

# The Evolving Ecosystem for Parallel Software

In parallel computing, academic and commercial communities have long sought a language that could hide the complexity of parallel machines while providing high performance across machine scales. This ideal would lower the barrier to entry for parallel computing, make parallel machines more attractive as commercial products, and create a strong workforce for high-end scientific programming.

Most programmers have written applications in serial languages, leaving the high-end computing community relatively isolated in its attempts to exploit parallel machines.

In spite of a rich and varied landscape of parallel language projects and many demonstrated successes in both performance and usability, a single “silver bullet” language has remained elusive. After all, parallelism provides only a constant factor performance increase. Moore’s Law, meanwhile, has doubled serial performance every two years without requiring new programming models or other changes to software.

As a result, most programmers have written applications in serial languages, leaving the high-end computing community relatively isolated in its attempts to exploit parallel machines. In particular, the defense and scientific communities have invested in developing parallel software because of application challenges that require hundreds or thousands of processors and are compelling enough to make parallelism worth the effort. With the advent of multicore processors, however, parallelism has become a more pressing problem for the entire computing community (“The Manycore Revolution: Will HPC Lead or Follow?” *SciDAC Review* 14, Fall 2009, p40). Most computing fields increasingly expect machines with more features and higher performance, meaning the community of parallel programmers and programs will necessarily grow.

A new programming language needs more than technical innovations to gain widespread acceptance. The language must survive in a complex ecosystem that involves commonly available hardware, and existing software and programming tools. It must also garner a large enough set of algorithms and applications to draw a substan-

tial user community. In fact, a language that is overly innovative may be too difficult to implement efficiently. It may also outstrip the readiness of the user community to take advantage of new concepts. Instead, new languages must fill a need created by changing hardware or applications. It is not enough to be more advanced. The new language must offer improved usability or performance relative to previous languages.

During the transition from terascale to petascale, there was relatively little change in the ecosystem. Because the systems at both scales were built as networked clusters of commodity processors, the Message Passing Interface (MPI) with traditional serial languages (C, C++, FORTRAN) was the programming model of choice; variations of the Unix operating system were also popular. In contrast, the transition from gigascale to terascale required a major shift. Clusters soon replaced purpose-built vector supercomputers with their own operating system and compilers that supported automatic vectorization of serial code. Now, the first petascale machines have emerged, and already there are signs that the ecosystem is ready for another major change. As game and graphics processors become popular building blocks, multicore trends suggest that the memory per processor core will likely drop and support for remote memory access has become standard in interconnection networks. With these changes there is an increased interest in languages like Unified Parallel C (UPC).

UPC has succeeded in breaking into the high-performance computing ecosystem, giving application programmers direct access to some

# UPC Wins Multiple Awards in HPC Challenge

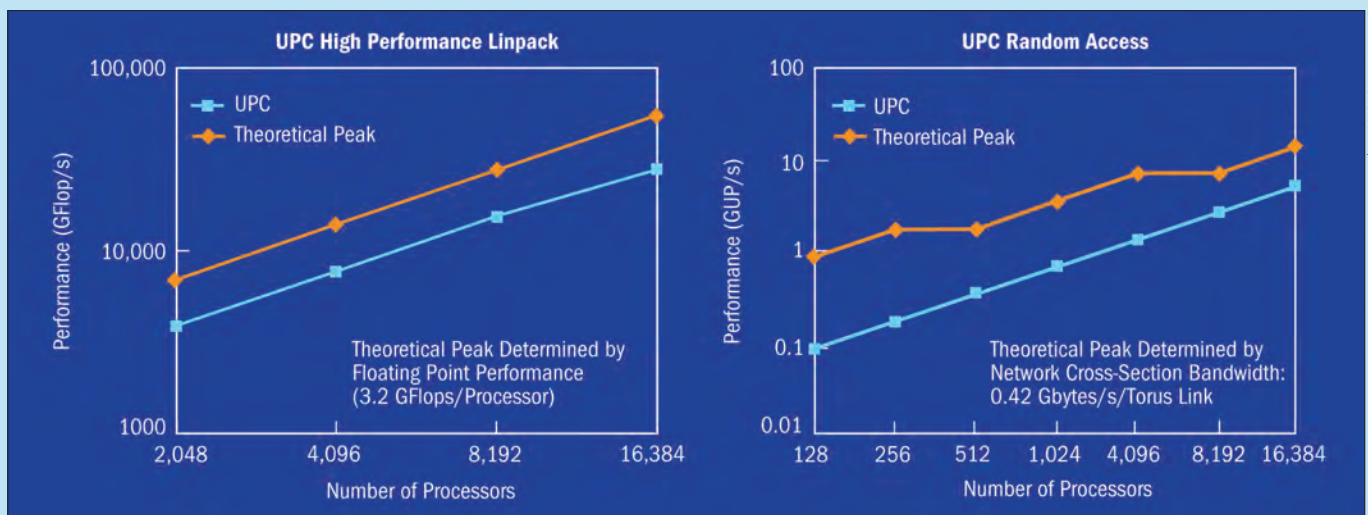
Since 2005 the IBM XLUPC compiler has participated in the class II High Performance Computing Challenge competition, held at the Supercomputing Conference (SC). The competition emphasizes both productivity and performance, looking for both elegantly written and scalable implementations of a set of four benchmarks. The IBM team took the (shared) first prize each of the three times it participated (2005–2008).

A team from IBM Research ported UPC to the Blue Gene/L computer in 2005 and demonstrated good scaling of selected benchmarks to the full size of the machine (128,000 processors). This significant achievement demolished previously held views that the high-productivity features of UPC, for example, the PGAS programming

model, caused inherently non-scalable, fine-grained communication patterns in UPC programs.

The IBM team, aided by previous experience with the High Performance Fortran language, had two key insights. First, compiler analysis can enhance access locality and coalesce remote accesses, therefore decreasing the volume and granularity of communication. Second, UPC language primitives map more or less directly into Blue Gene messaging hardware, allowing dramatically lower software overheads than previously thought possible.

The 2009 IBM entry in the SC competition (figure 1) features ports of the compiler to three of the company's flagship computers: Power 5 clusters, Blue Gene/P, and the Barcelona Supercomputing Center's MareNostrum.



**Figure 1.** IBM's 2009 High Performance Computing Challenge submission shows UPC High Performance Linpack (left) and RandomAccess (right) benchmarks on the four racks of IBM Research's WatsonShaheen Blue Gene/P installation. Like previous winning entries, this one demonstrates UPC can be scaled to thousands of processors while maintaining good performance.

hardware innovations and providing a uniform programming model for shared and distributed memory hardware. This uniform programming model should prove especially useful as the number of cores per processor continues to grow. UPC makes it easy to program certain kinds of algorithms with irregular memory access patterns over large datasets, an increasingly common problem in some data analytics problems in particular. It also offers performance (sidebar "UPC Wins Multiple Awards in HPC Challenge") that is competitive with message passing. As a result of these and other factors, a UPC implementation is now commonly required in high-end system procurements. This requirement will further aid in improving the portability and performance of applications written in the language and offer incentive for others to take the risk of experimenting with a new language.

## What is UPC?

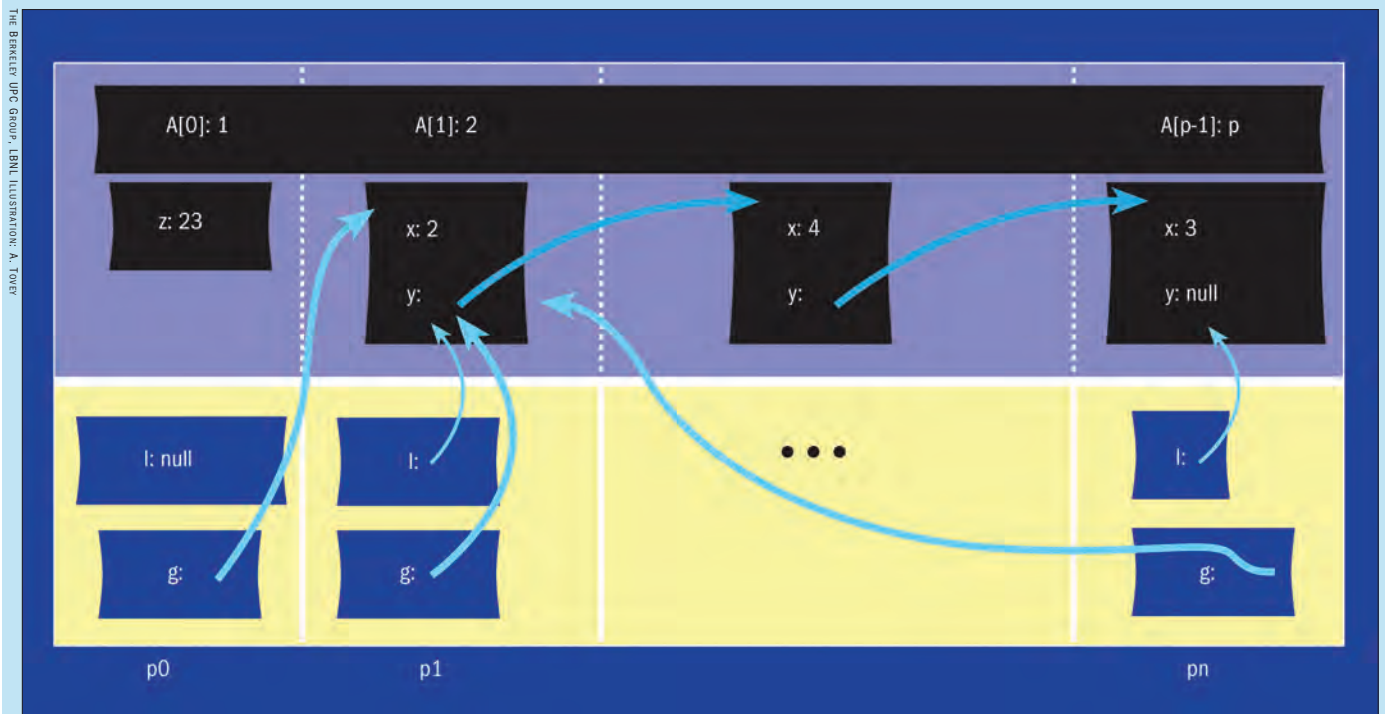
UPC was derived from three previous languages developed in the early 1990s: Split-C, PCP, and AC. These three came from the University of California (UC)–Berkeley, Lawrence Livermore National Laboratory, and the Institute for Defense Analysis, respectively. Two members from each of the language efforts met to design a unified language, which became Unified Parallel C, or UPC.

UPC is a proper superset of ISO C with a small set of language extensions, the most essential being the "shared" keyword for declaring shared variables. UPC is similar to programming with OpenMP or other shared memory languages, because any thread can directly read or write to any shared variable. But it differs from these shared memory models in that each thread has its own local, physical part of the shared memory space. This part is known to be nearby and there-

## UPC's Memory Model

The partitioned global address space in UPC is shown pictorially in figure 2. UPC's address space has two basic regions, a shared part (purple) and a private part (yellow). Variables in the shared space can be read or written by any processor, but each processor has its own partition of the shared space that is nearby (fast to access). All other partitions are likely to be slower to access.

UPC allows for simple shared variables like  $z$  and shared arrays like  $A$ , both of which are visible to all processors and live in the shared part of the address space. The type qualified as "shared" in the variable indicates which variables live in the shared space versus the private space. Shared linked structures such as trees or lists can also be built as shown and are created by dynamically allocating memory in the shared space. Private variables such as  $l$  and  $g$  live in a separate region of the memory space that is partitioned for each processor. The private space can contain simple variables and arrays, but an array in the private space lives entirely within one memory partition rather than being spread over several partitions, as is the case with shared memory. In this example, both  $l$  and  $g$  are pointers:  $l$  points only to local parts of the shared address space (in the same partition) whereas  $g$  can point to any partition. There are also methods of controlling array layouts and a set of allocation functions that can be used for dynamically allocated arrays and linked structures.



**Figure 2.** Unified Parallel C's partitioned global address represents a compromise between shared memory programming models and message passing. While some memory is shared, some data become associated with the processor most likely to access them.

fore fast to access, whereas the shared memory associated with other processors, while directly accessible, will likely be slower. Hence, the variables live in an address space that is global (every thread can see them) but partitioned (some parts of memory are faster than others). Languages that provide this abstraction of memory are therefore called partitioned global address space (PGAS) languages (for details on how PGAS works in UPC, see the sidebar "UPC's Memory Model").

The PGAS languages can be seen as a compromise between shared memory programming models like OpenMP or POSIX Threads and message passing models, such as MPI. Like shared memory models, UPC allows a program to directly access

remote data stored on another processor's memory without interrupting or coordinating with the application code on the other processor. This may spare the overhead costs associated with making copies of data that exist remotely and thereby minimize the amount of memory applications require. This is important because conserving memory space and bandwidth is becoming critical as the number of cores per chip multiply without concomitant increases in associated memory.

However, UPC makes no performance claims that remote data accessed multiple times in succession will have a faster access than the first access. In other words, UPC runtimes and the underlying hardware need not cache remote data,

so there is no global cache coherence required. If a request to read data comes from a remote processor, a network interface that does direct memory access operations may have to interact with a local processor cache. This two-way coherence, however, is much more practical than global cache coherence across a full machine.

The partitioned nature of the global address space allows UPC to run well on clusters and other distributed memory architectures. Like message passing programming systems, the data layout is under control of the programmer, not the compiler or runtime systems.

### **Expressiveness: UPC Means Never Having to Say “Receive”**

The recent emergence of multicore processors for general-purpose computing has increased interest in productivity of parallel languages. At the high end, the Defense Advanced Research Projects Agency’s High Productivity Computing Systems (HPCS) program emphasizes productivity of both machines and programming environments.

Although there is work in the HPCS program directed at measuring language productivity, the factors that make one language more productive than another are difficult to identify and even harder to quantify. Interestingly, all three of the HPCS languages (Chapel from Cray, X10 from IBM, and Fortress from Sun) share some aspects of the PGAS model, although they also differ significantly from UPC. UPC, being based on C, does not have the linguistic support of data abstraction, object orientation, exception handling, or implicit memory management that are often considered productivity features in modern languages. What UPC does offer is the global address space, which can be more convenient than message passing for certain irregular computations, allowing programmers to directly specify shared data structures and write expressions that contain accesses to remote parts of those data structures. In contrast, in a message passing program there are no shared data structures, only the local ones associated with each process. So, if a set of local structures logically make up a single shared one, that relationship is only in the programmer’s mind, not explicit in the code, and references to remote parts can only be done by asking a remote process for help. In this way, PGAS languages are more expressive than message passing ones, because they directly express shared data structures.

Many scientific applications, from climate modeling to ray tracing, require the use of a large table or other dataset of information that must be randomly accessed by the processors. In UPC, this dataset can be stored as a shared array, and any processor may perform a lookup operation simply by reading the

appropriate array element. In a two-sided message passing model, every send must be matched with a corresponding receive, which is executed by the application code on the processor that owns the table entry. The information about when a receive will be coming is not known until mid-execution, when the sending process needs information from the table. Such applications can be written in MPI but are not a natural fit to the programming model. Similar problems arise when computing histograms, hash tables, graph traversals, and any computation in which the communication event occurs at a time known only to initiating processor. In UPC, one never has to decide where to place receive operations: incoming read and write requests are handled below the application level, either by runtime software or by hardware.

The latency of fine-grained remote accesses across a petascale machine is orders of magnitude more expensive than local memory accesses, which are themselves orders slower than arithmetic and logical operations. But some applications require random access to an enormous dataset, and while the programming model should reveal that remote accesses are expensive, UPC does this through the type system, rather than ruling them out entirely. Dr. William Carlson of the Institute for Defense Analysis calls this UPC’s “Just do it!” feature.

### **Implementation: UPC is not HPF**

UPC has language support to control data layout in memory, including blocked and cyclic array layouts that are reminiscent of High Performance Fortran (HPF). The history of HPF, however, is not a happy one for application developers. Many of these developers were encouraged to perform large code rewriting efforts in HPF only to find that performance was disappointing and (eventually) compiler support was no longer forthcoming. This was a classic example of the failure of the ecosystem to sustain a programming model. Whether it was because the funding agencies abandoned the necessary compiler research before it came to fruition, compiler vendors did not support the R&D efforts, supercomputer centers avoided HPF in their procurement requirements due to fear that it would limit responses, or the application programmers had unrealistic expectations that led to overly complex language features – all of these factors fed the failure of HPF to gain widespread acceptance and eventually caused the language’s demise. As a result, the notion of a new programming model and one with distributed arrays strikes fear into application developers and architects alike.

While UPC does offer a global address space and array distributions, in most other aspects it is different from HPF. The parallelism model in UPC is a static Single Program Multiple Data (SPMD) model that typically corresponds to one

The partitioned nature of the global address space allows UPC to run well on clusters and other distributed memory architectures.

# An Approach to Portable Languages

The Berkeley approach to UPC language portability (figure 3) has two key components: (1) a source-to-source translator (rather than a full compiler that generates assembly code), and (2) a layered runtime system (which requires only the lowest layer for a complete functional port of the runtime).

## Source-to-Source Translation

The Berkeley approach uses a “translator” that turns UPC’s extended syntax into standard ISO C. That C code can then be compiled into the native language of a computing platform (assembly) using existing C compilers. This approach means that, given a C compiler, UPC programs can be ported to almost any machine, from a dual-core Macintosh up to a multiprocessor, multicore supercomputer. The compiled code calls on GASNet, a lightweight, easily portable runtime and communications system, to support UPC extensions.

## Layered Runtime System

GASNet is a custom-build language runtime layer that supports UPC’s special pointer types, distributed arrays, synchronization operations (barriers and lock), and the partitioned address space. For multicore and shared memory platforms, the runtime system uses either POSIX threads or processes with operating system-supported shared memory, so that *puts* and *gets* can be translated to simple *loads* and *stores*. But on machines with physically distributed memory, the runtime layer is a relatively thin layer over GASNet, which performs all runtime work involving communication. To avoid paying the overhead of function calls throughout the runtime system, many of the operations can be called through a macro expansion interface, that includes the necessary code in place.

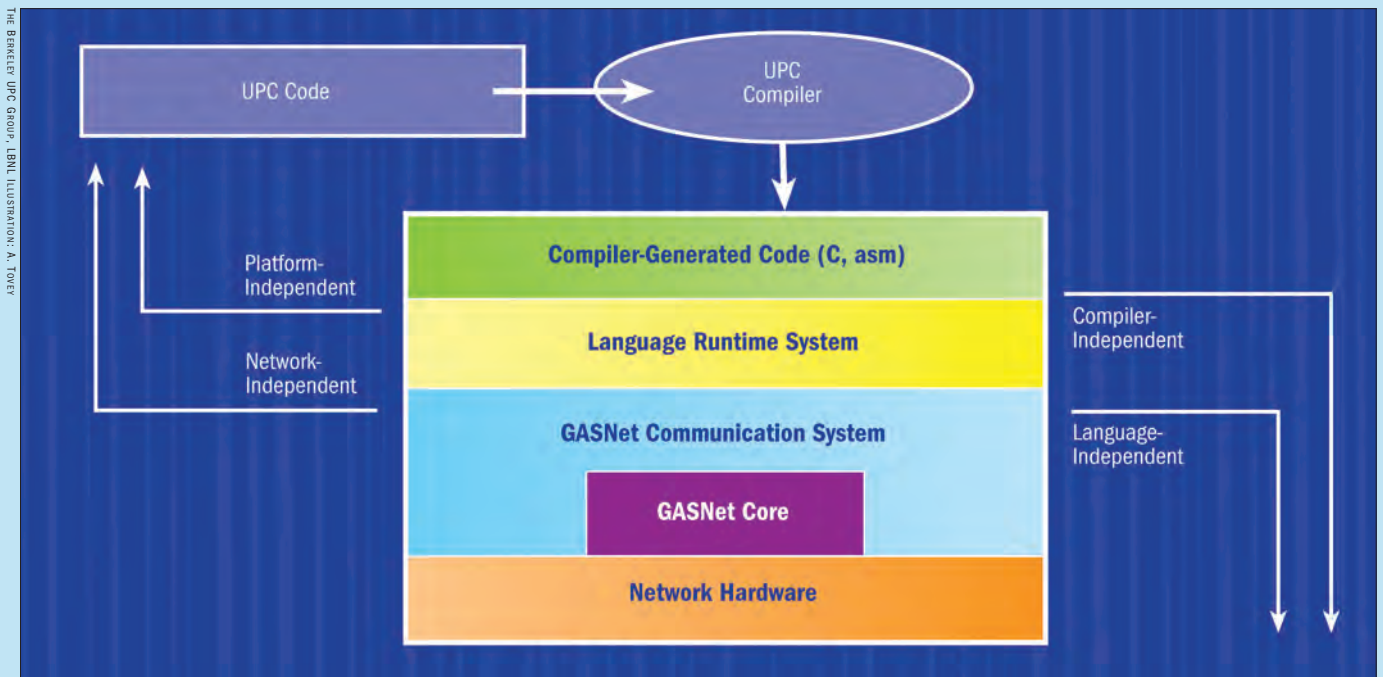
GASNet implements one-sided communication, starting with *put* and *get* operations on individual words and blocks of memory, along with

various synchronization mechanisms including barriers and locks. The goal of the GASNet design and the UPC language more generally is to expose the best possible communication to application programmers. For portability, there are supported implementations of GASNet on both Ethernet and MPI, which ensure that UPC runs anywhere. However, well-optimized implementations of GASNet for particular high-performance interconnect networks will perform significantly better.

The layers in GASNet, designed by Dan Bonachea of UC–Berkeley and Lawrence Berkeley National Laboratory, also encourage porting. New implementations can be developed using only a small subset of GASNet, known as the “core.” Some ports have been performed by groups outside the Berkeley team: The University of Florida built GASNet for the Dolphin interconnect, and SiCortex built one for their own custom networks. The Berkeley team has also built GASNet implementations for the Cray XT, X1E, and T3E platforms, the IBM BlueGene/P and SP, SGI Altix, InfiniBand, and Myrinet.

All of these optimized GASNet implementations build on the lowest public interface of the machine, not on top of MPI, giving the runtime access to lightweight communication. These implementations also take advantage of the direct remote memory access features in many networks, giving programmers access to the best performance of the network and encouraging network developers to provide low-latency, low-overhead networks.

In addition, the Berkeley team releases all sources for GASNet, including a reference implementation of the full GASNet interface based entirely in terms of the core. By implementing the core, one immediately has a full GASNet implementation. Berkeley releases GASNet runtimes for popular versions of operating systems, including Linux, AIX, Solaris, Mac OSX, and Cygwin on Windows.



**Figure 3.** Implementation layers in the Berkeley implementation of UPC. The compiler translates UPC code into C code that contains calls to the language runtime. The runtime is, in turn, built on GASNet, a communication substrate that is itself layered.

UPC thread per core or per hardware thread. The choice of which computations to run on which processors is entirely under programmer control, rather than being derived from the array layouts, as was the case under HPF.

It takes about one person-year to build a functional UPC compiler and existing runtime system, and less if the hardware supports a global address space. Because of UPC expressiveness, it is easier to write programs that require low latency networks, for instance, by not carefully controlling the placement of data or accessing remote data inside inner loops. Programs will run anywhere, but some programs will be very sensitive to underlying hardware performance. In this way, UPC actually encourages architectural innovation in networks and makes irregular parallel applications easier to write, which broadens the types of parallel applications that can and will be written.

### **Portability: UPC Can (and Does) Run Anywhere**

One of the early goals of the Berkeley UPC project was to demonstrate that UPC could be implemented on a broad range of architectures without an inherent requirement for specialized hardware support. At the same time, HP and Cray both undertook UPC compiler efforts for their high-end machine offerings, joined later by IBM. In addition, Intrepid, Inc. built a UPC compiler based on the open-source Gnu compiler called gcc. Michigan Tech University built a runtime system based on MPI that worked with a special version of the HP compiler. The Intrepid and Cray compilers were originally designed for machines with hardware-supported global addressing but were later modified to use the Berkeley runtime support so that they could run on a broader set of distributed memory architectures.

The Intrepid and Berkeley UPC implementations are entirely open source, which is a key support of the ecosystem: even if a group or company ceased work on UPC, there is some confidence that the language could still be supported, allowing cluster integrators to offer one of these compilers as their response to procurement calls.

Portability is also a critical feature of any programming model, as application developers need assurance that their codes will run on any current or future machine of interest. In addition, portability across machine scales allows programmers to develop code on a laptop or small cluster first before taking it to a computing center for scaling and production science. One cannot expect a language to run on any newly conceived hardware architecture right away, but compilers should be portable to new platforms with modest investments. Applications need not be rewritten to move from one machine to another. While applications may have to be tuned when moving from one system to another, an appli-

cation that is designed for locality (one that is aware of the partitioned memory space) and uses scalable algorithms will run well across machines. The Intrepid, MTU, and Berkeley compilers all offer functional portability (for a description of Berkeley's approach to UPC portability, see the sidebar "An Approach to Portable Languages")

The essential difference between GASNet and MPI's send/receive model is that GASNet supports one-sided rather than two-sided communication. Although MPI version 2.0 added one-sided communication, the Berkeley team explored the use of MPI's one-sided mechanism and found it unsuitable for a language environment, in large part because there were restrictions on when the one-sided operations could be used. In contrast, UPC programs can access any shared data at any time. For version 3.0, the MPI committee is considering a revised version of its one-sided support that tracks more closely with GASNet semantics.

The impact of the GASNet work has been felt far beyond the Berkeley UPC effort. The Intrepid UPC compiler also uses the GASNet structure, although it generates assembly code rather than C, leveraging the existing code generation support in the open-source Gnu C compiler, gcc. In addition, the Cray Chapel compiler, the Rice implementation of Co-Array Fortran (CAF), a research prototype for parallel Matlab-like language developed at the University of California–Santa Barbara, and the most recent commercial Cray compilers for UPC and CAF for their XT platforms use the GASNet communication layer.

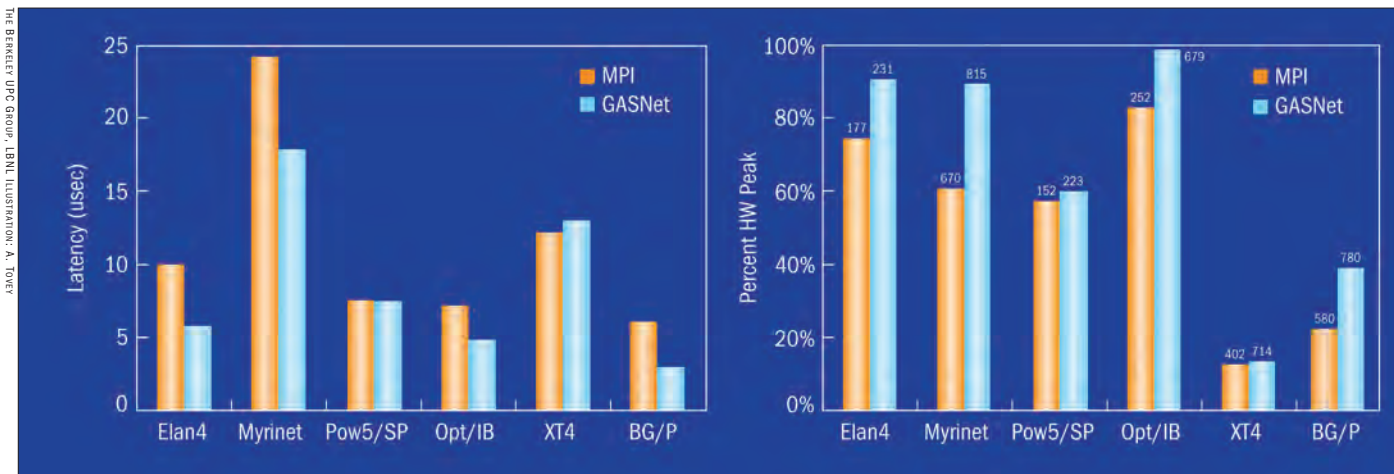
Research groups and instructors around the world use the Berkeley UPC implementation, and it runs on all of the major DOE Office of Science platforms.

### **Performance of UPC**

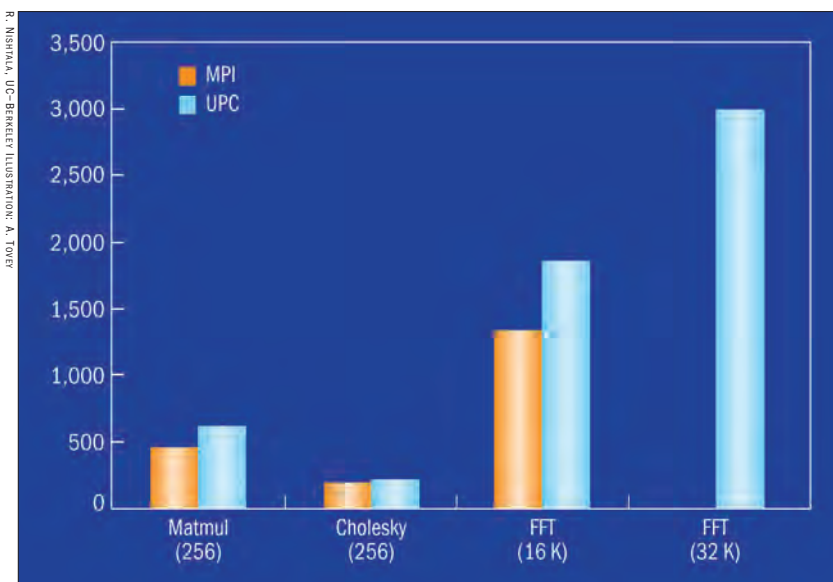
UPC was originally developed to make it easier for programmers to write irregular parallel applications. As a tool for high-performance computing, however, performance and scalability soon became paramount to the success of the language. From a performance standpoint, UPC's use of C as the base language is an advantage; it does not incur the runtime overheads associated with "productivity features" such as object orientation and automatic memory management. As with any language, performance can vary widely based on the quality of the compiler, but UPC's serial performance is typically as good as that of the C compiler, and even the Berkeley compiler with its extra level of translation from UPC to C has often proven competitive with others, because it can call the best C compiler available for generating serial code.

Performance of the parallel features of UPC depends strongly on the underlying hardware. Communication is based on one-sided communication,

One of the early goals of the Berkeley UPC project was to demonstrate that UPC could be implemented on a broad range of architectures without an inherent requirement for specialized hardware support.



**Figure 4.** These charts compare performance of communication primitives in GASNet against those of MPI. The left-hand graph shows the roundtrip latency of an 8-byte (small) message in both MPI and in the GASNet layer. The right-hand graph shows the bandwidth of both layers for a test in which 4-kilobyte (kB) messages are flooded into the network as quickly as the system permits. The 4 kB size is chosen to represent the bandwidth for medium-sized messages.



**Figure 5.** Performance of UPC versus MPI on matrix multiply, dense Cholesky factorization, and a 3D Fast Fourier Transform (NAS FT benchmark) all running on a BlueGene/P system using the Berkeley UPC compiler. The UPC code outperforms standard libraries written in MPI in each case.

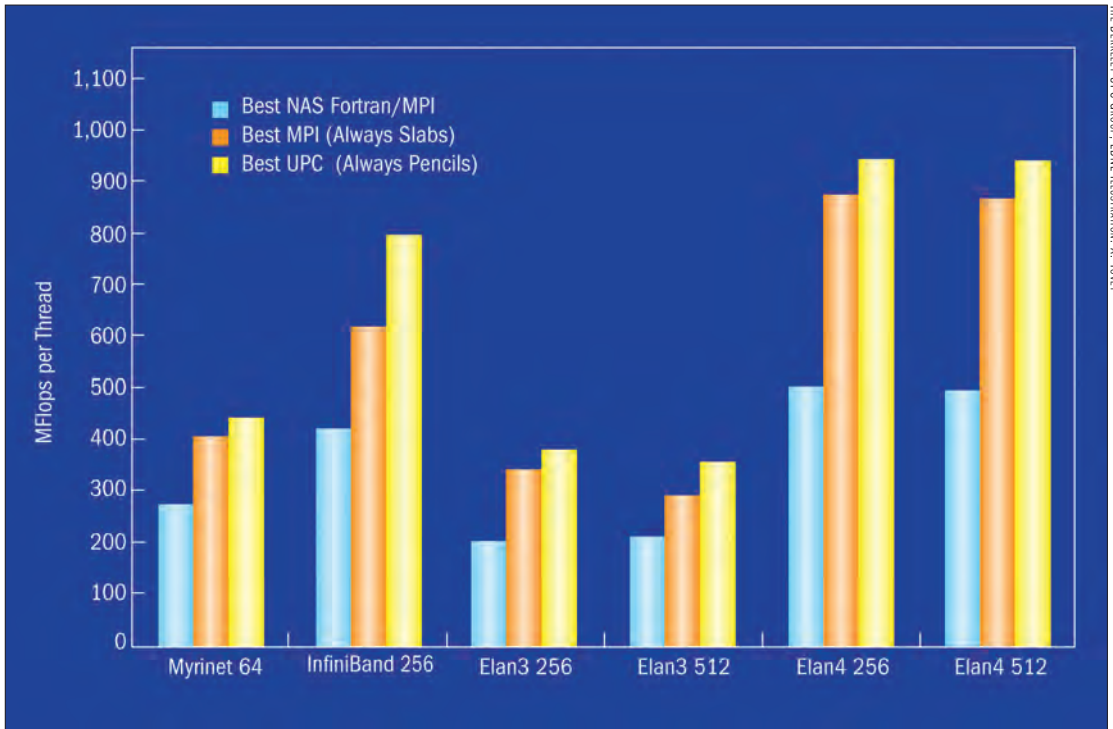
and figure 4 shows the comparison between communication primitives in GASNet compared to MPI. GASNet is often faster than MPI in terms of both latency and medium-sized message bandwidth (shown here for 4-kilobyte messages), in part because it does not have the implied synchronization or need for message tag matching that MPI has.

Applications in UPC scale to tens of thousands of processor and are also comparable to highly optimized benchmarks written in MPI. Figure 5 shows the performance of UPC on three benchmarks: Matrix Multiply and Cholesky factorization, which are compared to the MPI-based ScaLAPACK library, and the 3D FFT benchmark from the NAS Parallel

Benchmarks written in UPC compared to MPI. The FFT example is especially interesting, because it involves a global transport of the 3D array, which is limited by the bisection bandwidth of the network. While the hardware is identical in the 16K processor runs of the FT code, the UPC code is able to perform the transpose more quickly. In a more detailed analysis of the FT benchmark, the Berkeley UPC group implemented three different algorithms in both UPC and MPI, all using the same serial code from the FFTW library. This small-message algorithm is called a *pencil* algorithm, because it sends a row at a time, while the *slab* algorithm sends a contiguous set of rows, delaying the start of communication slightly until all of the local computation on rows is completed. The *bulk* algorithm is the more traditional approach used in bulk-synchronous programming: all computation on all rows is completed before any communication begins. All three algorithms have the same volume of communication, but the bulk algorithm sends the fewest and largest messages, while the pencil algorithm sends the most messages that are the smallest of the three; the slab algorithm is between these two extremes. In MPI, the fastest algorithm overall on each of the machines shown in figure 6 is the slab algorithm, while the UPC code performs best with pencils and, in doing so, outperforms MPI.

### Future Directions

The next several years are a critical time for high-end computing as the ecosystem adjusts to the hardware changes brought about by manycore and multicore systems. While clock speed increases prompted one of the three orders of magnitude in going from terascale to petascale, all predictions of exascale suggest that clocks rates will change very little in the next



THE BERKELEY UPC GROUP, LBNL ILLUSTRATION: A. TOVEY

**Figure 6.** Performance comparison of three implementations of the NAS FT benchmark across a range of cluster platforms using Myrinet, InfiniBand, and Quadrics/Elan interconnects. The first bar shows the performance of the reference NAS implementation, which uses a bulk-synchronous algorithm. The second bar shows the performance of three implementations: (1) bulk-synchronous, (2) overlapping at the level of slabs, which are a set of contiguous rows within one plane of the array, and (3) overlapping with pencils, which is a single row of a plane. For the MPI implementation, slabs are always the best performing of the three, while for UPC pencils always perform best. All three versions of the code, including the version labeled as NAS, use the same FFTW-based serial code.

decade, because power will be a larger concern than processor speed. UPC is well suited to systems built from large networks of shared memory multicore processors, because it makes efficient use of the on-chip shared memory and also exposes the lightweight communication primitives of the network to the applications. But if heterogeneous systems built from combinations of traditional processors and graphics processors or other accelerators become the dominant high-end architecture, then UPC will need to adapt along with other programming models. One challenging feature of graphics and game processors is partitioned memory, which uses software control over movement of data on and off chip and between processor cores, rather than hardware-managed caches with coherence protocols. This is conceptually aligned with the partitioned address space model of UPC, although the language and implementation will need to adapt in other ways to support the heterogeneous processors.

A new ecosystem will emerge over the next decade to support many low-power cores on a single chip, and there are already efforts at exploring new programming models or variations of current models for such systems. However, without a clear view of the architecture within a computational node, identifying an effective programming model

for that node is impossible. Whether UPC, or more generally the PGAS model, becomes part of that programming model remains to be seen, but the experience of the UPC language effort will be invaluable. The need for multiple implementations, including highly portable open-source compilers and runtime systems, is one key to portability of applications across multiple machine generations. Carefully titrated language features also seem important; disturbing the current programming models as little as possible, while ensuring enough new features to make the languages more productive or high-performing, will probably drive new hardware features in ways that motivate changes in the application development community. ●

**Contributors** Katherine Yelick, University of California–Berkeley (UCB) and LBNL; Paul Hargrove, LBNL; Costin Iancu, LBNL; Rajesh Nishtala, LBNL; Gheorge Almasi, IBM Research; Calin Cascaval, Qualcomm Research, formerly of IBM Research; Margie Wylie, Technical writer, LBNL

**Further reading**  
 UPC Community  
<http://upc.gwu.edu>  
 Berkeley Lab UPC  
<http://upc.lbl.gov>