heterogeneous environment of supercomputers.

Scalable portability analysis using SPEM methodology was applied to the tested applications. The experimental results show that the newly defined

Portability Degree metric provides a unique quantitative measurement to describe the scalable portability of a parallel application-environment combination as sizes are varied which cannot be provided by any other scalability metric.

3 List of Participants