

Programming for Exascale Computers

Exascale systems present programmers with many challenges. A review of appropriate parallel programming models offers both insight into the feasibility of using existing systems, thus preserving the investment in legacy applications, as well as a way to compare the benefits and likelihood of new programming models and systems.

Scaling up applications to exascale will require significant programming efforts, even if current programming models prove adequate: 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 memory relative to the amount of computing; failures will be more frequent and possibly include silent errors; and power management and error handling will cause different parts of the system to run at different speeds.

We discuss in this article 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 whereas a programming system is an implementation of one or more models. Thus, message passing is a programming model; the message passing interface (MPI)² is a programming system that supports that model as well as other models, such as remote direct memory access (RDMA).

A good programming model needs a performance model to estimate performance as a function of input and platform parameters. The performance model is approximate and has no formal definition, but it’s essential: without such a model, the programmer has no insight into a program’s likely performance. 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, aren’t under software control, or can be managed well by compiler and runtime.

A “machine-level” parallel computing model exposes the hardware in as direct a 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 memory. The model specifies the communication mechanisms between threads—either shared variables within a node or message passing across nodes. A program then specifies the instruction sequence that each physical thread executes. MPI+OpenMP supports this model: each node becomes an MPI process that executes an OpenMP program that uses exactly one work-sharing construct (parallel loop or section) to start an execution on each thread. The OpenMP program uses a fixed number of threads and affinity scheduling to have a fixed association of logical threads to cores. Typically, you

1521-9615/13/\$31.00 © 2013 IEEE
COPUBLISHED BY THE IEEE CS AND THE AIP

WILLIAM GROPP

University of Illinois at Urbana-Champaign

MARC SNIR

Argonne National Laboratory

don't use all physical threads so that system code can execute on separate resources. Also, it's often preferable to split a physical node into several MPI processes.

Higher-level programming models virtualize resources and use a runtime layer to map "virtual" entities (such as threads or variables) to physical entities (cores or memory locations). This is done to facilitate programming and 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. In this article, we'll cover the design choices made by several proposed exascale computing models.

Design Choices

A high-performance parallel programming model involves several design choices beyond those found in a sequential programming model. We discuss five of the most important considerations here.

Scheduling

The mapping between logical execution threads and physical threads can be dynamic and managed at runtime. In such a case, the programming model can accommodate a varying number of logical threads and operations that spawn new threads or wait for their completion. How these threads are scheduled to run and whether they can move to different cores or nodes can significantly impact performance. Some models, such as Thread Building Blocks (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*).

Hybrid models are also possible—logical threads are statically allocated to nodes but dynamically allocated to cores within nodes. MPI+OpenMP supports this type of model.

Communication

On current machines, communication between cache and memory, across caches within nodes, and across nodes takes significantly more time than computation. Communication between caches and memory is 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 code with good temporal, spatial, and thread locality.

But not all algorithms can be expressed with good locality. Furthermore, the cost of associative caches (in silicon and energy) can lead to their partial or complete replacement in future systems by explicitly addressable scratchpads. In addition, it might 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. Although compilers can easily generate explicit data move commands from code with loads and stores, they have a hard time optimizing these moves—for example, by aggregating or executing them collectively. In these cases, the cost of shared-memory communication has to be part of the performance model.

Communication can be *one-sided*, effected by software running in one location—for example, reads and writes to local memory or get and put operations on remote memory. It can also be *two-sided*, with software running at both locations involved—for example, send/receive message passing operations. Alternatively, it can be *collective*, with a group of locations jointly involved; such operations can be implemented efficiently in hardware but will perform poorly when faced with jitter (variations in execution speed due to background system activities or other causes).

Various performance models for internode communication follow the postal model (communication of m bytes takes time $a + bm$) or, occasionally, the more complex LogP model.⁶

Synchronization

Two-sided communication synchronizes the involved parties. One-sided communication doesn't synchronize, so separate synchronization is required to enforce data dependencies. Both shared- 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 won't execute operations or instruction blocks concurrently, but they don't specify the order in which the threads will execute. Distributed-memory synchronization operations, such as two-sided communication or barrier communication, are asymmetric constructs. The use of ordering synchronization constructs can eliminate nondeterminism and thus facilitate debugging.

A simple synchronization model is the bulk-synchronous parallel (BSP) model, in which 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 phase $j < i$.⁷ A bulk-synchronous execution is easy to understand, with producer-consumer relations synchronized by a global clock. This model is supported in MPI if all communications are collective or if barriers define the phases; any sends executed at phase i are matched by the receives that complete at the end of phase i or at a later phase.

This model can be extended (while still preserving its conceptual simplicity) to a *nested bulk-synchronous* model, which splits threads into teams that execute under the BSP model.⁸ MPI communicators can help implement this model. This model is also supported by suitably restricted OpenMP programs.

Data Distribution

Communication costs depend on the data's "home": where it's stored when it isn't 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 along with methods; the objects' location determines both where the data are 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.

Communication costs will depend on proper collocation of data and the operations on it. A data-centric view (data parallelism) focuses on data distribution and matches computation location to it. A control-centric view (task or control parallelism) 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. Recent research tries to provide a more symmetric view by facilitating both data and control migration. Regardless of whether you move data to control or control to data, you still have communication that's inherent to the parallel algorithm: data needs to move to data.

Global View versus 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 differ 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 can happen concurrently on multiple physical threads. The single instruction, multiple data (SIMD) model achieves parallelism with sequential control and parallel data operations.

Similarly, you can have a local view of data, in which 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 (for example, `a(index)` and `a(index)[numproc]`, in Fortran). In the latter case, the same expression—`a[index]` in Unified Parallel C (UPC)—can refer to a local or a remote location, depending on the distribution of array a .

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 need to look more closely at current programming models and their likely extensions.

Current Systems

Today's programming models for parallel computing cover a wide range of approaches; Table 1 summarizes the major ones.^{9–20} These examples show that one programming system, such as MPI, can support multiple programming models. In some cases, multiple programming systems can be used together, the most common being MPI and OpenMP.

Single System

Because an exascale system is likely to have distributed-memory hardware, one of the most natural programming systems is MPI, which is currently used on more than 1 million cores with exceptional scalability. It's therefore reasonable to ask whether MPI can be used on an exascale system. This question raises the following issues, which aren't necessarily valid problems:

- The amount of message buffer space doesn't grow as $O(P)$ if properly implemented.²¹ For many applications, it only grows as $O(\log(P))$.
- Other MPI internal memory, such as the description of MPI communicators, also doesn't

Table 1. Programming models and the systems that implement them.

Programming model	Example programming systems
Shared memory Dynamic scheduling, nested bulk synchronous Dynamic scheduling, general synchronization	OpenMP, ⁴ TBB, ³ Cilk++ ⁹ Pthreads ¹⁰ OpenMP, TBB, Cilk++
Distributed memory Bulk synchronous Static scheduling, two-sided communication Static scheduling, one-sided communication Hybrid scheduling (static across nodes, dynamic within nodes)	BSP, ¹¹ MPI with collectives/barriers, X10 with clocks ¹² MPI point-to-point MPI RDMA, SHMEM, ¹³ UPC, ¹⁴ Fortran ¹⁵ MPI+OpenMP
Local view of data and control Local view of control, global view of data Global view of data and control	MPI, Fortran UPC, Global Arrays ¹⁶ OpenMP, Chapel ¹⁷
Coprocessor/accelerator separate memory	OpenMP
Domain-specific languages and libraries	PETSc, ¹⁸ Liszt, ¹⁹ TCE ²⁰

need to scale as $O(P)$. It's possible to trade a little bit of time for memory space.²² Depending on the application's needs, the overhead can 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.²³
- The BSP model might not work well on an exascale machine because of its extreme concurrency and asynchrony. Even if this is true, MPI supports other models.
- General all-to-all communication doesn't scale well, irrespective of MPI, and an algorithm that frequently uses such communication isn't scalable. MPI introduced "neighbor" collectives to support sparse "all-to-some" communication patterns that are scalable.²

The "MPI everywhere" approach does face several challenges:

- The MPI process model encourages programmers to make local copies of data to improve performance. Because exascale systems are likely to have relatively small amounts of memory per core, applications must be very memory efficient. One possible solution for MPI programs is to use the direct access to shared memory introduced in MPI-3.²
- Although MPI provides enough support for programmers to implement efficient, scalable code even in the presence of performance uncertainty, it's 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.

Although these challenges aren't insurmountable, they might require significant effort both in

building a scalable MPI implementation and 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, for example, but distributed-memory implementations of shared-memory systems don't normally achieve acceptable performance.²⁴ PGAS systems such as UPC and Fortran could be implemented for an exascale system and are discussed later, but these approaches also have problems. Some operations are hard to scale, and 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.

Shared-Memory Programming

The programming system most often used for shared-memory parallelism in scientific codes is OpenMP. But scaling it to hundreds of cores will require changes both in programming style and in the language itself. OpenMP provides a pure control parallel model, but no mechanisms for controlling data distribution. Therefore, OpenMP codes can't be mapped efficiently to a nonuniform memory architecture (NUMA) system. OpenMP provides many nonscalable synchronization constructs (locks, atomic sections, and 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 extending the OpenMP language and for new compiler and runtime techniques that can alleviate these problem. Some of these 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 a PGAS system as a shared-memory programming language. However, such systems, as currently designed and implemented, don't provide good support for load balancing.

Hybrid Systems

Scalability issues can be eased by using a hybrid system, the easiest one being a system that follows the hardware architecture: MPI is used for inter-node parallelism and some other shared-memory programming model for intranode parallelism. This model is often referred to as MPI+X, and it 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 are replicated within each node, and can better utilize shared memory.

The biggest problem when mixing programming systems is that they compete for resources, such as threads (for runtime progress), as well as for bandwidth (including accessing memory via the internode interconnect). Although some efforts have considered this problem,²⁵ much remains to be done.

New Programming Models

Fortunately, help is on the way, and new programming models for high-performance computing have emerged in recent years.

One-Sided Communication and PGAS Languages

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

The PGAS model can be supported by a library such as MPI, Shared Memory (SHMEM),¹³ or Global Arrays (GA)¹⁶ or by a language such as UPC¹⁴ or Fortran (specifically, the co-array features added to Fortran 2008).¹⁵ In the former case, remote accesses are explicitly done via function calls, whereas in the latter, they're 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. UPC and X10¹² support one-dimensional block-cyclic array distributions, whereas GA supports multidimensional distributions, with blocks of equal or distinct sizes. The Chapel language¹⁷ 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, and X10) provide a local view of control. UPC provides a collective `forall` statement that switches to a global view of control; Chapel emphasizes that global view. MPI and UPC also support a static scheduling model with a fixed number of threads, whereas X10 and Chapel support dynamic scheduling. In X10, tasks can be migrated from one process to another or spawned asynchronously on another process. In Chapel, the user can control the location where a statement is executed, to collocate that execution with a datum or a locale.

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. However, it can also lead to inefficient programming patterns: because data location isn't salient, it's easy to write code with excessive accesses to remote data. Moreover, it isn't always possible to distinguish at compile time local accesses to shared data from remote accesses, resulting in additional runtime overhead for shared-data accesses. Global control encourages a programming style in which each synchronization joins all threads and then forks control again, resulting in unnecessary overheads.

Caching is essential to achieving performance in (hardware-supported) shared memory. It's most effective when the stream of memory references have good temporal and spatial locality. PGAS implementations use local memory as a cache to remote memory data; caching is handled via runtime. Unfortunately, it's too expensive to run a global coherence protocol and too 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 don't work effectively for internode communication, and compilers often fail to aggregate multiple remote word accesses into larger messages. To achieve good temporal and spatial locality, programmers often

have to explicitly copy data, losing much of the convenience that PGAS languages have compared to one-sided libraries.²⁶

Future Requirements

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

Support for teams. Multiphysics codes often consist of multiple modules, each running on a disjoint set of processes; the modules can 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.²⁷

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 and waiting for a memory access, then another thread can use the CPU. In addition, cache prefetching can help avoid cache misses; hardware can generate prefetches for contiguous strided accesses, and a compiler can insert them for irregular accesses.

Similar techniques are needed to hide internode communication latency. Nonblocking communication—for example, nonblocking gets—is the equivalent of cache prefetching, but due to the much higher latencies, it's unlikely that prefetches could be compiler-generated. Something more convenient and effective is to run more threads than cores and then schedule those threads dynamically, either with nonblocking threads activated when their input message arrives and terminated when they generate an output message^{5,28} or with longer-lived threads that block when they execute a blocking internode communication and reschedule after the communication is complete.²⁹ The Dynamic Exascale Global Address Space (DEGAS) project studies languages that combine PGAS and multitasking.³⁰

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 our ability to cope with the increased variance.

Virtualization. While PGAS languages typically map locales to fixed physical locations, there's no inherent reason why this mapping couldn't change during execution. The Charm++ system

periodically redistributes “chares” across nodes to improve load balance. Data and computation migration can be done transparently via runtime or under user control.

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's 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.³¹

Domain-Specific Environments

A domain-specific language (DSL) is “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.”³² 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 required to use it effectively. DSLs are thus most effective when they serve a narrow domain with a large user community: the effort in the DSL's development are amortized against the larger community gains. As an example, we wrote this article in LaTeX, a DSL for generating documents: it's a narrow application with many users. A “domain” usually isn't a specific science domain; rather, it's a mathematical or algorithmic domain. For example, Matlab can be viewed as a DSL for the domain of linear algebra. While often motivated by a particular science domain, many DSLs implement some combination of data structures and operations on those data structures and derive their advantages by being specialized to certain operations.

DSLs for various scientific computing domains that generate parallel code have been a subject of research for several decades.^{19,20,33–35} Two obvious obstacles to their broader use is that the community of scientific HPC programmers is small and the performance of DSL-generated code isn't always competitive with the performance of

hand-tuned code. This first obstacle can be alleviated by developing technologies that facilitate the development of new DSLs,³⁶ and the second by using autotuning.³⁷

Although DSLs are considered to be distinct from libraries and frameworks, that distinction is shrinking. A library such as PETSc¹⁸ is in effect an embedded DSL, a host language extended with domain-specific constructs. The domain-specific knowledge can be built into a source-to-source translator that optimizes code using these extensions.³⁸

We expect DSLs to be part of the solution for exascale programming, but they don't replace general-purpose parallel programming environments—indeed, they depend on them. Furthermore, DSLs are more effective when they're more specialized; they're 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's meaningful to her, while the performance programmer implements the libraries' programs at a level that's closer to the hardware. Mechanisms and tools that facilitate such a division of labor are a subject of intensive research.^{39,40}

We've discussed approaches that could alleviate the concerns of scalability, low memory, and high communication costs on exascale systems. We haven't yet covered heterogeneity, power management, and resilience. The reason for these omissions is that it isn't obvious that new programming models will be needed to handle these issues.

We expect that one system will be used to program shared-memory nodes and handle the heterogeneities of core and memory architecture with significant compiler help. Version 4 of the OpenMP standard⁴ provides extensions for specifying which code will be offloaded to accelerators, and controlling the movement of data across address spaces.

Programming for low energy use is mostly synonymous with reducing communication, which requires explicit user control of communication. Load-balancing runtimes can help limit temperature and reduce energy consumption.⁴¹ Much uncertainty exists about programming model requirements for resilience. It's possible—especially if silent errors can be avoided—that no new features will be needed,⁴² but a minimum

requirement will be for failures that occur frequently to generate well-defined exceptions and leave the application in a well-defined state.

We've made little progress in the past few decades toward the “Holy Grail” of one simple model that can be used to program at any scale. Perhaps this goal is not feasible. Nevertheless, we've slowly improved programming models and programming methodologies for HPC, and the quest for better programming models must continue.

There's still a lot of uncertainty about the models that will be used at exascale, but there's definitely certainty about the needed requirements: large number of threads, reduced communication and synchronization, tolerance for variance in execution time, and so on. New codes, even if written with current programming models, must be written to satisfy these requirements, so that future ports don't require algorithm changes. This is particularly important for newly written OpenMP code, as it's natural to write such code without regard to locality and with fine-grain synchronization. Good programming practices are more important than good programming models.

CS
SE

Acknowledgments

This work was supported by the US Department of Energy, Office of Science, Advanced Scientific Computing Research, under contract DE-AC02-06CH11357 and under award DESC0004131. We thank Gail Pieper for her careful review of this article.

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, 1993, pp. 28–38.
2. *MPI: A Message-Passing Interface Standard Version 3.0*, Message Passing Interface Forum, 2012; www.mpiforum.org/docs/mpi3.0/mpi30-report.pdf.
3. J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, O'Reilly Media, 2007.
4. *OpenMP Application Program Interface Version 4.0*, OpenMP Architecture Rev. Board, 2013; www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
5. L. Kale and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based on C++,” *Proc. ACM SIGPLAN Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, A. Paepcke, ed., ACM, 1993, pp. 91–108.
6. A. Alexandrov et al., “LogGP: Incorporating Long Messages into the LogP Model—One Step Closer

- Towards a Realistic Model for Parallel Computation," *Proc. 7th Ann. ACM Symp. Parallel Algorithms and Architectures*, ACM, 1995, pp. 95–105.
7. L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM*, vol. 33, no. 8, 1990, pp. 103–111.
 8. E.D. Brooks III, B.C. Gorda, and K.H. Warren, "The Parallel C Preprocessor," *Scientific Programming*, vol. 1, no. 1, 1992, pp. 79–89.
 9. C.E. Leiserson, "The Cilk++ Concurrency Platform," *J. Supercomputing*, vol. 51, no. 3, 2010, pp. 244–257.
 10. D. Buttlar, J. Farrell, and B. Nichols, *PThreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly Media, 1996.
 11. J. Hill et al., "BSPLib: The BSP Programming Library," *Parallel Computing*, vol. 24, no. 14, 1998, pp. 1947–1980.
 12. V. Saraswat et al., *X10 Language Specification, version 2.3*, IBM, 2013; <http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf>.
 13. K. Feind, "Shared Memory Access (SHMEM) Routines," *Proc. Cray User Group Spring Meeting*, Cray User Group, 1995, pp. 203–208.
 14. W.W. Carlson et al., *Introduction to UPC and Language Specification*, Center for Computing Sciences, Inst. for Defense Analyses, 1999.
 15. J. Reid, "The New Features of Fortran 2008," *ACM SIGPLAN Fortran Forum*, vol. 27, no. 2, 2008, pp. 8–21.
 16. J. Nieplocha, R.J. Harrison, and R.J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *J. Supercomputing*, vol. 10, no. 2, 1996, pp. 169–189.
 17. Cray, *Chapel Language Specification, Version 0.92*, Cray, 2012.
 18. S. Balay et al., "PETSc Users Manual, Revision 3.3," ANL, 2012; www.mcs.anl.gov/petsc/petsc-dev/docs/manual.pdf.
 19. Z. DeVito et al., "Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers," *Proc. 2011 Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC11)*, ACM, 2011, article no. 9.
 20. A.A. Auer et al., "Automatic Code Generation for Many-Body Electronic Structure Methods: The Tensor Contraction Engine," *Molecular Physics*, vol. 104, no. 2, 2006, pp. 211–228.
 21. P. Balaji et al., "MPI on Millions of Cores," *Parallel Processing Letters*, vol. 21, no. 1, 2011, pp. 45–60.
 22. D. Goodell et al., "Scalable Memory Use in MPI: A Case Study with MPICH2," *Recent Advances in the Message Passing Interface Proc. 18th European MPI Users' Group Meeting (EuroMPI 2011)*, LNCS 6960, Y. Cotronis et al., eds., Springer, 2011, pp. 140–149.
 23. P. Balaji et al., "PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems," *Recent Advances in the Message Passing Interface Proc. 17th European MPI Users' Group Meeting*, LNCS 6305, R. Keller et al., eds., Springer, 2010, pp. 31–41.
 24. C. Terboven et al., "First Experiences with Intel Cluster OpenMP," *OpenMP in a New Era of Parallelism*, Springer, 2008, pp. 48–59.
 25. H. Pan, B. Hindman, and K. Asanovic, "Composing Parallel Software Efficiently with Lithe," *Proc. 2010 ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM, 2010, pp. 376–387.
 26. J. Zhang, B. Behzad, and M. Snir, "Optimizing the Barnes-Hut Algorithm in UPC," *Proc. 2011 ACM/IEEE Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC11)*, ACM, 2011, article no. 75.
 27. J. Mellor-Crummey et al., "A New Vision for Coarray Fortran," *Proc. 3rd Conf. Partitioned Global Address Space Programming Models*, ACM, 2009, p. 5.
 28. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM, 1992, pp. 256–266.
 29. J. Zhang, B. Behzad, and M. Snir, "Design of a Multithreaded BarnesHut Algorithm for Multicore Clusters," tech. report ANL/MCS-P4055-0313, MCS, Argonne Na'tl Laboratory, 2013.
 30. "Dynamic Exascale Global Address Space or DEGAS," 12 Feb. 2013; www.xstackwiki.com/index.php/DEGAS.
 31. G. Bikshandi et al., "Programming for Parallelism and Locality with Hierarchically Tiled Arrays," *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, ACM, 2006, pp. 48–57.
 32. A. Van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, 2000, pp. 26–36.
 33. T. Ruppelt and G. Wirtz, "Automatic Transformation of High-Level Object-Oriented Specifications into Parallel Programs," *Parallel Computing*, vol. 10, no. 1, 1989, pp. 15–28.
 34. E.N. Houstis et al., "PELLPACK: A Problem-Solving Environment for PDE-Based Applications on Multicomputer Platforms," *ACM Trans. Mathematical Software*, vol. 24, no. 1, 1998, pp. 30–73.
 35. S. Husa, I. Hinder, and C. Lechner, "Kranz: A Mathematica Package to Generate Numerical Codes for Tensorial Evolution Equations," *Computer Physics Comm.*, vol. 174, no. 12, 2006, pp. 983–1004.
 36. D. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, nos. 2–3, 2000, pp. 215–226.
 37. A. Hartono, B. Norris, and P. Sadayappan, "Annotation-Based Empirical Performance Tuning Using Orio," *IEEE Int'l Symp. Parallel & Distributed Processing*, IEEE, 2009, pp. 1–11.
 38. D.J. Quinlan et al., "Treating a User-Defined Parallel Library as a Domain-Specific Language," *Proc. 16th*

- Int'l Parallel and Distributed Processing Symp.*, IEEE CS, 2002, pp. 105–114.
39. D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," *Proc. 5th Int'l Conf. Software Reuse*, IEEE, 1998, pp. 143–153.
 40. J.J. Willcock, A. Lumsdaine, and D.J. Quinlan, "Reusable, Generic Program Analyses and Transformations," *ACM Sigplan Notices*, vol. 45, ACM, 2009, pp. 5–14.
 41. O. Sarood, E. Meneses, and L.V. Kale, "A 'Cool' Way of Improving the Reliability of HPC Machines," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis*, ACM, 2013, article no. 58.
 42. M. Snir et al., "Addressing Failures in Exascale Computing," tech. report ANL/MCS-TM-332, Argonne Nat'l 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 in the Institute of Advanced Computing Applications and Technologies, and director of the Parallel Computing Institute at the University of Illinois at Urbana-Champaign. His research interests

are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. Gropp has a PhD in computer science from Stanford University. He is a fellow of ACM, IEEE, and Society for Industrial and Applied Mathematics (SIAM), and a member of the National Academy of Engineering. Contact him at wgropp@illinois.edu.

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. His current research focuses on software for extreme-scale computing. Snir received a PhD in mathematics from the Hebrew University of Jerusalem. He is a fellow of American Association for the Advancement of Science (AAAS), ACM, and IEEE. Contact him at snir@illinois.edu.



Selected articles and columns from IEEE Computer Society publications are also available for free at <http://ComputingNow.computer.org>.

IEEE computer society NEWSLETTERS

Stay Informed on Hot Topics

COMPUTING NOW
TRAINING SPOTLIGHT
 TRANSACTIONS CONNECTION
 WHAT'S NEW BUILD YOUR CAREER
 IN COMPUTER CAREER COMPUTING CONNECTION
CS CONNECTION NEWS FLASH
 DIGITAL LIBRARY NEWS FLASH
 CONFERENCE CONNECTION
WHAT'S
 NEW IN COMPUTER
 BUILD YOUR CAREER
 MEMBER CONNECTION
 CONNECTION
 COMPUTING NOW
 TRAINING SPOTLIGHT
 CS MEMBER CONNECTION



computer.org/newsletters

