Shared Memory Programming for Large Scale Machines

Christopher Barton[†], Călin Caşcaval[‡], George Almási[‡], Yili Zheng[††], Montse Farreras[‡‡]
Siddhartha Chatterjee[‡] and José Nelson Amaral[†]

[†] Department of Computing Science,
University of Alberta, Edmonton, Canada
{cbarton,amaral}@cs.ualberta.ca

[‡] IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
{cascaval, gheorghe, sc}@us.ibm.com

[††]School of Electrical and Computer Engineering
Purdue University, West Lafayette IN
yzheng@purdue.edu

[‡‡]Department of Computer Architecture
Universitat Politecnica de Catalunya, Barcelona Spain
mfarrera@ac.upc.es

Abstract

This paper describes the design and implementation of a scalable run-time system and an optimizing compiler for Unified Parallel C (UPC). An experimental evaluation on BlueGene/L®, a distributed-memory machine, demonstrates that the combination of the compiler with the runtime system produces programs with performance comparable to that of efficient MPI programs and good performance scalability up to hundreds of thousands of processors.

Our runtime system design solves the problem of maintaining shared object consistency efficiently in a distributed memory machine. Our compiler infrastructure simplifies the code generated for parallel loops in UPC through the elimination of affinity tests, eliminates several levels of indirection for accesses to segments of shared arrays that the compiler can prove to be local, and implements remote update operations through a lower-cost asynchronous message. The performance evaluation uses three well-known benchmarks — HPC RandomAccess, HPC STREAM and NAS CG — to obtain scaling and absolute performance numbers for these benchmarks on up to 131072 processors, the full BlueGene/L machine. These results were used to win the HPC Challenge Competition at SC05 in Seattle WA, demonstrating that PGAS languages support both productivity and performance.

Categories and Subject Descriptors Software [*Programming Techniques*]: Concurrent Programming

General Terms Performance, Experimentation

Keywords PGAS Programming Model, UPC, BlueGene

1. Introduction

With the advent of petascale computing, programming for large scale machines is becoming evermore challenging. Traditional languages designed for uniprocessors, such as C or Fortran, allow only the simplest kernels to scale to millions of threads of computation. When building solutions for real-life applications, understanding the problem and designing an algorithm that scales to a large number of processors is a challenge in itself. Thus, adequate programming tools are essential to increase the programming productivity for scientific applications. Initiatives like the DARPA High Productivity Computing Systems (HPCS) program [13] are encouraging industry and academia to take a fresh look at the issue of programming large scale systems.

This paper describes the design and implementation of a Run-Time System (RTS) and a compiler for Unified Parallel C (UPC) [7, 15] on the BlueGene/L machine. UPC is an example of the Partitioned Global Address Space (PGAS) programming model that embeds a few simple shared-memory primitives into C to enable a Single Program Multiple Data (SPMD) style of parallel programming. In the UPC execution model, all the threads are started before the user code begins. Threads are synchronized using barriers and locks. The PGAS memory model gives each thread access to a private section, a shared-local section, and a shared-remote section of memory. Threads have exclusive, low-latency, access to the private section of memory. The latency to access data in the shared-local section is typically lower than the latency to access data in the shared-remote section.

UPC provides two memory consistency models: a strict model and a relaxed model. Strict consistency is used to guarantee the ordering of memory references at thread level. Relaxed consistency is typically used for performance. The consistency model can be specified globally or on a per-access basis. The UPC memory and threading models can be mapped to either distributed-memory machines, shared-memory machines or hybrid (clusters of shared-memory machines). Our current implementation supports both a shared-memory mapping and a distributed-memory mapping, and it is available on the IBM alphaWorks [30].

BlueGene/L [16] is a distributed-memory machine that features as many as 65,536 dual-processor compute nodes, each operating at very low power, and hence at the relatively low frequency of 700 MHz. Designed for 360 Teraflop/s peak performance, the machine sustains 280 Teraflop/s when running the optimized version of the Linpack benchmark [29]. In addition to the original BlueGene/L installation at Lawrence Livermore National Labs (LLNL), there are now a number of smaller installations scattered across the globe.

The strength of BlueGene/L is its network — a $64 \times 32 \times 32$ 3D torus that spans all compute nodes. The default compute node software includes a compact kernel and a port of the MPI library, as well as the standard IBM Fortran, C and C++ compilers. It has been shown that careful programming and judicious use of MPI allow scaling of applications to the full size of the machine.

A long standing issue in high-performance computing is the productivity of efficient software development for high-end parallel machines. The expected increased dissemination of machines built

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

on a hybrid memory-access model compounds this problem. A hybrid memory-access model that consists of a collection of multiprocessor shared-address processing nodes connected through a message-passing fast network is likely to be dominant for high-performance computing in the near future. A programming language that is designed under a PGAS programming model, such as UPC, facilitates the encoding of data partitioning information in the program. Closing the gap between the programming and the machine models increases software productivity and results in the generation of more efficient code.

UPC implementations prior to ours were designed for SMPs or clusters composed of several hundred processors. Such a design point was eminently justifiable until recently given the lack of availability of parallel machines at the scale of BlueGene/L. Given that our implementation targets much larger systems, scalability was a prime concern in our design, which required us to deviate from certain design principles used in prior implementations. For example, we use a Shared Variable Directory (SVD) to keep track of the shared data, and thus avoid the possible memory fragmentation which would occur if shared data had to be mapped to the same address location in each node.

All improvements in the runtime are contingent on the ability of the compiler to exploit them efficiently. This paper describes new compiler optimizations that simplify the code generated for parallel loops in UPC by removing an affinity test from inside the loop whenever possible. The compiler eliminates several levels of indirection when it can prove that a segment of a shared array is local to the processor referencing it. Finally, the compiler makes efficient use of support for data-update operations implemented in the UPC RTS. When these operations are used, an update on a remote data item can be implemented through a single asynchronous message.

Thus, the main contributions of this paper are:

- a new UPC compiler and run-time system that allow scaling of UPC programs to more than a hundred thousand processors;
- the design and implementation of a distributed shared variable directory that solves the problem of addressing shared data in very large scale PGAS systems;
- demonstrate that productivity through the use of PGAS languages for large scale machines is possible. Using very simple, naïve implementations of two of the HPC Challenge benchmarks, we won the HPC Challenge Productivity Award [19], receiving the community endorsement for this work.

The rest of the paper is organized as follows. Section 2 describes the XL UPC compiler and UPC RTS. Section 3 describes the compiler optimizations implemented in the XL UPC compiler. Section 4 outlines the experiments and the results obtained from running the benchmarks on a BlueGene/L system. The related work is presented in Section 5 and finally conclusions and future work are discussed in Section 6.

2. Environment

We implemented a UPC compiler based on a development version of the IBM® XL Compiler framework. Utilizing this framework offers the advantage that the language semantics can be carried on from parsing, through different levels of optimization, all the way to the code generator. By contrast, source-to-source translators have to rely on the native compiler and the run-time environment for many low level optimizations. Experimental evaluation of global-address systems have shown that the single thread performance may vary dramatically [11].

2.1 XL Compiler Framework

The XL Compiler framework [23] has three main components: a Front End (FE) that parses different languages into a common intermediate representation (W-Code), the Toronto Portable Optimizer (TPO) – a high-level optimizer that performs machine-independent compile-time and link-time optimizations, and a code generator (TOBEY) that performs machine-dependent optimizations and generates code appropriate for the target machine. The XL UPC compiler uses all these components, of which only TOBEY is unmodified.

Figure 1 shows the role of each component in the compilation of UPC programs for a variety of platforms. The FE translates the UPC source to W-Code. To deliver a functional system early in the project, the FE translated UPC directly to calls to the UPC RTS. This path is still available in the XL UPC compiler and is shown as the left-hand-side path through TPO in Figure 1. Because of the direct translation, the compilation can bypass the TPO component and go directly to the code generation as shown with a dashed arrow. While this version of the compiler allowed for rapid prototyping, the performance of the generated code is not optimal. Specifically, when unmodified W-Code is used, each individual access to a shared variable has to be converted to an appropriate RTS call. This conversion has two implications for the optimizations that are performed. First, unless it can prove otherwise, the compiler must assume that the function calls have side-effects and therefore must be treated as kill-sites. This assumption reduces the scope of many data-flow optimizations such as copy propagation and common sub-expression elimination. Second, while the compiler will inline many of the RTS function calls, the inlining occurs late – after many of the optimization passes have run. Thus, the inlined code is not exposed to several data-flow analyzes and transformations that could successfully optimize it.

As a result of these performance limitations, the translation of the UPC code to calls to the RTS should be delayed until later in the compilation process. To facilitate this delayed translation, the XL UPC compiler enhances the intermediate language W-Code with several primitives to support UPC. The extensions to W-Code include the representation of shared variables, strict and relaxed attributes for memory accesses, and the `upc_forall` construct.

The FE uses the extended W-Code to annotates all the shared variables and other constructs with their UPC semantics. TPO processes the W-Code and performs optimization and translation, shown in the right-hand path through the UPC TPO in Figure 1. The advantage is that now all the optimization passes see the shared-array references as memory accesses and can apply all the classical code-optimizations available in TPO. These optimizations include existing link-time optimizations because the entire UPC RTS code is available as a library to the compiler. Although used in the reported results, these optimizations are not further discussed in this paper because they are not UPC specific. The UPC specific optimizations are discussed in Section 3.

2.2 Runtime System

The UPC RTS provides a platform-independent interface that allows compiler optimizations to be applied independent of the machine code generation. This interface can be implemented on a variety of platforms. A similar approach was followed in the GASNet runtime system [4]. We have implemented the UPC RTS interface on three different platforms: (1) shared-memory multiprocessors (SMP) using the Pthreads library [5]; (2) clusters of workstations based on the Low-level Application Programming Interface (LAPI) [27] library; and (3) BlueGene/L using the BlueGene/L message layer [1]. In this paper we discuss results only on the BlueGene/L machine. Most of the optimizations presented here are applicable to the other implementations.

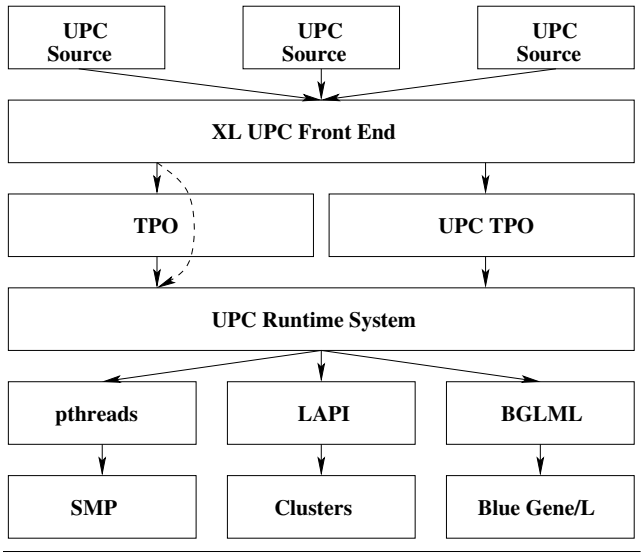

Figure 1. XL UPC Compiler and Runtime System

The UPC RTS exposes a few simple abstractions to the compiler, mainly derived from the UPC language features: a shared object handle, a shared object address, the thread affinity of a shared object, and an API to assign, dereference, and cast shared objects. The RTS also implements the UPC predefined functions and libraries.

Shared objects are a fundamental abstraction in UPC programs. Our RTS recognizes five kinds of shared objects: shared scalars; shared structures, unions, and enumerations; shared arrays; shared pointers with shared targets; and shared pointers with private targets. A transparent handle is used to refer to a shared object. These handles are kept internally, by the RTS, in a Shared Variable Directory (SVD). The UPC RTS provides routines to initialize and manipulate these handles. It is the responsibility of the compiler to manage the SVD entries when variables are created or go out of scope.

UPC shared objects have *affinity* to a thread, i.e., they reside in the shared memory section local to the thread. Shared handles point to a structure representation of a shared address that allows the program to reference shared objects anywhere in the partitioned global address space (essentially a fat pointer representation of the global address). The cost of accessing shared data through these handles is significantly higher than a simple dereference of a traditional C pointer. Therefore, if the compiler can statically determine the thread affinity of shared objects, it can convert these shared accesses to direct memory dereferences (C-like pointer accesses), and thus improve performance. A detailed discussion on the analysis required to determine thread affinity is presented in Section 3.

We designed the UPC RTS with extreme scalability in mind. The SVD is a partitioned data structure used by the RTS to manage allocation, deallocation, and access to shared variables. It is designed to scale to a large number of threads while allowing efficient manipulation of shared data. As opposed to other UPC implementations, we do not require that local sections of arrays be mapped to the same memory location in all the threads. Such restricted mapping can lead to unacceptable levels of wasted memory when amplified by the large number of processors we are targeting. Rather, like Titanium [31], we allow the local sections of a shared array to be of arbitrary length and rely on the RTS to do the bookkeeping. The SVD has the following design principles:

1. Threads must be able to create and destroy shared variables independent of each other and must keep the SVD consistent with a minimum amount of communication;
2. For collective operations, such as `upc_all_alloc`, when all the threads execute the same operation, no locking should be required;
3. No structure other than the SVD is allowed to keep pointers or references based on the number of threads; if remote information about a variable is required, the requester should get the information from the SVD. This in turn leads to a message exchange unless the SVD caches remote information to improve performance.

An example of the SVD for a distributed memory machine is presented in Figure 2. As specified by the UPC memory model, each thread owns a section of the memory (the shared-local portion) and also has a private section of the memory. The SVD consists of a two-level data structure: at the first level there is an array with `THREADS`+1 entries, where `THREADS` is the number of threads in an UPC program. Each entry points to a partition. This partition stores handles to shared variables that have affinity to the thread identified by `MYTHREAD`. For instance, handles for variables allocated using `upc_local_alloc` are stored in this partition. There is one extra partition, we call it the `ALL` partition, which is used for all statically declared non-scalar variables and for all variables allocated using `upc_all_alloc`. The reason for this separation is that, in a typical UPC program, there are many "globally" shared variables that belong to the ALL partition and only a few local shared variables. Moreover, different threads may have different numbers of shared variables. Using this design, the partitions can be resized independently when threads allocate shared data dynamically.

The example in Figure 2 shows the steps needed to access a shared array using the SVD. Given a shared-variable handle, the system locates the control block ("fat-pointer") for the shared array by dereferencing the entry in the SVD pointed to by the handle. The control block contains information about the layout of the array, such as blocking factor and element size, as well as information about the local section of the array, such as the local size and the local address. Given an array element specified by a global index, the layout information is used to compute the thread where the element is located, while the local information is used to compute the physical address of the element in the shared-local memory of that thread. While each thread has a copy of the SVD, the data stored in the control blocks are different, depending on the location of the variable in the thread's shared-local memory. Similar control blocks are defined for the other types of shared variables.

As shown in Figure 2, several levels of indirection are needed to address a shared variable. In Section 3.2 we discuss an optimization that reduces the cost of addressing shared variables when the compiler can determine the affinity of the shared-variable access. Other optimizations, such as caching the values of shared variables and the addresses of shared objects, are possible, but are not addressed in this paper.

UPC provides routines for dynamically allocated data, such as `upc_global_alloc`, `upc_all_alloc`, and `upc_local_alloc`. The SVD is designed to support both statically-declared shared variables and dynamically-allocated shared data. For example, `upc_all_alloc` is a collective operation that requires synchronization and communication between threads. In a machine where messages are not guaranteed to arrive in order, such as BlueGene/L, this requirement could increase the cost of accessing the SVD because of additional locking. We addressed this issue by having each thread manage its set of shared-local variables. Essentially, there is no constraint on the message ordering because operations on the SVD are "atomic". Each thread is responsible for updating the

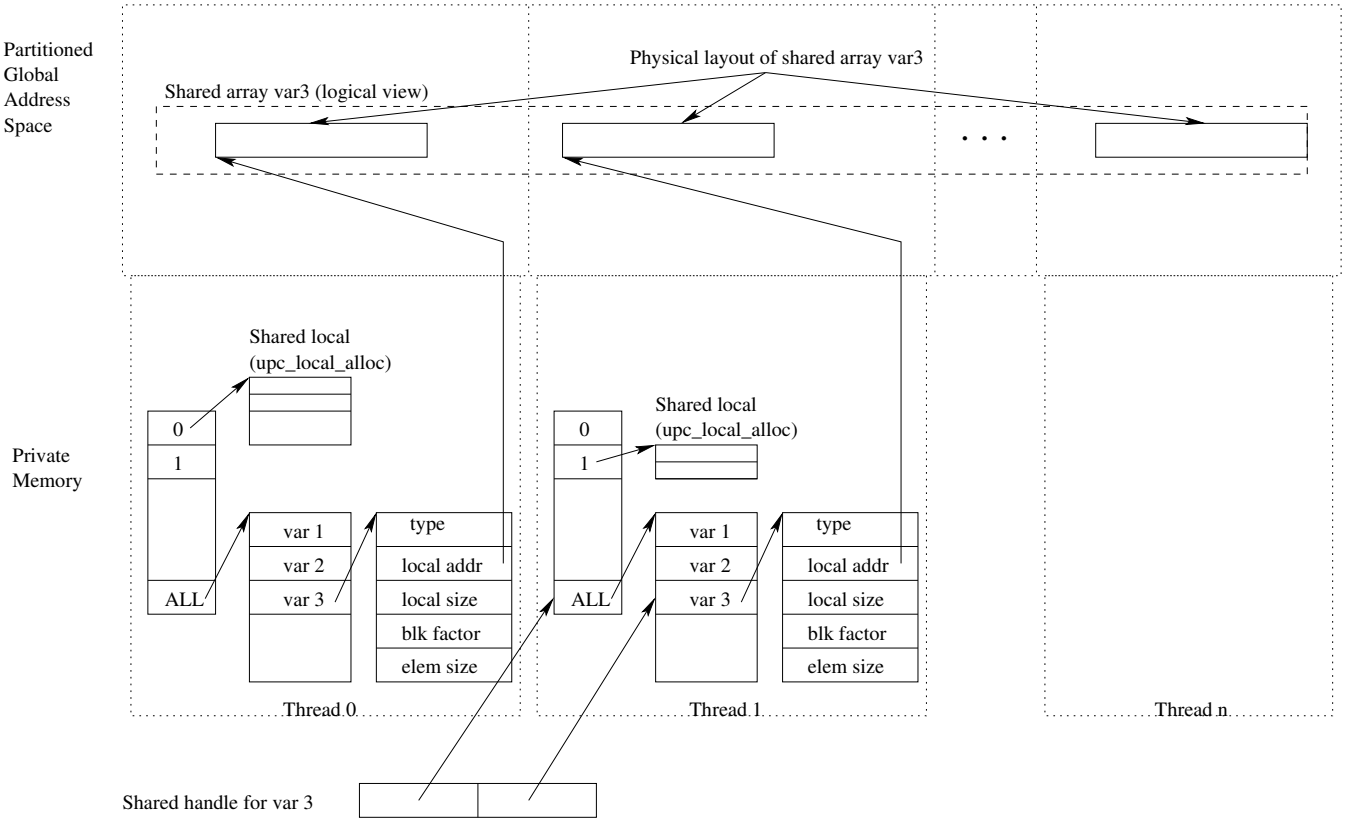

Figure 2. Shared Variable Directory in a PGAS distributed-memory machine.

SVD. A thread T_1 will not see the shared variables owned by thread $T_2 \neq T_1$ until its copy of the SVD is updated by T_2. The variables stored in the ALL partition are allocated through collective operations, and therefore guaranteed to be consistent.

2.3 Messaging Library

Although the UPC RTS was designed to leverage multiple types of hardware, such as shared-memory machines and the LAPI library, all measurements presented in this paper were made on BlueGene/L using a port of the UPC RTS to the BlueGene/L communication library.

The communication library was designed to support both MPI and lighter-weight communication paradigms. This communication library supports UPC, Global Arrays [24] and Hierarchically Tiled Arrays [2]. Our design target was low overhead and high scalability. This is how we achieved these goals:

- **Messaging library choice:** The relatively low ratio of CPU speed to network speed makes it imperative to send and process messages in as few CPU cycles as possible. Low overhead communication is very important for UPC performance, because non-optimized UPC code typically performs individual remote variable dereferences. These references result in very short network communications that are latency bound.

  Implementing the UPC RTS on top of the standard BlueGene/L MPI library would have caused unacceptably high latency because of the software overhead of starting, monitoring, and receiving an MPI message. Many of our communication library optimizations address this problem.

- **Packetization and network order:** The BlueGene/L network is packet-based with packets from 32 to 256 bytes long. Due to the way packets are routed in the BlueGene/L network, ordinary data packets can arrive out of order. Packets can be forced to arrive in order, but doing so with a large number of packets tends to create hot spots in the network. These hot spots decrease overall throughput.

  In practice any data transfers in UPC that require more than one packet of data have to be accomplished by handshake, resulting in long latencies. We use ordered packets for very short data communications, (*e.g.,* single-value `get`/`put` operations) to avoid the need to hand-shake with the receiver for the transfer of a 4-byte value.

- **Alignment issues:** The CPU-network interface is accessible only in 16-byte chunks. Moreover, each access has to be 16-byte aligned. Thus, when UPC buffers are arbitrarily aligned the messaging library has to copy data to and from aligned buffers during send/receive. This copying results in CPU overhead — memory copies are inordinately expensive on BlueGene/L— that translates into higher latencies for short transfers and into potentially decreased bandwidth when a node transmits and receives simultaneously. We mitigate the bandwidth problem by employing techniques already described in the context of the MPI library implementation [1].

- **Overlapping computation and communication:** One-sided communication is not directly supported by the BlueGene/L network hardware. All packets are inserted into/extracted from the network by the processor(s). Moreover, the same set of

double-wide floating-point registers are used by the machine to perform floating point computation and to talk to the network device, limiting the capability of a processor to compute and communicate at the same time.

Each BlueGene/L node features two processors. The original design point of BlueGene/L called for one of the processors to act as a communication processor, while the other performs computation. However, co-operation between the processors is limited by a lack of coherent view of the memory, making low-latency communication using a dedicated communication processor impractical.

Another way to achieve the effect of overlapping computation and communication would have been to interrupt the computation processor when a network packet arrives. However, switching to a network handler for every packet involves at least a context switch, with the added burden (compared to other machines) of saving and restoring a relatively larger number of registers, causing performance loss.

Ultimately, while running programs on BlueGene/L we noticed that most applications that were written to scale to a high number of processors tend to perform synchronized (and most often, collective) communication anyway. In hindsight, the problem of overlapping computation and communication seems not to be as important as it seemed.

- **Memory and scaling issues:** Because the network hardware does not support one-sided communication, the remote `get` operation has to be implemented by sending a request to the processor that owns the data. This processor then has to send a reply to the processor that originally requested it.

  Therefore a remote `get` operation involves the allocation of resources at the passive target. This allocation causes two problems. First, memory allocation on the passive target constitutes overhead. We mitigate this overhead by allocating and maintaining a pool of pre-allocated requests. The second problem occurs when a processor is the target of too many remote `get` requests. Applications written for scalability typically do not exhibit such patterns, and thus we followed the decision made in the BlueGene/L MPI implementation of shifting the burden of managing high volume of communication to the programmer.

3. Compiler Optimizations

In this section we discuss three compiler transformations that improved the performance for the set of benchmarks studied. These optimizations are: reducing the overhead of the parallel loop construct, transforming shared-variable accesses that have affinity to the accessing thread into local accesses, and identifying and exploiting the update primitives of the UPC RTS.

3.1 `upc_forall` Loop Simplification

The `upc_forall` statement is used in UPC programs to distribute iterations of a loop among all threads. Instead of each thread executing all iterations of the loop, an iteration is conditionally executed by a thread based on an *affinity* test. The affinity test is specified by the programmer using a fourth parameter in the `upc_forall` loop declaration. This parameter must contain either a pointer-to-shared type, an integer type or the `continue` keyword. When a pointer-to-shared type is used, an iteration i of the loop is executed by thread j if and only if j owns the shared data specified in the affinity test. Thus, it is common to use the induction variable in the affinity parameter in order to ensure iterations are evenly distributed among the threads. When the affinity parameter is an integer type, an iteration i is executed by a thread j if and only if the integer value of the affinity parameter modulo the number of threads is equal to j. When the `continue` keyword is used, or no statement is specified, the loop body is executed by all threads.

All `upc_forall` loops that use the (unmodified) induction variable of the loop as the affinity parameter are optimized to remove the branch condition from the loop body. The lower bound of the loop is modified to start at the value `MYTHREAD` and the increment of the loop is modified to increment iterations of the loop by the number of threads. This transformation guarantees that each thread only executes the iterations of the loop, as specified by the affinity parameter, without requiring a branch inside the loop body. The removal of the branch statement can benefit many code-reordering optimizations. We are currently improving the way the compiler optimizes `upc_forall` loops to include integer affinity parameters that use a modified induction variable as well as pointer-to-shared affinity parameters. However, even this simple optimization captures many of the loops in the existing UPC benchmarks.

3.2 Local Memory Optimizations

OPTIMIZESHAREDARRAYINDEX($Procedurep$)
1. **for each** loop L_i in p
2. **if** L_i is not a `upc_forall` loop
3. **continue**
4. **endif**
5. **for each** shared memory reference R_s is L_i **do**
6. **if** DIST_MEM_ARCH **and** R_s is non-local
7. **continue**
8. **endif**
9. $R_{handle} \leftarrow$ SVD handle for R_s
10. $L_i^{Preheader}.Add(R_{address} \leftarrow$ `BaseAddress(`R_{handle}`)`$)$
11. $off \leftarrow elt\_sz * ((blk\_sz * course) + phase)$
12. **if** R_s is a *def*
13. $sym_{data} \leftarrow$ data to store to R_s
14. **if** $R_s.DataType$ is intrinsic
15. $L_i^{Body}.Add($`store`$_{ind}($`(`$R_{address}$`, `off`, data)`$)$
16. **else**
17. $L_i^{Body}.Add($`memcpy(`$R_{address} + off$`,data,`elt_sz`)`$)$
18. **endif**
19. **else**
20. $sym_{dst} \leftarrow$ location to store data from R_s
21. **if** $R_s.DataType$ is intrinsic
22. $L_i^{Body}.Add($`dst` $\leftarrow$ `load`$_{ind}($`(`$R_{address}$`, `off`)`$)$
23. **else**
24. $L_i^{Body}.Add($`memcpy(dst,`$R_{address} + off$`,`elt_sz`)`$)$
25. **endif**
26. **endif**
27. $L_i^{Body}.Remove(R_s)$
28. **endfor**
29. **endfor**

Figure 3. Optimizing Shared Array Indexes (Local Memory). DIST_MEM_ARCH is a flag indicating that the target architecture is distributed memory.

This optimization consists of converting accesses to shared data performed through shared handles (see Section 2.2) into pointer dereferences when the location of the reference can be determined. As discussed before, the overhead of addressing shared data through handles (fat pointers) can be significant; it requires several levels of indirection in the SVD and the shared-variable control block. Thus C like pointer dereferences are much less costly. In a distributed-memory architecture, pointers that are known to be non-local must remain fat because it is necessary to use functions defined in the UPC RTS to perform the memory access.

Accesses to shared arrays are optimized using the OPTIMIZE-SHAREDARRAYINDEX algorithm shown in Figure 3. The algorithm examines each shared reference in each `upc_forall` loop in a given procedure. Non-local memory references in distributed-memory architectures are not candidates for this optimization (step 6 of the algorithm). The detection of remote accesses in this algorithm relies on the affinity test of the `upc_forall` loop.

In general, for an affine array-index expression $f(i_1, i_2, \ldots, i_n)$, and a `upc_forall` affinity expression g, the necessary condition to ensure that an array element is local is:

$$(f(i_1, i_2, \ldots, i_n)/blk\_sz)\%THREADS = g.$$

Note that blk_sz (the block size of the shared array or the shared pointer) is known at compile time.

In many cases this condition can be statically verified. For any fat-pointer shared-array reference that satisfies this condition the compiler can transform the array access into a split operation: first, the code to calculate the base address of the array is generated — this code is common to all the elements of an array and can be hoisted out of the innermost loop; and second, the code to calculate the offset and to perform the actual memory operation using traditional C pointers is generated. Array references for which the affinity can not be determined statically will remain fat pointer accesses.

When computing the base address of the array, the same address translation must be performed using the SVD as when the UPC RTS functions are used to access the shared object. Thus, the base address calculation has approximately the same cost as using the UPC RTS functions to access the shared object. The benefit from this optimization comes when the base address of an array is computed once and subsequent shared-array accesses are transformed to direct-memory accesses.

Step 9 obtains the handle used by the UPC RTS to identify R_s. Step 10 inserts a call to a function in the UPC RTS to obtain the base address of R_s in the SVD. The loop preheader contains statements that should only be executed if the loop body executes but do not need to be executed in every iteration of the loop. It is typically used to initialize loop invariant variables used in the loop.

The offset from the base address of R_s is computed using the following equation:

$$elt\_sz * ((blk\_sz * course) + phase)$$

The elt_sz is the size of each shared-array element. The course is used to identify the block that contains the given array element. The phase indicates the element offset within the affinity block. Computing the course and phase requires an integer divide by the number of threads. When the number of threads is known to be a power of two this division can be replaced by a shift operation.

The algorithm then determines the type of reference that R_s represents. If R_s is a definition of (store to) shared data, the symbol representing the data stored to R_s is obtained (Step 13). The sym_{data} symbol is obtained through the expression tree containing R_s (in TPO each reference can locate the expression tree that contains it). If the data type of the reference (*i.e.* the type of the shared array) is an intrinsic, an indirect store is generated to store the data to the memory location $R_{address} + off$ (Step 15). If the data type is not intrinsic, a call to `memcpy` is used to copy the data to $R_{address} + off$ (Step 17). These instructions are inserted immediately after the statement containing R_s in the statement list (in TPO each reference can also identify the statement that contains it).

If R_s is a *use* of (load from) shared data, the symbol representing the location to store the data is obtained. If the type of the shared data represented by R_s is an intrinsic, then an indirect load is used to obtain the data, which is stored to the destination (Step 22). If the type is not an intrinsic, a call to `memcpy` is inserted to copy the shared data from $R_{address} + off$ to the destination (Step 24).

Note that the data types used to test for intrinsics (Steps 14 and 21) must be obtained from R_s. The sym_{data} and sym_{dst} symbols could represent addresses (*i.e.* pointers to shared data) and thus they would not contain information about the underlying type. However, the algorithm can safely assume that the address represented by the pointer will point to memory large enough to contain the shared data because the front end would have generated an error in the event of a type mismatch.

Step 27 removes the statement containing R_s. Because the new statement that replaces R_s was inserted immediately after the statement containing R_s, the original data flow is maintained and no data dependencies are violated.

3.3 Update Optimizations

The RandomAccess benchmark (see Section 4)is part of an important class of applications that use read-modify-update operations. The BlueGene/L messaging library supports an active message paradigm, which enables the following optimization: when the data is not used by the local thread, the update can be performed by the thread that owns the data, remotely. We discuss next how the compiler can detect this situation and what are the benefits.

A read-modify-update operation for a memory reference R_s is defined as $R_s = R_s \texttt{OP} B$, where `OP` is a binary logical operator and B is a variable or constant. TPO recognizes this operation as a scalar reduction on R_s. The challenge for this optimization is to have the analysis recognize shared memory references in this operation and compute the correct affinity, so that optimal code is generated.

The read-modify-update statement is replaced with a call to an appropriate RTS function, specifying the SVD handle for R_s, the logical operation, the data type for the operation, and the value used in the logical operation (B).

If R_s has affinity with the thread P performing the operation, no communication is required, and the update is done in place, by P. Otherwise, an asynchronous message is sent from P to the owner of R_s. This `update` message will trigger an action when received by the owner of R_s. Since the message is asynchronous, the sender will not receive a confirmation.

When the update message is received by the owner of R_s its handler is triggered. The SVD handle is used to locate the underlying memory for R_s. The operation specified in the message is performed using the data value. Finally, the result is assigned to the shared memory location.

The thread performing the remote update does not know when the update has completed. Thus, the remote update optimization is limited to relaxed shared accesses. Remote updates to strict shared accesses are performed using the traditional approach (get the current value of the shared memory location, perform the update and write the new value back to the shared memory location) because execution cannot proceed until the update has finished.

The main benefit of the update optimization is the reduction of inter-node communication from potentially three messages to a single message.

4. Experimental Results

This section presents the environment used to run experiments, the benchmarks used to evaluate the UPC compiler, and the performance results obtained.

4.1 Hardware

The benchmark runs for this paper were done on a number of BlueGene/L installations. Most of the development work was done

on free-standing "node cards" (64 processors) each, and on a single rack of BlueGene/L (2048 processors). The production runs were scheduled on the `BG/W` machine at IBM TJ Watson (20 racks, 40960 processors) and at the LLNL installation (64 racks, 131072 processors).

In all the runs one UPC thread is scheduled for each BlueGene/L processor. Therefore in the following discussion threads and processors are used interchangeably.

4.2 Random Access Benchmark

RandomAccess is one of the four benchmarks that constitute the HPC Challenge Competition [19]. We implemented the UPC version of the benchmark from first principles, following instructions laid out on the HPC Challenge web site. To keep the source code simple, we used the simplest possible algorithm. The resulting UPC code has 111 lines.

Algorithm: the main loop in RandomAccess resolves to a number of read-modify-write (RMW) operations to remote locations across the machine. Each remote RMW operation translates to at least one network packet. We expected performance of this code to be bounded by network latency. Hence good run-time and communication library performance are crucial, as is the compiler's update optimization (Section 3.3.

The RandomAccess benchmark is designed to scale weakly (the memory required by the program is directly proportional to the number of processors). We arranged for 50% of the memory to be used. With perfect scaling, a RandomAccess run should take about 300 seconds regardless of the number of processors it is running on. Since performance does not scale linearly (see the efficiency column in Table 1), the total runtime increases on larger runs.

Verification: the RandomAccess benchmark can be easily verified by running it twice. All updates are *exclusive-or* operations. They restore the original content of the array when executed for the second time. Verification is part of our benchmark implementation.

Performance: Table 1 show the absolute and scaling performance of RandomAccess measured on up to 64 racks of BlueGene/L. The following definition of efficiency for N processors is used to measure scaling performance:

$$\frac{T_{single}}{T_{parallel} \times N}.$$

The benchmark is affected by two performance limiting effects. At low numbers of processors the gating factor is communication latency. For large numbers of processors the gating factor becomes the torus network's cross-section bandwidth. The cross-section bandwidth of a booted BlueGene/L partition is determined by its longest torus dimension; cubic partitions have the highest cross-section bandwidth relative to the number of nodes they contain.

The largest machine configuration we ran RandomAccess on (128K processors), has an effective cross section of:

$$32 \times 32 \times 2 \times 2 = 4096$$

network links. This results from the 32×32 geometry of the cross-section and two doubling factors: each link is bi-directional and the machine is a 3D torus, not a mesh.

Thus cross-section bandwidth for the 128K processor machine configuration can be determined as the product of the wire speed, 175 MBytes/s, and the number of links in the cross-section, yielding $175 \times 4096 = 716,625$ MBytes/s, or approx. 716.6 GBytes/s.

Given that RandomAccess update packets end up as 75 bytes each on the wire, and that only half of all RandomAccess updates have to travel through the cross section, the maximum theoretical GUPs number for the benchmark on this configuration can be calculated as:

$$\frac{2 \times 716.6}{75} = 19.1\ GUPS$$

As Table 1 shows, the actual measured benchmark performance is very close to this theoretical peak.

Threads	Performance	Memory TBytes		efficiency
	(GUPS)	used	total	(%)
1	5.4E-4	0.000128	0.000512	100
2	7.8E-4	0.000256	0.000512	72
4	1.3E-3	0.000512	0.001	61
64	0.02	0.008192	0.016	61
2048	0.56	0.250000	0.500	51
4096	1.11	0.500000	1.000	50
8192	1.70	1.000000	2.000	38
16384	3.36	2.000000	4.000	38
32768	6.10	4.000000	8.000	34
65536	11.54	8.000000	16.000	33
131072	16.72	8.000000	16.000	23

Table 1. Random Access performance results.

Our benchmark beats the absolute performance of RandomAccess measured on any machine other than BlueGene/L, and achieves about 50% of the best known hand-coded optimization written for the same machine.

4.3 EP STREAM Triad Benchmark

EP STREAM Triad is another of the HPC Challenge benchmarks. As with RandomAccess, we implemented this code from first principles, ending up with 105 lines of code.

In the EP (embarrassingly parallel) version of the STREAM triad, all the computation is done locally. We obtained this effect in UPC by using the affinity clause of the `upc_forall` loop.

The memory requirements of STREAM are dictated by 3 shared arrays: the HPC Challenge requirement is that the size of these arrays has to be more than a quarter of the main memory and may not fit in the cache. Thus, STREAM scales weakly. We chose to be conservative and selected the arrays to fill half the memory in each machine where STREAM run.

Verification: on a single processor for an array of more than 366 billion elements verification is expensive and would consume all our machine allocation quota. Therefore we chose to do verification by sampling. Each thread randomly selects a set of indices (the set size being the number of threads running the program) and verifies that the array element at that location has the correct value. Note that as opposed to the embarrassingly parallel triad operation, in which each node operates on local data exclusively, the verification step involves communication across the machine.

Performance: because the benchmark (Table 2) is embarrassingly parallel, there is no scaling drop. Results for 128, 256, and up to 32768 nodes are left out of the table because they contribute no information.

Threads	Performance	Memory	efficiency
	(GB/s)	TBytes	(%)
1	0.73	0.000128	100
2	1.46	0.000256	100
4	2.92	0.000512	100
64	46.72	0.008192	100
65536	47827.00	8.000000	100
131072	95660.77	8.000000	100

Table 2. STREAM Triad performance results.

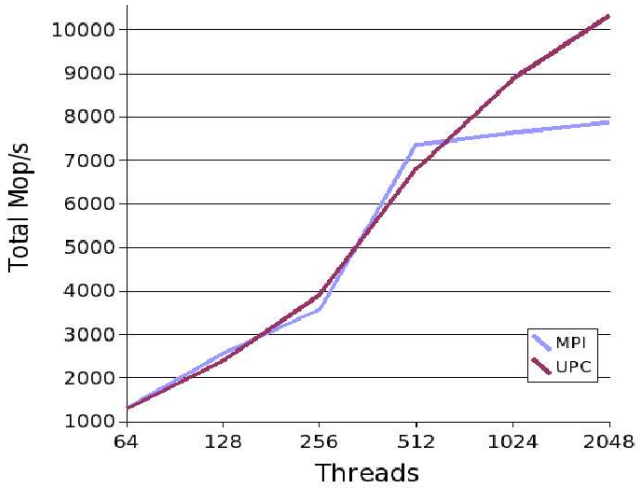

Figure 4. UPC vs MPI scaling on CG class C.

4.4 NAS Conjugate Gradient Benchmark

For this benchmark we used the NAS CG code as implemented by El-Ghazawi and F. Cantonnet [14], with a few changes – we privatized a number of shared variables in the benchmark implementation that need not be shared, for purposes of code clarity and performance.

The resulting code looks similar to the MPI version of the benchmark. A butterfly pattern is set up by the code to aid in the execution of `Allreduce` operations, which are executed by MPI point-to-point primitives. In the UPC version of the code these primitives are replaced by calls to `upc_memget`, `upc_memput` and `upc_barrier`. We ended up using barrier calls because point-to-point synchronization primitives are not yet available in the runtime and in the communication library. NAS CG has built-in verification.

Performance: Figure 4 compares the scaling of the UPC version of the CG benchmark with the NAS NPB MPI version, on input size class C. For up to about 512 processors the performance of both UPC and MPI is equivalent. However, for more than 512, since the problem size remains constant (strong scaling), message sizes become too small to hide MPI overheads for two-sided communication. In the UPC implementation, due to the use of one-sided communication, the overheads are smaller and the benefits appear at 1024 processors and up. The scaling trend in Figure 4 suggests that CG class C will not scale much beyond 2048 processors.

4.5 Effect of compiler optimizations

What is the effect of the compiler optimizations presented in Section 3 in the performance? Table 3 shows the performance obtained by enabling each optimization in isolation. The optimizations presented are as follows: *FE trans* – the translation is done in the FE, *no opt* – TPO translation without any UPC specific optimization, *indexing* – the indexing optimization discussed in Section 3.2, *update* – the update optimization presented in Section 3.3 and *forall* – the forall loop optimization shown in Section 3.1.

An analysis of the results in Table 3 leads to the following observations:

- the baseline code generated by the FE translator already does optimizations, especially inlining and array-access splitting. Therefore, the baseline TPO-generated code is slower on both benchmarks, by as much as 50% on STREAM;
- the *indexing* optimization affects mainly the STREAM benchmark, because all accesses are local, as opposed to Random Access where most accesses are remote;
- the *update* optimization improves Random Access by as much as 200%, because it essentially replaces two messages and three trips across the network (a get and a put) with a single message (the update);
- the *forall* optimization benefits both benchmarks, slightly more STREAM because of the tighter loop;

The most interesting observation is that while each of these optimizations show modest (up to 210% gains), by combining all of them together, we obtain speedups of 7 for Random Access and 240 for STREAM. The compiler was able to transform most of the fat pointers into standard C pointers (local references), enabling the code generation step to optimize the code in the same way that it optimizes a sequential program.

5. Related Work

In addition to UPC, there are a number of partitioned global address space (PGAS) language extensions available. Co-array Fortran [25] and Titanium [32] are the representatives for Fortran and Java, respectively. The family of UPC implementations include Berkeley UPC [9], Cray UPC [12], HP UPC [20], GCC-based Intrepid UPC [17] and MTU UPC [26].

The Berkeley UPC compiler is a source-to-source (UPC-to-C) translator. Its companion runtime system is built on top of GASNet. While a source-to-source translation scheme improves portability, it incurs optimization limitations for accommodating the impact to different back-end compilers. It remains to be seen how the Berkeley UPC compiler and its runtime will scale to hundreds of thousand of threads since current published results are limited to the current generation of machines with a few hundred threads. The SVD design discussed in this paper allows us to scale our runtime to the full size of BlueGene/L.

Chen *et al.* implemented redundancy elimination, split-phase communication and message coalescing in the Berkeley UPC Compiler [10]. When tested with the GUPS benchmark, which performs random read/modify/write accesses to a large distributed array, they observed speedups of 29.3 22.8 and 39.1 on Alpha, Itanium2, and Opteron systems containing 32 processors. They were able to perform split-phase communication by unrolling the read-/modify/write loops in GUPS. Further analysis revealed that message coalescing could not be performed for GUPS because of the presence of indirect memory accesses. Their approach did not distinguish between local and remote accesses and did not attempt to remove unnecessary communication for local shared pointer accesses. However, they did identify this technique as a potential optimization for future work. In the XL UPC compiler, this technique has been implemented and hence the optimization is done automatically.

For communication analysis and optimization, Zhu and Hendren use compiler analysis to select the "best" place for inserting communication, reduce redundant remote access and increase message aggregation [33]. Significant research effort has been also focused on communication optimizations for data parallel programs [8, 18, 28].

Iancu *et al.* optimize communication by demand-driven synchronization [22]. Their runtime system uses virtual memory support to determine the dynamic program point before which the communication should complete. Cantonnet *et al.* propose a technique that resembles the Translation Look Aside Buffers (TLBs)

Benchmark	Measure	FE trans	TPO trans				
			no opt	indexing	update	forall	all opts
Random Access	GUPS	0.00311	0.00270	0.00272	0.00561	0.00438	0.01918
	Time (sec)	172.681	198.492	197.033	95.729	122.673	27.987
	Speedup	1.15	1.00	1.01	2.07	1.62	7.09
Stream	GB/s	0.2028	0.1343	0.1769	0.1343	0.2831	32.3609
	Time (sec)	23.665	35.730	27.129	35.730	16.952	0.148
	Speedup	1.51	1.00	1.32	1.00	2.11	240.77

Table 3. Compiler optimizations effects on Random Access and Stream Benchmarks, running on 64 threads. Speedups are measured relative to the TPO no opt case.

to reduce address-translation overhead [6]. The BlueGene/L UPC runtime runs on top of a polling-based light-weight message layer. Therefore, it does not incur the software overhead caused by interrupt handling.

There is a considerable amount of work evaluating the performance of UPC programs [3, 11, 14, 21]. However, in all these studies, scalability has been studied up to a few hundred processors. This is the first study evaluating the scalability of UPC to hundreds of thousands of processors.

6. Conclusions

The results in this paper show that shared-memory programming for large-scale distributed-memory machines is not a myth. Scaling non-trivial shared-memory programs to hundreds of thousands of threads is possible with the right support from the compiler and from the run-time system. We have described our XL UPC compiler infrastructure and the UPC Run-Time System. We presented the essential compiler optimizations and the runtime features that contributed to high performance. We also illustrated our work with three benchmarks, two of which scaled to more than a hundred-thousand processors on the BlueGene/L machine.

In the course of this evaluation, we encountered several challenging problems, which we will continue to address. One of these challenges is the lack of benchmarks and algorithms written in UPC that can scale to the size of a BlueGene/L computer. Existing efforts, such as the DARPA HPCS program, to provide scalable algorithms and applications for Petaflops computing are the right approach. Using PGAS languages to develop these applications will enable programmers to be more productive, while not sacrificing performance. This paper shows that this is possible.

Trademarks

IBM and BlueGene/L are registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

Acknowledgments

This work was supported in part by DARPA Contract NBCH30390004. We are grateful to a number of people who offered support and advice. In particular, we would like to thank Roch Archambault, Anthony Bolmarcich, Jose Castanos, John Gunnels, Manish Gupta, Roland Koo, Raymond Mak, Philip Luk, Larry Lindsay, Fred Mintzer and Tom Spelce (LLNL) for helping at different stages of this project and with preparing and running our programs on BlueGene/L.

References

[1] G. Almasi, C. Archer, J. G. Castaos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen. Design and implementation of message-passing service for the BlueGene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, 2005.

[2] G. Almasi, L. D. Rose, B. B. Fraguela, J. Moreira, and D. A. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2958 of *Lecture Notes in Computer Science*, pages 162–176, College Station, TX, October 2003. Springer.

[3] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the Cray X1. In *International Conference on Supercomputing (ICS)*, pages 184–195, New York, NY, USA, 2004.

[4] D. Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, U.C. Berkeley, November 2002.

[5] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber. Fast address translation techniques for distributed shared memory compilers. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, 2005.

[7] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, George Washington University, 1999. ftp://ftp.seas.gwu.edu/pub/upc/downloads/upctr.pdf.

[8] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *Programming Language Design and Implementation (PLDI)*, pages 68–78, New York, NY, USA, 1996.

[9] W.-Y. Chen. Building a source-to-source UPC-to-C translator. Master's thesis, University of California at Berkeley, Berkeley, CA, 2005.

[10] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 267–278, Washington, DC, USA, 2005.

[11] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, and Y. Yao. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Symposium on Principles and practice of parallel Programming (PPoPP)*, pages 36–47, New York, NY, USA, 2005.

[12] Cray UPC home page. http://docs.cray.com/books/S-2179-50/html-S-2179-50/z1035483822pvl.html.

[13] DARPA High Productivity Computing Systems. http://www.darpa.mil/ipto/programs/hpcs.

[14] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the Conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[15] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specifications*, v1.1.1 edition, October 2003.

[16] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-burow, T. Takken, and P. Vranas. Overview of the BlueGene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[17] GCC UPC home page. http://www.intrepid.com/upc/.

[18] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of the Conference on Supercomputing*, page 71, New York, NY, USA, 1995.

[19] HPC challenge award competition. http://www.hpcchallenge.org.

[20] HP/Compaq UPC. http://h30097.www3.hp.com/upc/index.htm.

[21] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *International Conference on Supercomputing (ICS)*, pages 63–73, New York, NY, USA, 2003.

[22] C. Iancu, P. Husbands, and P. Hargrove. Hunting the overlap. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 279–290, Washington, DC, USA, 2005.

[23] M. Mendell and R. Archambault. IBM's BlueGene/L compiler implementation. In *BlueGene/L: Applications, Architecture and Software Workshop*, Sparks, NV, Oct 2003. http://www.llnl.gov/asci/platforms/bluegene/papers/10mendell.pdf.

[24] 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*, 10(2):169–189, 1996.

[25] R. Numrich and J. Reid. Co-array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[26] J. Savant and S. Seidel. MuPC: A run time system for unified parallel C. Technical Report CS-TR-02-03, Department of Computer Science, Michigan Technological University, 2002.

[27] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI - a new high-performance communication library for the IBM RS/6000 SP. In *12th. International Parallel Processing Symposium (IPPS)*, pages 260–267, April 1998.

[28] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, I. Eugene W. Hodges, and P. Banerjee. Advanced compilation techniques in the paradigm compiler for distributed-memory multicomputers. In *International Conference on Supercomputing (ICS)*, pages 424–433, New York, NY, USA, 1995.

[29] Top500 supercomputer sites. www.top500.org.

[30] IBM XL UPC compiler. http://www.alphaworks.ibm.com/tech/upccompiler.

[31] K. Yelick. Partitioned Global Address Space Languages: Titanium and UPC experience. Presentation at IBM TJ Watson Research Center, Nov. 2005.

[32] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998.

[33] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. In *Programming Language Design and Implementation (PLDI)*, pages 199–211, New York, NY, USA, 1998.