Programming Languages for Modern Scientific and Engineering Computations *

L. V. Kalé
Dept. of Computer Science,
University of Illinois,
Urbana Illinois 61801
kale@cs.uiuc.edu

March 29, 1998

Abstract

With advances in theoretical understanding of physical processes, and availability of high-performance computers, computational methods in science and engineering have acquired a new prominence. A multitude of programming languages are being used to program the diversity of applications in computational science and engineering (CSE). Programming languages are like toolboxes which are used to construct an engineered artifact, namely the program. Choosing the right sets of tools can make a significant impact on the productivity of the software development process, and the quality of the end product. This survey briefly examines various categories of programming languages in use today, and the contexts in which they are useful.

*This work was supported in part by National Institute of Health (NIH PHS 5 P41 RR05969-04 and NIH HL 16059) and National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827EQ).

1 Introduction

Programming languages are often considered counterparts to the natural languages used by humans. Just as we use a natural language, such as English, to communicate with other humans, the argument goes, we use a programming language to communicate with the computer. However, this analogy is somewhat limited. Unlike human communication, the communication with the computer is essentially one-way: we specify a behavior using a programming language, and the computer executes that behavior.

More appropriately, one can think of a programming language as a toolkit that we use to construct an engineering artifact, namely the program. In this view, the standard libraries provided by languages, language compilers, and even the individual features supported by the language are tools within a well integrated toolbox.

With either metaphor, it is clear that the choice of programming language significantly affects the quality and speed of our programs, the efforts required to produce them, and even the software design process itself. In addition, the optimal programming language may vary from application to application. It is not surprising, therefore, that a panoply of languages have been designed, and are being used, for programming scientific and engineering applications.

The emergence of parallel computers, along with the expectation that the computers of the future will have to be parallel if they are to be orders of magnitude faster than today's computers, has fueled development of a new crop of programming languages. Most of these languages build on existing sequential languages by adding new parallel features to them.

This paper surveys the state of art in programming languages used for scientific and engineering applications.

2 Sequential languages

In addition to the evolution of traditional programming languages, the last decade has seen a rise in the use of mathematical modeling languages, scripting languages, and domain specific languages.

2.1 Traditional Languages: A shift towards C/C++, but FORTRAN holds its own

FORTRAN was one of the first high-level languages developed. Because of its simplicity and efficiency, it has remained one of the most popular languages for programming scientific and engineering applications.

FORTRAN-77 has an especially small set of language features, each quite easy to understand. This set of features is also well-suited for efficient compilation. For example, use of pointers, and existence of undetectable aliases, which present complex challenges to compilers for other languages, do not arise in FORTRAN programs. Also, compiler writers have been working on optimizing FORTRAN compilers for over four decades by now. These two factors allow FORTRAN compilers to produce code that can run at speeds close to the highest possible on a given machine.

FORTRAN was developed at a time when assembly language was about the only option available for programming computers. It supports global variables, arrays, the assignment statement, simple control constructs, and subroutines. Several initial versions and standards for the language were consolidated in the '70s, in a standard named FORTRAN 77.

One of the major critiques of FORTRAN has been that it does not facilitate the development of modular and maintainable programs, especially when the program had to be large. As computer science developed further understanding of programming languages, researchers and practitioners sought to extend FORTRAN to accommodate modern programming language features. FORTRAN 90, the resultant standard, supports high-level array operations, as well as recursive procedures, modules, pointers (but no pointer arithmetic) and dynamic allocation of memory. The large number of new features has made FORTRAN 90 appear substantially different from FORTRAN 77, although it was clearly meant as an extension. Partly as a result of this, and partly because Fortran 90 is still slower than Fortran 77, FORTRAN 77 continues to remain the most popular version of FORTRAN in use.

C was developed with the intent, at least initially, of supporting systems programming. In addition to providing access to low-level operations of the machine, C also supports user-defined data types, unrestricted pointers with pointer arithmetic and recursive procedures. For a variety of reasons, such as the speed and flexibility of C over its competetitors at the time, C became the language of choice for many application developers, especially for applications that were not dominantly numeric.

More recently, use of C++ has become quite widespread. C++ is a superset of C, in that C++ supports all features of C. Combining ideas of modularity from earlier languages like SIMULA with the efficient low-level features and well-established syntax of C, C++ is probably the most widely used language for modern applications, especially those involving graphical user interfaces.

There has been as slow but steady increase in the use of C/C++ for scientific applications. This is in part due to a newer generation of programmers, who have been trained in C/C++, and in part because C++ is believed to encourage development of modular and maintainable programs, with reusable components. Also, C++ is seen as suitable for expressing complex but efficient new algorithms, and data structures.

A criticism of C++ has been its poor performance. Not only was C++ seen as much slower than FORTRAN, it was also seen to be substantially slower than C itself. However, compiler vendors have started using advanced compiler technology to narrow this gap. Haney [18] has published results of several benchmarks showing that C++ code was several times slower than the equivalent C code, which itself was slower than FORTRAN. Arch Robinson, in an article [28] responding to the above, argues with new benchmarking evidence that better compilation techniques have narrowed, if not eliminated, the performance gap between C and C++.

Although the performance of C as well as C++ on small benchmarks remains weak compared with FORTRAN, there is accumulating evidence that entire applications coded in C++ often tend to perform better than their FORTRAN counterpart. As an example that I am personally familiar with, a molecular dynamics program coded in C++ [9] outperformed a similar program written in FORTRAN. Since the performance in the "inner loops" of the C++ program cannot be faster than FORTRAN, we infer that the performance gains of the C++ applications come from more efficient algorithms and data structures. Such improvements can also be made in the FORTRAN code itself, but it may be argued that the C++ language itself encourages programmers to think of more efficient algorithms explicitly.

Java is a recent language that is now being seriously considered for scientific applications. Although it is currently deployed largely for Internet-based applications, Java as a programming language is independently attractive for CSE applications. It has a familiar C++-like syntax, but is not tied by backward compatibility requirements to low-level languages such as C. It is perceived to be a "clean" object-oriented language, with a small subset of features of C++. Programmer productivity can be expected to be high when developing Java programs. One significant limitation of Java for CSE applications at the moment is speed. Java programs are typically compiled into machine independent byte-code, which is interpreted by the Java run-time system. Even with technologies such as just-in-time compilation, Java programs execute a few times slower than the corresponding C programs. As the compilation technology for Java matures, and especially if a few efficiency oriented changes are made to Java language, native-code compilers for stand-alone applications in Java will be able to produce efficient code with speeds comparable to C/C++. Under those conditions, the relatively clean nature of Java should make a larger number of compiler optimizations possible. Although currently Java exhibits prohibitive performance limitations, it can still be used *in conjunction* with C/C++. In this approach, one develops an entire application in Java, and then reimplements a few performance-critical parts in a more efficient language such as C++. This is feasible because Java explicitly supports interfaces to C/C++.

2.2 Integrated languages for mathematical modeling

A segment that has grown in popularity in the last decade involves interactive languages that combine attractive, easy to produce, graphics with powerful support for mathematical modeling. Typically, a high level programming language is supported, along with packages for symbolic or algebraic manipulation, matrix operations, and visualization. Examples of such products include Macsyma, Maple, Mathematica, and Matlab. The emphasis in such products is on ease of use. Typically, speed of execution is slower than direct programming in a language such as Fortran. System-defined primitives are precompiled and are competetive with the speeds of low-level languages. However, the interactive usage requires the system to interpret the user's program, rather than compile it, which slows the execution. These performance issues are being addressed by the vendors, as the systems mature, and performance can be expected to improve over time.

2.3 Scripting Languages

Many CSE applications are quite complex, yet provide flexibility for the user to use them in a variety of contexts. As a result, an individual run of such an application often requires a considerable amount of "configuration" set up, to customize the application for the particular simulation at hand. Such setup can be effectively specified via a scripting language. A scripting language is interpreted, rather than compiled, so a script can interact with a running program, or can even be changed while the program is running. In addition to controlling a running program, scripts are often used to pre-process and post-process data. Scripting languages support variables and simple control structures, and typically allow one to invoke application subroutines directly. Two broad categories of scripting languages are in use: application independent, and application-specific.

Application independent scripting languages, such as Perl, Tcl and Python, can be used to develop scripts for controlling any application. They support familiar, well-supported syntax and features. The utility of such scripting languages has increased in recent years due to the availability of (independently developed) software "components". Components are individual programs that carry out a specific data processing task, and are designed to be useful in a variety of different contexts. A script then serves the purpose of integrating the disparate components into a single application, and driving the execution of the application.

Application-specific scripting languages are developed "from Scratch" for each individual application (e.g. scripting language used with the well-known program "XPLOR", used for biomolecular modeling). As a result, their syntax and features can be tailored to the particular application. Language design is a complex activity, requiring forethought and significant effort. So, although a well-planned scripting language tuned to a particular application can be a valuable asset, it is equally true that many scripting languages developed in an ad hoc manner tend to be a hindrance than help. Given a choice, application independent scripting languages should be preferred for these reasons.

2.4 Domain Specific Languages and Libraries

Languages that are specially designed for individual application domains can often improve productivity substantially compared with the use of general-purpose languages. Domain specific languages may specialize themselves to relatively broad domains such as discrete event simulations, or relatively narrow ones such as specific kinds of numerical algorithms.

Ellpack [27] is a very high-level language designed to support elliptic boundary value problems. It incorporates a large collection of solution methods, and is implemented as an extension to FORTRAN. Typically, users simply specify the equation to be solved, the boundary data, and the names of the discretization method and the solution method to be used. The system automatically generates relevant code and links in the appropriate libraries. If necessary, the system allows the user the flexibility to use low-level FORTRAN code in parts of the program.

Discrete event simulation applications help in modeling physical systems where events happen asynchronously over time. Examples of such systems include city traffic, digital circuits, battlefields, and manufacturing plants. Example of Simulation languages include GPSS, Simscript, and Modsim. Simulation languages that exploit parallel computers are described elsewhere in this issue.

Domain specific languages have been developed for several other application domains, such as VLSI CAD, computer architecture, etc.

Instead of defining new syntax, and a compiler, for a domain-specific language, there is a new trend towards domain specific GUI based systems (also called problem-solving environments). These combine advantages of domain specific libraries, and interactive assistance to the users in defining the computational problem.

3 Parallel Languages

Applications in science and engineering often require large amount the computation power. Simulations of larger subsystems, with more detail, and for longer durations are almost always desirable in order to attain better understanding of natural systems, or for analyzing the performance of engineering designs. So it is not surprising that during the last decade, when the parallel computers started becoming available commercially, parallel CSE applications have become increasingly popular.

Historically, CSE applications requiring high-performance used traditional vector supercomputers, such as the powerful line of Cray machines. These machines were typically programmed using FORTRAN, extended with directives, statements or functions for vector operations. As parallel supercomputers started surpassing (or incorporating) vector machines, application developers had to make a transition to this new category of machines. The parallel machines were substantially different from the vector supercomputers, so the old programming paradigms had to be discarded or modified. The programming paradigms that have emerged can be broadly classified as shared-memory models, message passing libraries, data parallel languages, data-driven approaches, and other novel approaches.

The word "language" usually describes a language with its own syntax, and a compiler to translate it into machine language. Several new parallel programming paradigms are not languages in this strict sense. They provide a library, with a set of functions that can be called by user programs (their "Application Programming Interface", or API). However, using a library for writing parallel programs can sometimes dramatically change the programming paradigm — the style in which a parallel algorithm is expressed. For brevity, in this article we use the word "language" broadly to denote such libraries as well.

3.1 Programming for Shared Memory Multiprocessors

Shared memory symmetric multiprocessors are becoming ubiquitous. One can find them on desktops as 2-processor or 4-processor PCs, or workstations with up to 64 processors. One of the programming models used for programming such machines may be broadly called shared memory programming. In this paradigm, the user program creates a number of processes (also called kernel threads, or light-weight processes), their number being equal to or sometimes more than the number of available processors. All processes are allowed read and write access to the same global memory. In addition, processes may also have private memory that other processes cannot access. Shared memory allocation, locks and barriers are the main coordination primitives in these languages, in addition to a primitive for creating multiple processes. A *lock* can be used to achieve mutual exclusion, so that two processes do not simultaneously attempt to modify the same shared variable. A *barrier* is a synchronization primitive: a process that calls barrier is made to wait until all the other processes have arrived at the barrier.

Posix Threads (Pthreads) [25] is a recent standardization of this model on Unix machines. Similar models are supported on Windows NT and OS/2. In Pthreads, users *may* create many more threads (each mapped to one process) than the number of processors, but they are advised to create as many threads as the number of processors for computation-oriented applications.

The curent set of prevailing parallel machines shows a trend towards NUMA shared memory machines (e.g. Origin 2000, Convex Exemplar). These machines support a large number of processors at the cost of incurring non-uniform memory access (NUMA) times: some of the memory is local to the processor, and is much faster to access than the rest of the (remote) memory. The above model can also be used on NUMA machines. However, on NUMA machines, users must be careful to avoid memory access patterns involving a large number of accesses to small portions of remote memory. Instead, explicit copying of large chunks of data from shared memory into the private memory of the processor can be used for better performance.

For distributed memory machines (such as the IBM SP3), various software based shared memory schemes can be used. As software based schemes allow different processors to maintain copies of the same shared data, maintaining consistency of the shared data becomes an issue. A continuum of trade-offs between performance and functionality is being explored by different experimental systems. (e.g. TreadMarks[4] , and CRL[19]).

3.2 Message Passing

Message passing languages are used to program distributed memory computers. The earliest distributed memory computers, such as the NCube and Intel iPSC supported a relatively simple dialect of message passing. More sophisticated and powerful (and consequently complex) portable libraries such as PVM[14] and MPI[16] evolved subsequently. MPI is an industry standard in widespread use, while PVM has a loyal following, and finds being used especially on workstation clusters.

MPI contains over a hundred functions that users can call. Yet the basic message passing paradigm can be understood with just a small subset of these features: functions to send, receive, and broadcast data. In a distributed memory computer, each processor executes its own program, within its own memory. (Typically, the programs running on all the processors are identical). They communicate with each other by exchanging "messages". The message can be thought of as an arbitrary sequence of bytes (data) enclosed in a tagged envelope. The *send* function copies the data from the sending processor's memory into a message, attaches a user specified tag to it, and sends the message to the destination processor specified by the user. At the destination processor, the *receive* call can be used to copy the data from the message into the user's program memory. In its simple version, the receive call waits for a message with a specified tag. Variations of these calls support synchronous/asynchronous behavior. In addition, MPI supports user-defined data types, and "communicators" that (among other things) allow different library modules to use the same tags for their messages, without causing a mix up. PVM provides support for dynamic creation of new processes (presumably on new workstations), and functions for packing individual data types into messages.

Message passing is a powerful, relatively low-level, paradigm. One can construct highly efficient applications with it, because programmers can control the locality of accesses directly. At the same time, it requires a higher degree of effort on the programmer's part than the data parallel languages.

3.3 Data parallel languages and loop-based parallelism

The data parallel approach is based on the observation that subcomputations in CSE applications often involve identical operations on large amounts of data, typically stored in arrays. As these operations are naturally expressed using loops, we will also refer to this approach as loop-based approach to parallel programming. In this style, users write a program, typically in FORTRAN, just as they would write a sequential program. To exploit parallelism, they simply add specific compiler directives at strategically chosen places in the program. The quintessential construct that defines this approach is the *parallel loop*. When a user specifies, via a directive, that a particular loop is parallel, the system is free to execute different iterations of that loop using different processors. As multiple iterations may read and write data from the same variables, it is the user's responsibility to make sure that the iterations are independent when the loop is specified to be parallel.

3.3.1 Parallel Loops

Several vendors of shared memory machines (such as SGI, HP and DEC) had been supporting their own variations of loop based parallelism. Recently, a new standard called *OpenMP* [1] has emerged that is meant to subsume such variations. In addition to parallel loops, OpenMP supports parallel sections (where each processor executes the same section of code), and task parallelism.

OpenMP assumes that all the data is accessible from each of the processors. So, this approach, in its simple form, is used mostly on shared memory machines. This approach has been quite successful for a significant portion of applications, especially on machines with a small number of processors.

On NUMA machines with a large number of processors, the portion of a parallel loop being executed by a processor may need to access remote data frequently, leading to slow execution times. Advances in compiler technology, being implemented by several compiler vendors currently, are aimed at extending the success of this approach to large NUMA machines.

3.3.2 High Performance Fortan

High-performance FORTRAN (HPF) supports parallel loops while giving the users more control over the locality of data (i.e. which processor is to hold which piece of data), in order to scale up to large NUMA as well as distributed memory machines. To allow HPF programs to run efficiently on distributed memory machines, the user is allowed to specify the distribution patterns for each global array. HPF compiler then inserts appropriate communication calls (such as put/get primitives or send/receive primitives) while compiling the program for parallel machines. A typical approach used by HPF compilers is to analyse the communcation requirements of a parallel loop at runtime, carry out all the communication to get the data needed by iterations of the loop executing on each processor, and then run the loops without any communication. This allows the communication to be efficiently "bunched" in larger chunks of data. HPF is based on FORTRAN 90 and supports high level array operations, implemented in parallel.

It is important to remember that both OpenMP and HPF support several features in addition to parallel loops. We focused on parallel loops, as they are seen as a defining feature of both languages.

3.3.3 Restructuring compilers

The efficiency of a program is critically dependent on the fractions of loops identified as parallel. Making conservative assumptions would mean that significant portions of code will execute sequentially, at a much lower performance. However, in many applications, it may not be easy for the user to identify which loops are parallel. Also, the user may make mistakes in identifying loops as parallel, when they are not. Further, many loops that exhibit data dependences (so that the computations of an iteration depends upon that of an earlier iteration) can be restructured with some effort to eliminate the data dependency. Paralllelizing compilers are being developed to automatically identify parallel loops, and to restructure programs to increase the detectable parallelism. Recent examples of the success of this approach are provided by the Polaris system [7] and SUIF [3]. Some of the vendor-suppiled compilers (e.g. PFA from SGI)also carry out limited forms of restructuring transformations during compilation.

The data parallel languages work effectively for applications that are relatively regular. To attain good performance, one needs to develop the program while being conscious of the potential for parallelism in each loop. Such programs tend to perform better than automatically parallelized "legacy codes". A combination of improvements in compilers with such conscious programming can be used to attain good performance even on machines with a large number of processors.

3.4 New Approaches

In addition to the languages described so far, many novel (and sometimes experimental) languages are being used in parallel CSE applications. Need to support irregular control and data structures, ensuring reusability of parallel software components, and adaptive tolerance of run-time conditions are some of the motivations for the existence and utility of these languages.

Often, entities in a parallel program running on a distributed memory machine must wait for data from remote processors. To avoid making the entire processor wait for this piece of data, it is desirable if the processor were to execute some other useful computation during the wait. This can be accomplished if there are more than one actions (or subcomputations) that can be executed on a processor. Then, while one subcomputation is waiting for data, others can carry out useful work. The approaches described in Sections 3.4.1 and 3.4.2 activate user-defined entities (threads, objects or handlers) based on availability of data. Thus they can be called **data driven execution** based approaches.

3.4.1 Multithreading

This technique involves creating a number of *user level threads* on each processor. Each thread executes a user specified function. The difference between calling a function directly, and creating a thread to execute it is that with threads, the function's execution is delayed to a later time. (when the calling thread needs to wait for some data, for example). Also, a thread can suspend its execution and can be resumed at a later time. Thus, when threads are used with message passing, a thread may issue a `receive` call for a message with a specific tag. If the message is available, execution proceeds as usual. However, if the message is yet to arrive on the processor, the thread is suspended, and another ready thread is resumed. When the system notices that the message has arrived on the processor, it moves the waiting thread to the scheduler's pool of ready threads.

It is important to distinguish multithreading from the use of kernel level threads in small shared memory machines. (See Section 3.1) The user level threads used in multithreading are lightweight, in that suspending one thread and resuming another is relatively fast. Also, typically user level threads can be suspended only when the thread requests to be suspended (when it finds that the data it needs is not yet ready, for example), whereas kernel level threads are often suspended by the operating system at arbitrary intervals. User level threads *need not* run in parallel: they just take turns running on the same processor. The adaptive overlap between communication and computation created by multithreading can sometime lead to substantial improvements in performance.

Limitations of multithreading include the relatively high memory usage per thread, caused by the need to allocate a separate stack (usually several tens of Kilobytes) for each thread. In addition, context switching requires saving all the registers and is often substantially slower than a function call. The following subsections describe represent another approach to data driven execution.

3.4.2 Data driven objects and handlers

Languages based on data driven objects (such as Charm++[23] and ABC++[5] , for example) allow creation of a number of "objects" on each processor. An object, for the purpose of this discussion, is a collection of related data, and a set of subroutines that can access this data. These subroutines associated with an object are called the "methods" of the object, and the phrase "method invocation" is used to denote calls to such subroutines. There can be many objects which use the same set of subroutines, but with their own copies of the data. To distinguish between them, each object has an object ID that must be used when invoking methods on it. In a parallel environment, global object IDs are used that are valid irrespective of the processor on which they are used. Method invocations can be "sent" to objects on remote processors using their global object ID. On every processor, a scheduler uses a queue of waiting invocations (also called messages), and repeatedly selects a message from the queue, identifies the object it is addressed to, and executes the method identified in the message. As a result, no object is allowed to hold the entire processor waiting for a particular data, leading to an adaptive overlap between communication and computation. [22] was probably one of the first to use data driven execution on commercial parallel computers, although the idea itself existed in the earlier data-flow approach[2], and the Rediflow[24] project for parallel functional languages.

More recently, handler based approaches such as active messages [29] have been implemented. An active message encodes the name of a "handler" — a user-defined function of one parameter. The message is sent to a remote processor just as a regular message is. On each processor, a scheduler (either running in the background or invoked by the user explicitly) calls the handler for each arrived message, passing the message itself as a parameter. Originally, active messages were distinguished from message driven execution by the fact that active messages were designed to cause an interrupt on the remote processor on arrival. In contrast, recent variants of active messages are scheduler driven. Some systems such as Nexus[12] require each message to contain a global data pointer in addition to the handler. As the global data pointer is equivalent to the global object ID, this approach supports data driven objects.

These data driven approaches avoid the overhead of threads. This is because the scheduler does not need to create a thread for executing an object or handler. Instead, it executes the specified function (or method) itself, and only when that function completes, selects the next message. As a result, only one action is "in progress" at a time, and so extra bookkeeping (which is required for threads) is not needed. However, this comes at a cost to expressiveness: the objects or handlers themselves cannot suspend in the middle of their execution waiting for remote data, unless they are wrapped inside threads. This necessitates the use of "split-phase" programming style, where the request for remote data and the code to be executed when the remote data arrives are physically separated. An interesting compromise is provided by the language (an extension to Charm++) called *Structured Dagger* [17] .

3.4.3 Data parallel object-oriented languages and frameworks

This class of languages supports the data parallel model in the context of C++ based object-oriented programming. Distributed arrays are supported as templated classes with support for various data parallel operations. Compiler supported approaches such as the PC++ [13] use syntactic extensions for supporting HPF-style data distributions and data parallel operations. Frameworks such as POOMA[6] and Overture[10] use mechanisms provided in C++ itself (such as operator overloading and templates) and provide a rich library of operators useful in scientific/engineering applications.

3.4.4 Other approaches

Several additional languages are being used in parallel CSE applications. We will cite just a few examples. Linda[15] allows multiple parallel processes to communicate and coordinate by sharing a set of tuples: a tuple contains a sequence of data items. A process can deposit tuples in the shared tuple space, and request the system to retrieve tuples with specific characteristics. The language Cid[26] combines the notions of "future" with ideas in the earlier data flow language called Id, in an extension of C. Cilk[8] supports data driven execution, threads and remote function invocations. CC++[11] supports parallel execution within an object using threads, and remote method execution using "processor objects". Multipol [30] combines standard message passing with user invocable scheduler for executing handlers.

The languages and approaches described here overlap in their abilities and features. So, robustness of implementation and level of support as well as subjective preferences play a significant part in the selection of languages for an application. Recognizing that different languages may be better suited for different modules even within a single application, run-time frameworks such as Converse [20] support multi-paradigm interoperability.

4 Concluding Remarks

What languages will continue to be used in future CSE applications? It is clear that parallel machines will be a part of the landscape of computing in future. Automatic parallelizing compilers that can target scalable parallel machines, and data driven approaches that can deal with hard-to-paralllelize applications can be expected to grow in importance. Further, one can expect a growth in the development and use of domain specific languages, and scripting frameworks to flexibily utilize them. Whether or not the usage of technically "better" languages such as FORTRAN 90 or Java will increase in sequential computing is unclear at this point.

Given the breadth of the topic meant to be covered by this article, and the brevity required of it, it was not possible to do justice to the large number of programming languages available today, nor is the number of citations provided here fully adequate. To partially address this problem, a more detailed bibliography associated with this article is available at http://charm.cs.uiuc.edu/cseLanguages.html.

References

[1] OpenMP Home Page. Technical report, http://www.openmp.org, 1998.

[2] W. B. Ackerman. Data Flow Languages. *IEEE Computer*, 15(2):15–25, February 1982.

[3] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G.R. Gao. ABC++: Concurrency by Inheritence in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[6] Susan Atlas, Subhankar Banerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Marydell Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Supercomputing '95*, 1995.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 207–216, Santa Barbara, California, July 1995. MIT.

[9] John A. Board, L. V. Kalé, Klaus Schulten, Robert Skeel, and Tamar Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[10] D. L. Brown and W. D. Henshaw. Overture: an advanced object-oriented software system for moving overlapping grid computations. In *Proceedings of Computational Aerosciences Workshop*, August 1996.

[11] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993. ISBN 0-272-01139-5.

[12] 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.

[13] D. Gannon and J. K. Lee. Object oriented parallelism: pC++ ideas and experiments. In *Proceedings of 1991 Japan Society for Parallel Processing*, pages 13–23, 1993.

[14] A. Geist, A. Beguelin, J.J. Dongarra, J. Weicheng, R. Manchek, and V.S. Sunderam. PVM 3 User's' Guide and Reference Manual. ORNL/TM 12187, Oak Ridge National Laboratories, Oak Ridge, TN, 1993.

[15] David Gelernter, Nicholas Carriero, S. Chandran, , and Silva Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, Aug 1985.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[17] Attila Gursoy and L. V. Kalé. Dagger: Combining benefits of synchronous and asynchronous communication styles. In H. G. Siegel, editor, *Proc. 8th International Parallel Processing*, pages 590–596, April 1994.

[18] Scott Haney. Is C++ fast enough for scientific computing? *Computers in Physics*, 8:690–694, 1994.

[19] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[20] 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.

[21] 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.

[22] L. V. Kalé and W. Shu. The Chare Kernel base language: Preliminary performance results. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 118–121, St. Charles, IL, August 1989.

[23] L.V. Kalé and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[24] R.M. Keller, F.C.H. Lin, and J. Tanaka. Rediflow Multiprocessing. *Digest of Papers COMPCON, Spring'84*, pages 410–417, February 1984.

[25] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with PThreads, Second Edition.* SUN Microsystems Press, Prentice Hall, 1998.

[26] Rishiyur S. Nikhil. Parallel Symbolic Computing in Cid. In *Parallel Symbolic Languages and Systems*, 1995.

[27] John R. Rice and Ronald F. Boisvert, editors. *Solving Elliptic Problems Using ELLPACK.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[28] Arch Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10:458–462, 1996.

[29] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[30] Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Katherine Yelick. Run-time support for portable distributed data structures. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Rensselaer Polytechnic Institute, NY, May 1995.