Multiparadigm, Multilingual Interoperability: Experience with Converse

L. V. Kalé, Milind Bhandarkar, Robert Brunner, and Joshua Yelon

Dept. of Computer Science, University of Illinois

Abstract. The Converse run-time framework was designed with dual objectives: that of supporting quick development of portable run-time systems for new parallel programming paradigms, and that of permitting interoperability between multi-paradigm modules in a single application. This paper reports on the refinements made to the original Converse model since its inception almost two years ago, and assesses our experience in using Converse to satisfy the above objectives. A brief overview of the motivation and overall design of Converse is included for completeness. Extensions and refinements in Converse are discussed along with the reasons for their inclusion. Several languages/paradigms were implemented using Converse; techniques used in these implementations and our experience with specific features of Converse used in them are discussed. A major multilingual multi-paradigm parallel application developed within the Converse framework is described.

1 Introduction

Programming parallel machines is a complex task, especially if one considers the broad class of applications one may wish to parallelize. The currently popular paradigm is characterized by the raw or direct use of MPI: all processes execute essentially identical code, exchanging messages via explicit sends and receives at predetermined points. Although this paradigm is reasonably suited to many applications with relatively regular parallel structure, there are many applications for which it is not a good match. As the class of parallel applications broadens and includes more complex applications than attempted so far, it is natural that new parallel paradigms will be invented and studied. Indeed, many such approaches including multithreading, message driven objects, gather-scatter libraries, etc. are being explored.

Such new paradigms, some of which will undoubtedly prove useful in the long run, face two major hurdles. First, to implement the run-time system for a new paradigm on a parallel machine is often a daunting task. This is further compounded by the diversity and multitude of different parallel machines. Second, user acceptance of new paradigms is difficult to attain. Users are naturally reluctant to commit their entire applications to an experimental approach. Also, a new paradigm may prove inappropriate for some parts of the application, even though other parts may profit from it in efficiency or readability.

The Converse run-time framework was designed to overcome both of these obstacles. Converse is a runtime system that provides portable, efficient implementations of all the functions typically needed by a parallel language or library. One of Converse's primary goals is interoperability. Converse enables modules written in different languages/paradigms to coexist and, indeed, overlap in a single application. This interoperability reduces barriers to user acceptance. Users can employ a new paradigm for a small select number of modules, while retaining their favorite or traditional paradigm for the rest of the application. This allows the novel approaches to establish a toe-hold, from which they can expand based on their merit. Further, users can select an appropriate paradigm for implementing each module individually.

This report serves as a progress report and evaluation of Converse.

2 Converse Architecture and Refinements

The Converse run-time framework was designed with the objective of supporting a wide variety of parallel programming paradigms without requiring them to sacrifice efficiency. This means that the implementation of any parallel language using Converse must be almost as efficient as a native implementation on each particular machine. Results so far [3] suggest that Converse meets this goal. To achieve this level of efficiency, the Converse API must subsume capabilities offered by most parallel machines. To ensure that each language pays for only those features that it uses, the Converse architecture consists of a set of very simple, efficient core components. Higher-level functionality is layered on top of this core in the form of optional modules. The idea of need-based cost is discussed further in [11]. The set of core components is chosen so that a wide variety of programming models, and their coexistence, can be supported.

Converse models the machine as a set of semi-independent processors that communicate primarily via messages. Each processor is running a scheduler, which is responsible for all message reception. Each message contains a function pointer and some bytes of user data. The scheduler thread's job is to repeatedly pick and process the highest priority message (or other event) it has received. Processing a message consists of calling the function pointer which is in the message, passing in the message's data. This functionality is encapsulated in three distinguishable modules: a threads module, the scheduler, and the machine interface.

The machine interface's most important part is the communication primitives. These primitives insert messages into the scheduler queues at remote processors, where the scheduling thread finds and processes them. To ensure that the abilities provided by the machine are properly exploited, several variants of communication primitives are provided that support synchronous and asynchronous communication and various forms of broadcast.

The Converse scheduler serves not only as a message receiver, but also as a central allocator of CPU time. Many parallel paradigms, such as the multithreaded models and message driven objects, require a scheduler. If each library had its own scheduler, then control of the CPU would not transfer from one library to another.

The scheduler supports prioritized messages. These priorities are very flexible: we even allow arbitrary-precision numbers to be used as priorities. This is extremely useful for many AI applications. Importantly, though, the scheduler is designed in such a way that the cost of priority queueing is only paid when priorities are being used.

The thread component is much like traditional thread systems. However, the Converse user is given explicit control over very low-level primitives like context-switching, suspension, and thread handles. This separation of the primitive thread functions makes it possible for the users to design their own synchronization primitives and scheduling mechanisms very inexpensively. Converse threads are currently nonpreemptive. This is useful in that it eliminates most locking and thread-safety issues, in turn making functions like `malloc` much more efficient.

Threads are normally scheduled automatically by Converse, but hooks are provided so that the Converse user can write alternate scheduling code. This is not unlike a facility provided by OpenThreads [8]. However, to keep conflicts between modules to a minimum, alternate scheduling code only affects those threads that explicitly request it. This helps preserve interoperability.

Support for shared memory and shared variables is provided in a portable manner as described in the section on the machine model below. In addition, the machine interface supports several other utilities such as timers and terminal I/O.

In addition to these fundamental parts which have been present since Converse's inception, several new modules have been added. The following sections describe these recent additions.

2.1 Shared Memory

Converse assumes the machine to be a collection of *nodes*, where each node consists of one or more *processes* sharing memory (some systems call these processes "kernel threads".) Each process operates with its own scheduler, and messages are usually directed to processes, not nodes. Each process houses an arbitrary number of Converse threads. This model covers all the memory hierarchies in existing machines, from the distributed memory machine at one end to the fully-shared machine at the other, with clusters of SMPs in-between (Converse has recently been ported to the SGI Origin 2000 and networks of SMP workstations).

Even though shared memory is there on some machines, Converse intentionally keeps processors separate from each other, with mostly-independent data structures. In other words, Converse presents the impression of a distributed memory machine. Since the processors are kept separate, and threads are nonpreemptive, the Converse user rarely has to implement any kind of locking or thread-safety code. The Converse user can access shared memory, but this can only be done through explicit action. Typically, locking is only needed in modules that explicitly use shared memory. This is in contrast to many runtime systems that require every data structure to be protected with locks. Many Converse modules use the shared memory internally. In particular, Converse messaging uses it for efficient delivery where possible. Thus, much of the benefit of shared memory is gained, with none of the complexity.

The C programming language supports global variables (global in scope, not shared across processors). Compilers for distributed-memory hardware typically give each processor a separate copy of the globals, but compilers for shared-memory hardware typically make only one copy. Thus, attempts to use global variables tended to create portability problems. Converse solves this by adding explicit control over sharing of global variables. Converse extends C (through macros) to include thread-private, processor-private, and shared variables. Each Converse thread has its own copy of the thread-private variables. Each processor has a copy of the processor-private variables, and all the threads housed on that processor share the copy. Each node has a copy of the shared variables, which is available to all the entities in that node.

2.2 Global Pointers

A global pointer is a triplet consisting of a processor identifier, memory address local to that processor, and size of that memory block. Any processor can declare a block of its private memory globally accessible by creating a global pointer associated with it. This global pointer is a first class object which can be sent to other processors using messages[1]. Remote processors can then use functions provided in Converse to asynchronously or synchronously read and write (Get and Put) into this memory. Converse' global pointer mechanism is being used in the implementations of an all-software Distributed Shared Memory (DSM) scheme called CRL (in progress), and I-structures[1]. Parallelizing compilers implemented assuming a runtime support for DSM, can be retargeted to use Converse global pointers.

2.3 Vector Sends

Messaging routines often accept user data, concatenate a header onto that data, and then send the header and data together. For example, the MPI messaging routines accept a pointer to data, but they also accept a "communicator" object. The communicator and the data are then transmitted together. The act of concatenating a header to message data often requires copying the header and data into a new buffer. This is inefficient. To help the implementors of such routines, Converse now provides a set of *vector-send* primitives. These primitives accept an array of pointers to data. Each element in this array represents a piece of a message to be delivered. The pieces of data are concatenated and sent, but the concatenation is often done very efficiently, without the overhead of copying. For example, on machines where messages are packetized, the packets are simply gathered by scanning the array and packetizing each piece separately. On

[1] A mechanism is also provided to share global pointers by associating well-known names with it.

machines with shared memory, messaging is sometimes implemented by copying a message from the sender's memory to the receiver's memory. In this case, vector send is implemented by concatenating straight into the receiver's memory. All of these optimizations are done transparently to the Converse user by the implementation of the vector-send primitives, thus reducing programming complexity.

2.4 Threads

In version 4.5, thread support was added for some machines, in version 4.6, it was ported to all machines. The original implementation was somewhat machine-dependent, and thus we had multiple versions of the threads package. This turned out to be a maintenance problem: each time we added a feature to the threads package, we had to code the feature into each version separately. The problem was solved by reimplementing Converse threads on top of the QuickThreads [17] library. While Quickthreads does contain machine-dependent code, the fact that the Quickthreads API has been frozen for years prevents this from becoming a maintenance issue. Any features added to Converse threads are now implemented on top of the stable Quickthreads layer. Porting is much easier, since Quickthreads uses a minimalist design. We have ported QuickThreads to several new platforms, including the Cray T3D and T3E, the Origin 2000, the IBM RS6000, and the SP1 and SP2. The use of the QuickThreads API makes our porting efforts useful to all thread users, even outside the Converse user-community. Finally, Converse threads context-switch time for Converse has also been reduced, as many of the QuickThreads ports were hand-coded in assembly language.

2.5 Utilities and Libraries

In addition to its core components, several convenience modules are provided with Converse. These are designed to simplify the task of implementing runtime systems for parallel languages.

The *message manager* is a table object for storing messages according to a set of integer tags. It supports variable numbers of tags, and wild cards in the lookup process. It can also be used to store any data that must be indexed by integer tags. The *futures* library implements the futures abstraction [9] in a library-form. It provides a future "object", with methods to fill the object remotely, get its value, and block until the value has been filled. The *Converse Parameter Marshalling* system is a small C preprocessor and code generator that produces remote function-invocation code. One inserts the keyword `CpmInvokable` into a C source file in front of a function definition. The CPM preprocessor scans the C file, and generates code to invoke that function remotely. The CPM-generated code automatically packs up the arguments into a message, sends them to the destination, and invokes the function in question. The *POSIX threads* API has been implemented on top of Converse threads. These POSIX threads can interoperate with Converse threads, and the rest of the Converse system.

3 Languages and Libraries Implemented in Converse

Several parallel programming languages and libraries have been implemented in the past two years using Converse. The number of these languages and the ease with which we were able to implement them strongly demonstrates the utility of Converse. In the following subsection, we describe in detail how one simple language was implemented using Converse to illustrate how Converse is used for building language runtimes. The following subsections describe several other languages and libraries implemented on top of Converse.

3.1 Implementation of a Multithreaded Messaging Library

This section shows the implementation of CSM, a message-passing library. CSM was designed for illustration purposes, it is intentionally the simplest possible library that implements message-passing with threads. The basic design shown here can be used to implement any message-passing library, including MPI or PVM. The following function descriptions are from the CSM manual:

`void CsmTSend(int pe, int tag, char *buffer, int size)`

A message is sent to the given processor `pe` containing `size` bytes of data from `buffer`, and tagged with the given tag. The calling thread continues after depositing the message with the runtime system.

`int CsmTRecv(int tag, char *buffer, int size, int *rtag)`

Waits until a message with a matching tag is available, and copies it into the given buffer. A wild card value, `CsmWildCard`, may be used for the tag. In this case, any available message is considered a matching message. The tag with which the message was sent is stored in the location to which `rtag` points. The number of bytes in the message is returned.

Our implementation buffers messages on the destination processor. To implement this using Converse, two major data structures are needed. First, each processor needs a "message table" containing messages that were sent, but for which no `CsmTRecv` call has been issued yet. Second, each processor needs a "thread table" containing threads that are waiting for messages, indexed by the tags that they're waiting for. Given these data structures, the send and receive functions are implemented as follows.

`CsmTSend` creates a Converse message containing the user data and the tag. It configures the message to invoke the function `CsmTHandler`. `CsmTSend` then transmits a copy of this message to the destination processor. When the message arrives, the target processor calls `CsmTHandler`, passing it a pointer to the message (which contains the user data and tag). `CsmTHandler` takes the user data and tag, and inserts it into the local message table. It then checks the thread table to see if any thread was waiting for the message. If so, that thread is awakened.

When a thread calls `CsmTRecv`, it looks in the message table, and if a matching message is already there, it is extracted and returned. If not, `CsmTRecv` obtains its own thread ID, and inserts itself into the thread table. It then puts itself to sleep. When it wakes up, it knows it has been awakened by `CsmTHandler`. It retrieves the message from the message table, and returns it.

For the message and thread tables, we used an off-the-shelf table object provided by Converse (the "message manager"). Thus, our data structures were largely implemented for us. The thread functions were provided, as was the messaging. We had to design the format of the CSM messages (header, then tag, then user data), write the subroutines shown above, and declare and initialize the tables. All in all, this took about 100 lines (2 pages) of code. This is interesting, as the message-passing model we implemented is significantly different from the underlying message-driven model.

Notice that no explicit action was needed to keep CSM from interfering with other libraries also implemented on top of Converse. CSM messages, when they arrive, trigger changes to the CSM data structures. They have no other effect. If a library system does not explicitly monitor the CSM data structures, it will not be aware that a CSM message arrived. In general, two libraries implemented on top of Converse do not notice each other's existence unless explicit action is taken to create interaction. This is in contrast to such systems as MPI, where each independent module must take explicit action (e.g., the creation of new communicator objects, etc) to avoid interfering with other modules.

3.2 PVM and MPI

We have implemented both MPI and PVM on top of Converse. This makes it possible to link modules written with the PVM or MPI primitives to each other or other parallel languages.

The MPI implementation[3] is based on MPICH [22]. MPICH is an implementation of MPI which can be easily retargeted, we simply retargeted it to Converse. Interestingly, the Converse port of MPICH is very close in efficiency to the native port of MPICH on the machines we tested. However, the Converse version gains all the interoperability benefits of Converse.

Our version of PVM is a from-scratch reimplementation of much of the PVM 3.3 C library. The code is 2000 lines, most of which is the message-packing routines. The PVM process-management calls are not available under Converse, but dummy functions are provided for easy translation of standard PVM code. The implementation of PVM-Converse is much like the implementation of CSM as described above. The PVM-Converse library is currently used in a major production quality molecular dynamics application (See section 4.)

3.3 Message-Driven Languages

Charm[16] and Charm++[13] were developed before Converse. The design of Converse was a result of our experiences with making Charm and Charm++ portable and making modules written in both Charm and Charm++ interoperate within the same application. Charm and Charm++ were later retargeted to Converse. The most difficult part of this retargeting was moving Charm's support for multiprocessor nodes into Converse. For example, the specifically shared variables had to be reimplemented in order to take advantage of node-level memory access, and in-built node-level locks instead of using language-specific abstractions. Also, Charm and Charm++ contained dynamic load balancing facilities. However, in an application with multiple language modules, load balancing should be done in the global context taking into account the entire load across all the language modules. Thus load balancing facilities were moved into Converse, and Charm++ runtime was written to use this load-balancing facility.

More recently, Charm++ has been simplified to use an interface definition language to produce wrappers for objects and entry methods in Charm++. Also, new additions to entities supported by Charm++ were made in the form of object arrays. These additions were made at the Converse level and the modular architecture of Converse helped to quickly develop a prototype version of these libraries. Because of the interoperable nature of languages implemented on top of Converse, these entities could also be used across multiple language modules.

We also developed a Java binding for the Charm++ constructs and entities such as remote objects with global name space, and asynchronous method invocation using Converse[12]. Currently this binding is supported on machines where Sun's Java Development Kit 1.1 is available. Converse runtime system functionality was made available to this implementation through native methods of the `PRuntime` class. We had to modify the machine layer implementation on networks of workstations, because the common resources used by JDK1.1 are not separated into kernel-level threads in the JDK implementation. This problem will be fixed in the future versions of JDK.

3.4 Data Parallel Languages

DP[18], a subset of High Performance Fortran (HPF) was implemented on top of Charm++ before the development of Converse. After Charm++ was retargeted to Converse, DP was automatically retargeted and is available for programming data parallel algorithms in a multilingual application.

pC++ [5, 2] is a C++ extension for data-parallel computation using collection of objects. The method execution semantics of C++ objects is extended to include method invocation in parallel on a collection of objects. The pC++ implementation consists of a translator that converts pC++ constructs into ANSI C, and generates calls to the runtime system functions. The runtime system of pC++ is called Tulip. Tulip offers a subset of Converse functionality. In particular, it supports a single handler function per application. Also, the scheduler for Tulip operates in a "polling" mode, and calls to this scheduler are inserted in application by the translator. It was indeed very simple task to provide an implementation of pC++ on top of Converse.

3.5 Other Languages

Several experimental languages have been recently implemented on top of Converse. We describe these languages briefly.

Import [15] is a simulation language based on Modsim. Import models a simulation system as a set of objects. The sequential version of the language has a centralized discrete event queue where each event is a time stamp, an object, and a method. Our parallel implementation replaces the centralized discrete event queue with the Time Warp concurrency control mechanism. This simulates a strict time-stamp-ordered execution of events, when in fact events are being executed in parallel. The implementation of Time Warp was challenging, but the Converse interface was straightforward. The implementation relies upon the Converse messaging primitives, its priority mechanisms, and its and shared variable mechanisms. Speedups for our sample simulations have been excellent.

Agents [23] is an experimental object-oriented language dedicated to exploring the idea of immutable, static networks of objects. The language supports remote method invocation, and thus, its implementation is much like the implementation of Charm (see section 3.3). The runtime system of the language took only a few hundred lines of code, though the compiler and optimizer were much more complex.

Perl is a popular scripting language mainly used for text processing and system administration tasks. Message Driven Perl (**mdPerl**) is a package for Perl 5 programs that allows writing message-driven parallel programs in Perl. The basic capability of mdPerl is to invoke subroutines on remote processors. mdPerl provides a subroutine `mdCall` that specifies a processor number, name of the subroutine to be invoked on that processor and arguments to be passed to that subroutine. The implementation consists of approximately two hundred lines of C code to interface Perl with Converse. The scheduler of Converse is made available to Perl programs to schedule computations received from remote processors. For common programming tasks such as analyzing the log information of a WWW server, we have obtained a near linear speedup using mdPerl.

4 Multilingual Programming: Applications

NAMD [19] is a parallel molecular dynamics simulation program being developed with the Theoretical Biophysics group at the University of Illinois. NAMD simulates the motions of biological molecules such as proteins and DNA by repeatedly computing the forces exerted by individual atoms on one another, and integrating the motion due to these forces over time.

The original version of the program, NAMD 1, was built using a message-driven design. Two variants of the program were initially developed. One variant used Charm++, which provided support for straightforward expression of the message-driven design. The other variant used PVM in order to allow us to use the DPMTA [20] library developed by collaborators at Duke University. The DPMTA library provides efficient long-range electrostatic force computation, which is necessary for some simulations. The PVM variant of NAMD originally had a message-driven design, but new features tended to be added around rather than within the message-driven core design, sacrificing maintainability. Eventually, since DPMTA made the PVM variant more useful, more development time was spent on it, and the Charm++ version fell into disrepair. By that time, though, the haphazard mix of SPMD and message-driven code had reduced the readability of the program.

NAMD 2 [10] was conceived as a rewrite of the core parallel code to increase scalability and modifiability. Our experience with NAMD 1 led us to conclude that a message-driven design was appropriate for the parallel core code, but that adding threads to the design would allow the integration loop to be expressed as a loop construct, with occasional thread suspension to wait for data. We also observed that much of the startup code was easier to write in an SPMD style. Furthermore, we did not want to rewrite the input routines or the DPMTA library, so we needed support for PVM. Fortunately, Converse satisfies all of these goals.

The case for multilingual programming is vividly made by our new design for the integration logic that is the core of NAMD 2. The simulation space is divided into cubical regions called *patches*, each of which can be simulated in parallel, requiring only information from neighboring patches. Each patch has an associated object called a *sequencer*, which is responsible for integrating the equations of motion for the atoms owned by the patch. The sequencer contains the code that, for each time step, sends out atom positions, retrieves the forces calculated for those positions, and then computes the positions at the next time. Each sequencer is initially created by a Charm++ object in its own Converse thread. First, the sequencer sends atom positions to *compute* objects, Charm++ objects that actually perform the force computations. Then the sequencer suspends its thread. The receipt of the force messages from the last compute object causes the sequencer thread to awaken, and the forces are used to update the positions for the patch. The author of a particular sequencer code writes the logic as a loop, and the only attention he must pay to the parallel nature of the code is to insert the thread suspend calls in the correct places. This satisfies one of the major design goals of NAMD 2, namely that the user of the program, typically a physicist or chemist with experience writing sequential programs, can implement a new integration algorithm without having to understand details of the parallel code.

NAMD 2 is now being tested, and nearing release. Increased sequential and parallel efficiency makes the program faster than NAMD 1, indicating that a multilingual Converse program pays little or no price for programming convenience. NAMD 2 already supports almost all features of NAMD 1, and several significant features which were never part of the original program, providing evidence of the improved modifiability of its multilingual design.

5 Conclusion

The Converse run-time framework was designed with two objectives: To simplify the development of run-time systems of parallel languages, and to allow interoperability between modules written in different languages (often representing diverse parallel programming paradigms) within a single application. In this paper, we presented evidence that Converse has indeed attained these goals. We also described several extensions to Converse in the form of new features and libraries, that simplify development of run-time systems and enhance their portability. More than ten distinct libraries/languages have been implemented on top of Converse, demonstrating utility of the framework and completeness of its functionality. We also describe a large production quality application that employs modules written using different languages running on top of Converse.

Several other systems have been designed with objectives similar to Converse. Nexus [6] supports dynamic addition of processors on the Internet and provides flexible communication protocol. These features are useful for some applications but they add complexity to the programming model. Run-time frameworks aimed at supporting parallel languages on dedicated parallel machines include Tulip [2], Panda [4], and Chant [7]. We believe that a few additional frameworks are also being developed at research laboratories. We propose to collaborate with these researchers to combine useful features from these systems so as to broaden the set of paradigms that can be supported by run-time frameworks.

Current unfinished work on Converse includes messages directed to nodes instead of processors, profiling and tracing support, ports of more programming models to the Converse API, more convenience modules, transparent load-balancing across nodes, and ports to new machines. Future work may include support for networks of heterogeneous workstations, new language bindings, dynamically adding nodes, and support for thread preemption.

References

1. Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.
2. P. Beckman and D. Gannon. Tulip: A portable run-time system for object-parallel systems. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.
3. Milind Bhandarkar and L. V. Kalé. MICE: A Prototype MPI Implementation in Converse Environment. In *Proceedings of the second MPI Developers Conference*, pages 26–31, South Bend, Indiana, July 1996.
4. Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 213–226. USENIX, San Diego, CA, September 1993.
5. F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language, 1992.
6. Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, May 1994.
7. M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing 1994*, Washington D.C., November 1994.

8. Matthew Haines and Koen Lagendoen. Platform-independent runtime optimizations using openthreads. In *11th International Parallel Processing Symposium*, april 1997.
9. R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.
10. L. V. Kalé, Milind Bhandarkar, Robert Brunner, Neal Krawetz, James Phillips, and Aritomo Shinozaki. A case study in multilingual parallel programming. Technical report, Theoretical Biophysics Group, Beckman Institute, University of Illinois, Urbana, June 1997.
11. L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.
12. L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel java with a global object space. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, Las Vegas, Nevada, July 1997.
13. L. V. Kalé and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
14. L. V. Kalé and Joshua Yelon. Threads for Interoperable Parallel Programming. In *Proc. 9th Conference on Languages and Compilers for Parallel Computers*, San Jose, California, August 1996.
15. L. V. Kalé, Joshua M. Yelon, and T. Knauff. Parallel import report. Technical Report 95-16, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995.
16. L.V. Kalé. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.
17. David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.
18. E. Kornkven and L. V. Kalé. Efficient implementation of high performance fortran via adaptive scheduling – an overview. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.
19. Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. NAMD— A parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.
20. W. Rankin and J. Board. A portable distributed implementation of the parallel multipole tree algorithm. *IEEE Symposium on High Performance Distributed Computing*, 1995. [Duke University Technical Report 95-002].
21. Sanjeev Krishnan and L. V. Kalé. A parallel array abstraction for data-driven objects. In *Proc. Parallel Object-Oriented Methods and Applications Conference*, February 1996.
22. W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*, August 1995.
23. Joshua Yelon and L. V. Kalé. Agents: An undistorted representation of problem structure. In *Lecture Notes in Computer Science*, volume 1033, pages 551–565. Springer-Verlag, August 1995.

This article was processed using the LaTeX macro package with LLNCS style