Programming for Exascale Computers

William Gropp *Fellow, IEEE* and Marc Snir *Fellow, IEEE*

***Abstract*—Exascale systems will present programmers with many challenges. We review the parallel programming models that are appropriate for such systems and the challenges that implementations of those models face on an exascale system. We discuss the feasibility of using existing programming systems, thus preserving the investment in legacy applications, as well as the benefits and likelihood of new programming models and systems.**

***Index Terms*—Concurrent, distributed, and parallel languages, Simulation Languages.**

I. Introduction

Computing at exascale and beyond will present many challenges: Programs will need to control billions of threads, running on cores with different architectures; good power management will be essential; applications will need to use less communication and less memory, relative to the amount of computing; failures will be more frequent, possibly including silent errors; and power management and error handling will cause different parts of the system to run at different speeds. New programming models may be required to handle these challenges.

We discuss in this paper parallel programming models, and their ability to handle these challenges. We focus on *programming models*, as distinct from *programming systems*: A parallel programming model provides "a set of abstractions that simplify and structure the way the programmer thinks about and expresses a parallel algorithm" [1]; a programming system is an implementation of one of more models. Thus, message passing is a programming model; MPI [2] is a programming system that supports the message-passing model, as well as other models, such as remote direct memory access (RDMA).

A programming model needs a *performance model* that estimates the performance of a program as a function of input and platform parameters. The performance model is approximate and has no formal definition, but is essential: With no such model, a programmer has no insight into the likely performance impact of a program change.

A good programming model for performance computing should expose to the programmer those resources that have a significant impact on performance and that can be controlled by software; it should hide details that have a secondary impact on performance, are not under software control, or can be managed well by compiler and run-time.

A "machine-level" parallel computing model exposes the hardware in as direct manner as possible: A program executes on a platform consisting of a fixed number of nodes, each with a fixed number of physical threads, and a fixed amount of memory. The model specifies the communication mechanisms between threads — shared variables, within a node, and message-passing, across nodes. A program specifies the sequence of instructions executed by each physical thread. This model can be supported using MPI+OpenMP: Each node becomes an MPI process that executes an OpenMP program. This program uses exactly one work-sharing construct (parallel loop or section) to start an execution on each thread. The OpenMP program should use a fixed number of threads and affinity scheduling to have a fixed association of logical threads to cores. Typically, one will not use all physical threads so that system code can execute on separate resources. Also, it is often preferable to split a physical node into several MPI processes.

Higher-level programming models virtualize resources and use a run-time layer to map "virtual" entities (e.g., threads or variables) to physical entities (cores or memory locations). This is done to facilitate programming and improve portability. The use of higher-level programming abstractions will also encourage or mandate the use of restrictive programming patterns that reduce the likelihood of errors or facilitate virtualization.

The remainder of this paper is organized as follows. We first discuss some of the design choices for higher-level programming models. Section III presents those programming models that are broadly used now, and outlines how those can evolve to support exascale. In Section IV we discuss programming models that are now being researched and that have been proposed as the basis for a "revolutionary" approach to exascale programming. In Section V provides a summary of this paper.

II. Design Choices

A high-performance parallel programming model faces a number of design choices beyond those for a sequential programming model. We discuss five of the most important considerations here.

A. Scheduling

The mapping between logical execution threads and physical threads may be dynamic and managed by a runtime. In such a case, the programming model can accommodate a varying number of logical threads and operations that create (spawn) new threads or wait for their completion. How these threads are scheduled to run and whether they can move to different core or nodes can significantly impact performance. Some models, such as TBB [3] make a single scheduling decision when the thread is created. Others, such as OpenMP, for shared memory [4], or Charm++, for distributed memory [5], support thread migration at specific points in the thread execution (this is usually called *load balancing*).

One may also have *hybrid* models, for example, where logical threads are statically allocated to nodes but dynamically allocated to cores within nodes; MPI+OpenMP and DPLASMA [6] support such a hybrid model.

B. Communication

On current machines, communication takes significantly more time than computation. Communication includes communication between a cache and memory, communication across caches within nodes, and communication across nodes. Communication between caches and memory is traditionally managed implicitly — as a side effect of loads and stores. A simplified performance model will assume that access to memory is as fast as access to the L1 cache. Programmers ensure this approximation is valid by writing codes with good temporal, spatial and thread locality.

Not all algorithms can be expressed with good locality. Furthermore, the cost of associative caches (in silicon and energy) may lead to their partial or complete replacement, in future systems, by explicitly addressable scratchpads [7]. In addition, it may be difficult to support cache coherence on nodes with hundreds of cores. Therefore, core-to-memory communication and core-to-core communication are likely to become more software controlled and possibly exposed to the user. While compilers can easily generate explicit data move commands from a code with loads and stores, they have hard time optimizing these moves, for example, by aggregating them or by executing them collectively. In these cases, the cost of shared memory communication has to be part of the performance model.

Communication across nodes is typically under software control. For reasons discussed in Section IV, good temporal locality and spatial locality do not necessarily translate into efficient access to remote data; therefore, explicit user control of data movement is needed.

Communication can be *one sided*: it is effected by software running in one location. Examples are read and writes to local memory or *get* and *put* operations on remote memory. It can be *two sided* — with software on both communication locations involved, as for *send/receive* message-passing operations; and it can be *collective* with a group of locations jointly involved. Collective operations can be implemented efficiently in hardware, but will perform poorly in face of jitter (unsynchronized performance irregularities) [8].

Various performance models have been developed for internode communication. Communication across nodes is usually modeled by using the *postal model* (communication of m bytes takes time $a + bm$); or, occasionally, by using the more complex LogP model [9].

C. Synchronization

Two-sided communication is also synchronizing: it constrains the relative execution order of operations on distinct threads. One-sided communication is not synchronizing, and separate synchronization is required to enforce data dependencies. Both shared-memory and distributed-memory systems support synchronizing operations.

Most shared-memory synchronization operations, such as locks or atomic sections, are symmetric mutual-exclusion constructs: They ensure that different threads will not (appear to) execute concurrently operations or instruction blocks, but do not specify the order in which the threads will execute those. Distributed-memory synchronization operations, such as two-sided communication or barrier, are asymmetric, ordering constructs. The use of ordering synchronization constructs can eliminate nondeterminism and thus facilitate debugging.

A simple synchronization model is the *bulk-synchronous parallel* model (BSP), where the number of threads is fixed and all threads execute in synchronized phases, so that remote data consumed at phase i by a thread is produced by another thread at a phase $j < i$ [10]. A bulk-synchronous execution is easy to understand, as producer-consumer relations are synchronized by a global clock. This model is supported in MPI if all communications are collective, or if barriers are used to define phases, and sends executed at phase i are matched by receives that complete at end of phase i or at a later phase.

This model can be extended, while preserving the conceptual simplicity, to a *nested bulk-synchronous model* model, where threads can be split into teams that execute each under the BSP model [11]. MPI communicators can be used to implement this model. Dynamic nested parallelism also provides the same conceptual model, if a thread that spawns a team of child threads blocks until they complete and threads in a team only synchronize using barriers. This model is supported by suitably restricted OpenMP programs.

D. Data Distribution

Communication costs depend on the "home" of the data: where the data is stored when it is not actively used. Most distributed programming models provide user control on the data home. In an object-oriented system, such as Charm++, data is encapsulated in objects, together with methods; the location of the objects determines both where data is stored and where the methods are executed. The same, fixed association between storage location and loci of execution occurs in message-passing models and partitioned global address space (PGAS) languages, discussed in Sections III and IV. Some of the newer languages discussed in Section IV separate the two.

Communication costs will depend on proper collocation of data and operations on this data. One can approach this issue from a *data-centric* or *control-centric* view. In the first case (aka *data parallelism*), one focuses on data distribution and distributes computation to match the data distribution. In the latter case (aka *task parallelism* or *control parallelism*), one focuses on control distribution and moves data to where it's needed. Most current programming models encourage a data-centric view for distributed memory and a control centric view for shared memory. Some recent research (e.g., [12]) attempts to provide a more symmetric view, by facilitating both data and control migration. Note that whether one moves data to control or control to data, one will still have communication that is inherent to the parallel algorithm: data needs to move to data.

E. Global View vs. Local View

In a system with a *local view of control*, each physical thread executes its own (sequential) program. The programs are identical in a *single program, multiple data* (SPMD) model; they can be different, in a *multiple programs, multiple data* (MPMD) model. The multiple executions interact through communication and synchronization operations.

With a *global view of control*, one program uses parallel control statements (such as a parallel loop) or parallel data operations (such as a vector operation) to specify activities that may happen concurrently on multiple physical threads. The *single instruction, multiple data* (SIMD) model achieves parallelism with sequential control and parallel data operations.

Similarly, one can have a *local view of data*, where each thread has its own local data structures; or a *global view of data*, where aggregate data structures such as arrays span multiple nodes; a *data distribution* function specifies which part of the array is stored where. In the first case, a remote array location is accessed by using a different syntax from a local one (e.g., `a(index)` and `a(index)[numproc]`, in Fortran). In the latter case, the same expression (e.g., `a[index]`, in UPC) can refer to a local or a remote location, depending on the distribution of array `a`.

III. Evolutionary Approach

Can we program exascale systems with our current approaches? Can the evolution of current approaches provide adequate support for exascale systems? To answer these questions, we discuss the current programming models and their likely extensions.

Because this section considers how an exascale system may be programmed with existing parallel programming systems, the focus is on programming systems rather than programming models.

A. Current Systems

Current programming models for parallel computing cover a wide range of approaches; the major ones are summarized in Table I. These examples show that one programming system, such as MPI, can support multiple programming models. In some cases, multiple programming systems may be used together — the most common such case is MPI and OpenMP.

B. Single System

Because an exascale system is likely to have distributed-memory hardware, one of the most natural programming systems is MPI. MPI is currently used on over 1 million cores with exceptional scalability. It therefore is reasonable to ask whether MPI can be used on an exascale system. In considering this question, the following issues are sometimes raised but are not valid problems:

- The amount of message buffer space. This does not grow as $O(P)$ if properly implemented [28]. For many applications, it only grows as $O(\log(P))$.
- Other MPI internal memory, such as the description of MPI communicators, also need not scale as $O(P)$. It is possible to trade a little bit of time for memory space [29]. Depending on the application's needs, the overhead may be as little as $O(1)$.
- The time to start MPI processes need not be linear in the number of processes; scalable startup systems already exist [30].
- The bulk-synchronous programming model is sometimes raised as unworkable on an exascale machine because of the extreme concurrency and asynchrony. Whether this claim is true or not, MPI implements more than the bulk-synchronous programming model.
- General all-to-all communication does not scale well, irrespective of MPI, and an algorithm that frequently use such communication is not scalable. MPI has introduced "neighbor" collectives to support sparse "all-to-some" communication patterns that are scalable ([2], Sections 7.6 and 7.7).

The "MPI everywhere" approach does face several challenges. These include:

- The MPI process model encourages programmers to make local copies of data or to use memory to improve performance (e.g., for halo cells in stencil codes). As exascale systems are likely to have relatively small amounts of memory per core, applications need to be very memory efficient. One possible solution for MPI programs is to make use of the direct access to shared memory introduced in MPI-3 [2], Section 11.2.3.
- While MPI provides enough support for programmers to implement efficient, scalable code, even in the presence of performance uncertainty, MPI is a low-level system that relies on the programmer to use it well. In addition, as a library, an MPI implementation has some extra overhead.

The challenges faced by MPI are not insurmountable. However, they may require significant effort both in building a scalable MPI implementation and from the programmer in using MPI in a scalable way.

Several other candidates have been suggested for a one-system-everywhere approach. OpenMP can be extended to distributed-memory systems [31], but distributed-memory implementations of shared-memory systems do not normally achieve acceptable performance [32]. PGAS systems such as UPC and Fortran may be implemented for an exascale system and are discussed in Section IV.

However, these approaches also have problems. Some operations are hard to scale; in other cases, descriptions are enumerated in nonscalable ways. In addition, these approaches have found limited success in practice, with few major codes using them, perhaps because performance requires attention to locality similar to what MPI requires.

C. Shared-Memory Programming

The programming system most often used for shared-memory parallelism in scientific codes is OpenMP. The scaling of OpenMP to hundreds of cores will require changes both in programming style and in the language itself. OpenMP

TABLE I
PROGRAMMING MODELS AND SYSTEMS THAT IMPLEMENT THEM

Programming Model	Example Programming Systems
Shared memory	
Dynamic scheduling, nested bulk synchronous	OpenMP [4], TBB [3], Cilk++ [13]
Dynamic scheduling, general synchronization	pthreads [14], OpenMP, TBB, Cilk++
Distributed memory	
Bulk-synchronous	BSP [15], MPI with collectives/barriers, X10 with clocks [16]
Static scheduling, two-sided communication	MPI point-to-point
Static scheduling, one-sided communication	MPI RDMA, SHMEM [17], UPC [18], Fortran [19]
Hybrid scheduling (static across nodes, dynamic within nodes)	MPI+OpenMP, DPLASMA [6]
Local view of data and control	MPI, Fortran
Local view of control, global view of data	UPC, Global Arrays [20]
Global view of data and control	OpenMP, Chapel [21]
CoProcessor/Accelerator separate memory	OpenCL [22], OpenACC [23], CUDA [24]
Domain-specific languages and libraries	PETSc [25], Liszt [26], TCE [27]

provides a pure control parallel model, and provides no mechanisms for controlling data distribution. Therefore, OpenMP codes cannot be mapped efficiently to a non-uniform memory architecture (NUMA) system. OpenMP provides many non-scalable synchronization constructs (locks, atomic sections, sequential sections) and tends to encourage fine-grained synchronization. Therefore, many OpenMP programs are written in a style that impedes scaling. Various proposals have been made for extensions to the OpenMP language and for new compiler and run-time techniques that can alleviate these problem [33], [34], [35]; some of the needed changes were made in OpenMP v4.0, in particular, support for thread affinity — but evidence is lacking that OpenMP will scale to hundreds of cores.

A possible alternative is to use of one of the PGAS systems that are described in Section IV as a shared-memory programming language. However, these systems, as currently designed and implemented, do not provide good support for load balancing.

D. Hybrid Systems

The scalability issues can be eased by using a hybrid system, and the easiest is one that follows the hardware architecture: MPI is used for internode parallelism and a shared-memory programming model for intranode parallelism. This model is often referred as MPI+X (e.g., MPI+OpenMP), and exploits the fact that MPI is designed to be compatible with threading. Such a model reduces the pressure to scale either MPI or OpenMP, reduces memory pressure as less data is replicated within each node, and can better utilize shared memory.

The biggest problem when mixing programming systems is that they will compete for resources, such as threads (e.g., for runtime progress), and for memory bandwidth (including effects of accessing memory via the internode interconnect). Though some efforts are considering this problem [36], much remains to be done.

Hybrid systems have demonstrated mediocre scalability, but it is hard to separate the intrinsic problems of such systems from performance problems due to the use of inappropriate programming patterns [37], [38], or inefficient implementations.

IV. NEW PROGRAMMING MODELS

We discuss in this section new programming models for high-performance computing that have emerged in recent years.

A. One-Sided Communication and PGAS Languages

Modern communication hardware increasingly supports RDMA, to reduce the amount of copying that may occur during communication and reduce communication software overheads. One-sided communication supports the PGAS programming model well. In this model, data can be either private (accessed only locally) or shared; shared data can be local or remote; access to private data may be somewhat faster than access to shared local data; and access to local data is much faster than access to remote data that requires RDMA.

The PGAS model can be supported by a library such as MPI, SHMEM [17], or Global Arrays (GA) [20] or can be supported by a language, such as UPC [18] or Fortran (the Co-Array features added in Fortran 2008) [19]. In the former case, remote accesses are explicitly done via function calls. In the latter case, they are implicit: The language distinguishes, by type, private variables from shared variables; and may distinguish, by syntax, local references from remote references.

PGAS systems differ in the way the global name space is organized. Some, such as SHMEM or Fortran, provide a local view of data. Others provide a global view of data. UPC and X10 [16] support one-dimensional block-cyclic array distributions. GA supports multidimensional distributions, with blocks of equal or distinct sizes. The Chapel language [21] supports shared maps (with dense or sparse index sets) and arbitrary, user-defined distributions. PGAS systems also differ in their control structures. Most (MPI, UPC, X10) provide a local view of control. UPC provides a collective `forall` statement that switches to a global view of control; Chapel emphasizes a global view of control.

MPI and UPC support a static scheduling model, with a fixed number of threads. X10 and Chapel support dynamic scheduling. In X10, tasks may be migrated from one process to another or spawned, asynchronously, on another process. In Chapel, the location where a statement is executed can be

controlled by the user to colocate the execution with a datum or a locale.

B. Discussion

A global view of data or of control facilitates the porting of sequential code: Loops are replaced with parallel loops, and arrays are distributed. On the other hand, it often leads to inefficient programming patterns: Since data location is not salient, it is easy to write codes with excessive accesses to remote data. Also, it is not always possible to distinguish at compile time local accesses to shared data from remote accesses, resulting in additional run-time overheads for shared-data accesses. Global control encourages a programming style where each synchronization is effected by joining all threads and then forking control again, resulting in unnecessary overheads.

Caching is essential to achieving performance in (hardware-supported) shared memory. Caching is made effective through temporal locality and spatial locality in the stream of memory references. PGAS language implementations use local memory as a cache to remote memory data; the caching is done by the runtime. Unfortunately, it is too expensive to run a global coherence protocol and hard for the compiler to analyze when a cached value becomes invalid. Therefore, the local buffered copy is often used only once, even if the code has good temporal locality. Fixed line sizes do not work effectively for internode communication, and compilers often fail to aggregate multiple remote word accesses into larger messages. In order to achieve good temporal and spatial locality, programmers often have to explicitly copy data, thus losing much of the convenience that PGAS languages have, as compared with one-sided libraries [39].

C. The Future

PGAS languages are still evolving; new features will be needed before they are ready for use at extreme scale. These features include:

Support for teams: Multiphysics codes often consist of multiple modules, each running on a disjoint set of processes; the modules may run concurrently, on disjoint nodes, or sequentially, on the same nodes. MPI provides communicators to support such codes; similar *team* facilities have been discussed for PGAS languages [40]. Teams can also be supported by hybrid MPI+UPC systems [41].

Multitasking and message-driven execution: Concurrent or simultaneous multithreading can be used to avoid idle time when cache misses occur: If a thread is blocked, waiting for a memory access, then another thread can use the CPU. In addition, one can use cache prefetching to avoid cache misses; prefetches can be generated by hardware for strided accesses and can be inserted by a compiler for irregular accesses.

Similar techniques are needed to hide internode communication latency. One can use nonblocking communication — for example, nonblocking gets, the equivalent of cache prefetch, However, due to the much higher latencies it is unlikely that prefetches could be compiler-generated. More conveniently and effectively, one can run more threads than cores and schedule threads dynamically. This can be done either with nonblocking threads that are activated when their input message arrives and terminate when they generate an output message [5], [42] or with longer-lived threads that block and are descheduled when they execute a blocking internode communication and rescheduled after the communication is complete [43]. Languages that combine PGAS and multitasking are studied in the DEGAS project [44].

It is expected that execution time and communication time will exhibit higher variances on exascale systems, because of the larger levels of parallelism, varying execution speed due to power management and fault handling, and irregularities in the codes themselves. Multitasking increases the asynchrony in computations and, therefore, our ability to cope with the increased variance.

Virtualization: While PGAS languages typically map locales to fixed physical locations, there is no inherent reason why this mapping could not change during execution. The Charm++ system periodically redistributes "chares" across nodes, to improve load balance. Data and computation migration can be done transparently by the runtime or under partial or full user control. Frequent, fine-grained, concurrent object migration, as envisaged in systems such as ParalleX [12], require a sophisticated and potentially expensive global virtual address space management infrastructure.

Hierarchical design: PGAS+multitasking provides a two-level programming model, analogous to MPI+OpenMP. As system size increases, storage and communication hierarchies become deeper. It is widely believed that exascale systems will require support for hierarchical programming models, with more than two levels: At higher levels of the hierarchy, load balancing actions become rarer, and communications are expected to be less frequent. The Hierarchically Tiled Array project provides an example of a programming model organized around a hierarchical data structure [45].

D. Domain-Specific Environments

A *domain-specific language* (DSL) has been defined as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [46]. DSLs are usually supported by source-to-source compilers that translate the DSL notation into code written in a conventional language that uses domain-specific libraries. DSLs can facilitate code development, maintenance, and porting, at the expense of the effort needed to develop the DSL and the programming tools needed to use it effectively; Therefore, DSLs are most effective when they serve a narrow domain with a large user community: The effort in the development of the DSL and its environment are amortized against the productivity gains of a large community. As an example, this article is written by using LaTeX, a DSL for generating documents: This is a narrow application with many users. This also emphasizes that the meaning of "domain" is usually not a specific science domain; rather, it is a mathematical or algorithmic domain. For example, Matlab can be viewed as a DSL for the domain of linear algebra. Many

DSLs, while often motivated by a particular science domain, implement some combination of data structures and operations on those data structures and derive their advantages by being specialized to those specific types of operations.

DSLs for various scientific computing domains that generate parallel code have been a subject of research for several decades [47], [48], [49], [27], [26]. Two obvious obstacles to their broader use is that the community of scientific HPC programmers is small and the performance of DSL-generated code is not always competitive with the performance of hand-tuned code. This first obstacle may be alleviated by developing technologies that facilitate the development of new DSLs [50]; the second, by using autotuning [51].

While DSLs are considered to be distinct from libraries and frameworks, that distinction is shrinking. A library such as PETSc [25] is in effect an embedded DSL, where a host language is extended with domain-specific constructs. The domain-specific knowledge can be built into a source-to-source translator that optimizes code using these extensions [52].

We expect DSLs to be part of the solution for exascale programming. We emphasize that DSLs do not replace general-purpose parallel programming environments — indeed, they depend on them. Furthermore, DSLs are more effective when they are more specialized; they are unlikely to cover the entire range of HPC applications.

DSLs and libraries are particular examples of "multilevel programming": The application scientist programs in a notation that is meaningful to her, while the performance programmer that implements the libraries programs at a level that is closer to the hardware. Mechanisms and tools that facilitate such a division of labor are a subject of intensive research [53], [54].

V. Summary

We have discussed approaches that may alleviate the concerns of scalability, low memory and high communication costs on exascale systems. We have not discussed so far heterogeneity, power management, and resilience. The reason for these omissions is that it is not obvious that new programming models will be needed to handle these issues.

We expect that one system (possibly, an evolution from OpenMP) will be used to program shared-memory nodes and handle the heterogeneities of core and memory architecture, with significant compiler help. OpenACC [23] is an early example of such extensions, allowing the user to specify which device will execute which code and controlling data movement across address spaces. As multiple accelerators become integrated with CPU cores on one node or chip, more general extensions will be needed. Portability across the accelerators provided by different vendors will be essential.

Programming for low energy use is mostly synonymous with reducing communication, which requires explicit user control of communication, as discussed in previous sections. Load-balancing run-times can be used to limit temperature and reduce energy consumption [55].

Much uncertainty exists about programming model requirements for resilience. It is possible — especially if silent errors can be avoided — that no new features will be needed [56]. Otherwise, a minimum requirement is that failures that occur frequently generate well-defined exceptions and leave the application in a well-defined state.

We have made little progress in last few decades toward the "Holy Grail" of one simple model that can be used to program at any scale. Perhaps this goal is neither feasible nor practical. Nevertheless we have slowly improved programming models and programming methodologies for HPC; the quest for better programming models must continue.

While there is much uncertainty about the models that will be used at exascale, there is much certainty about needed requirements: large number of threads, reduced communication and synchronization, tolerance for variance in execution time, etc. New codes, even if written with current programming models, must be written to satisfy these requirements, so that future ports do not require algorithm changes. In particular, this is important for newly written OpenMP code as it is natural to write such code without regard to locality and with fine-grain synchronization. Good programming practices are more important than good programming models.

Scaling up applications to exascale will require significant programming efforts, even if current programming models prove adequate. The handling of reduced communication and asynchrony will require new algorithms. Therefore, programmer productivity matters. Good programming models, good systems that implement them, and good programming environment for these systems can enhance programmer productivity. On the other hand, programming a top supercomputer will never be as easy as writing a Matlab program. Writing efficient programs for a complex platform is a difficult engineering endeavor that requires a lot of knowledge and experience; complex engineering tasks are not easy to automate.

Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and under Award DE-SC0004131.

We thank Gail Pieper for her careful review of this paper.

References

[1] M. C. Rinard, D. J. Scales, and M. S. Lam, "Jade: A high-level, machine-independent language for parallel programming," *Computer*, vol. 26, no. 6, pp. 28–38, 1993.

[2] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.0." http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[3] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Incorporated, 2007.

[4] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0." http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[5] L. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)* (A. Paepcke, ed.), pp. 91–108, ACM Press, September 1993.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, *et al.*, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), 2011*, pp. 1432–1441, IEEE, 2011.

[7] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth international symposium on Hardware/Software Codesign (CODES 2002)*, pp. 73–78, ACM, 2002.

[8] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, Nov. 2010.

[9] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel Algorithms and Architectures (SPAA'95*, pp. 95–105, ACM, 1995.

[10] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] E. D. Brooks III, B. C. Gorda, and K. H. Warren, "The parallel C preprocessor," *Scientific Programming*, vol. 1, no. 1, pp. 79–89, 1992.

[12] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An advanced parallel execution model for scaling-impaired applications," in *International Conference on Parallel Processing Workshops (ICPPW'09)*, pp. 394–401, IEEE, 2009.

[13] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[14] D. Buttlar, J. Farrell, and B. Nichols, *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, Incorporated, 1996.

[15] J. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947–1980, 1998.

[16] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, *X10 language specification, Version 2.3*, 2013.

[17] K. Feind, "Shared memory access (SHMEM) routines," in *Proceedings of Cray User Group Spring Meeting*, pp. 203–208, 1995.

[18] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[19] J. Reid, "The new features of Fortran 2008," *ACM SIGPLAN Fortran Forum*, vol. 27, no. 2, pp. 8–21, 2008.

[20] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[21] Cray, *Chapel Language Specification, Version 0.92*, 2012.

[22] Khronos OpenCL Working Group, "The OpenCL specification, version 1.2." http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2012.

[23] "The OpenACC application programming interface version 1.0." http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, 2011.

[24] "CUDA API reference manual, version 5.0." http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Reference_Manual.pdf, 2012.

[25] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang, "PETSc users manual revision 3.3." http://www.mcs.anl.gov/petsc/petsc-dev/docs/manual.pdf, 2012.

[26] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, *et al.*, "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, p. 9, ACM, 2011.

[27] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, *et al.*, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.

[28] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on millions of cores," *Parallel Processing Letters*, vol. 21, no. 1, pp. 45–60, 2011.

[29] D. Goodell, W. Gropp, X. Zhao, and R. Thakur, "Scalable memory use in MPI: A case study with MPICH2," in *Recent Advances in the Message Passing Interface - Proceedings of the 18th European MPI Users' Group Meeting (EuroMPI 2011)* (Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, eds.), vol. 6960 of *Lecture Notes in Computer Science*, pp. 140–149, Springer, 2011.

[30] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Recent Advances in the Message Passing Interface - Proceedings of the 17th European MPI Users' Group Meeting (EuroMPI 2010)",* (R. Keller, E. Gabriel, M. Resch, and J. Dongarra, eds.), vol. 6305 of *Lecture Notes in Computer Science*, pp. 31–41, Springer Berlin / Heidelberg, 2010.

[31] J. P. Hoeflinger, "Extending OpenMP to clusters," *White Paper, Intel Corporation*, 2006.

[32] C. Terboven, D. An Mey, D. Schmidl, and M. Wagner, "First experiences with Intel cluster OpenMP," in *OpenMP in a New Era of Parallelism*, pp. 48–59, Springer, 2008.

[33] B. Chapman and L. Huang, "Enhancing OpenMP and its implementation for programming multicore systems," in *Proceedings of the International Conference on Parallel Computing: Architectures, Algorithms and Applications (ParCo 2007)*, pp. 3–18, 2007.

[34] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective," in *Proceeding, 5th International Workshop on OpenMP (IWOMP 2009)*, pp. 79–92, Springer, 2009.

[35] A. Pop and A. Cohen, "A stream-computing extension to OpenMP," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pp. 5–14, 2011.

[36] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with Lithe," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, (New York, NY, USA), pp. 376–387, ACM, 2010.

[37] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *Proceedings of the 2000 ACM/IEEE Conference on High Performance Networking and Computing (SC2000)*, 2000.

[38] N. Drosinos and N. Koziris, "Performance comparison of pure MPI vs. hybrid MPI-OpenMP parallelization models on SMP clusters," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, 2004.

[39] J. Zhang, B. Behzad, and M. Snir, "Optimizing the Barnes-Hut algorithm in UPC," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, 2011.

[40] J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and G. Jin, "A new vision for Coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, p. 5, ACM, 2009.

[41] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with MPI and unified parallel C," in *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, 2010.

[42] 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 annual International Symposium on Computer architecture (ISCA 1992)*, pp. 256–266, 1992.

[43] J. Zhang, B. Behzad, and M. Snir, "Design of a multithreaded Barnes-Hut algorithm for multicore clusters," Tech. Rep. ANL/MCS-P4055-0313, MCS, Argonne National Laboratory, 2013.

[44] "Dynamic exascale global address space or DEGAS." https://www.xstackwiki.com/index.php/DEGAS.

[45] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, pp. 48–57, ACM, 2006.

[46] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[47] T. Ruppelt and G. Wirtz, "Automatic transformation of high-level object-oriented specifications into parallel programs," *Parallel Computing*, vol. 10, no. 1, pp. 15–28, 1989.

[48] E. N. Houstis, J. R. Rice, S. Weerawarana, A. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes, "PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms," *ACM Trans-*

actions on Mathematical Software (TOMS), vol. 24, no. 1, pp. 30–73, 1998.

[49] S. Husa, I. Hinder, and C. Lechner, "Kranc: a Mathematica package to generate numerical codes for tensorial evolution equations," *Computer Physics Communications*, vol. 174, no. 12, pp. 983–1004, 2006.

[50] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02–03, pp. 215–226, 2000.

[51] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)*, pp. 1–11, IEEE, 2009.

[52] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan, "Treating a user-defined parallel library as a domain-specific language," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pp. 105–114, 2002.

[53] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for implementing domain-specific languages," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pp. 143–153, IEEE, 1998.

[54] J. J. Willcock, A. Lumsdaine, and D. J. Quinlan, "Reusable, generic program analyses and transformations," in *ACM Sigplan Notices*, vol. 45, pp. 5–14, ACM, 2009.

[55] O. Sarood, E. Meneses, and L. V. Kale, "A "cool" way of improving the reliability of HPC machines," in *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, (Denver, CO, USA), November 2013.

[56] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," Tech. Rep. ANL/MCS-TM-332, Argonne National Laboratory, Mathematics and Computer Science Division, Apr. 2013.

William Gropp is the Thomas M. Siebel Chair in the Department of Computer Science, Deputy Director for Research for the Institute of Advanced Computing Applications and Technologies, and Director of the Parallel Computing Institute at the University of Illinois in Urbana-Champaign. He received his Ph.D. in Computer Science from Stanford University in 1982, and worked at Yale University and Argonne National Laboratory. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He is a Fellow of ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

Marc Snir is director of the Mathematics and Computer Science Division at Argonne National Laboratory and Michael Faiman and Saburo Muroga Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He received a PhD in mathematics from the Hebrew University of Jerusalem in 1979 and worked at New York University, Hebrew University and IBM Research. His current research focuses on software for extreme-scale computing. Snir is a fellow of AAAS, ACM, and IEEE.