RC23853 (W0601-126) January 27, 2006
Computer Science

IBM Research Report

Shared Memory Programming for Large Scale Machines

Christopher Barton[1], Calin Cascaval[2], George Almási[2], Yili Zheng[3], Montse Farreras[4], José Nelson Amaral[1]

[1]Department of Compting Science
University of Alberta
Edmonton, Canada

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[3]Department of Electrical and Computer Engineering
Purdue University
West Lafayette, IN

[4]Department of Computer Architecture
Universitat Politecnica de Catalunya
Barcelona, Spain

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Shared Memory Programming for Large Scale Machines

Christopher Barton[†], Călin Caşcaval[‡], George Almási[‡], Yili Zheng[††], Montse Farreras[‡‡]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, NY
{cascaval, gheorghe}@us.ibm.com

[††]Department 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

In this paper we evaluate the use of a shared memory programming language, Unified Parallel C (UPC) on BlueGene/L, a distributed memory machine. We demonstrate not only that shared memory programming for hundreds of thousands of processors is possible, but also that with the right support from the compiler and run-time system, the performance of the resulting codes is comparable to MPI implementations.

We describe the compiler infrastructure, the design of the UPC run-time system and communication software. We also discuss several compiler transformations that were used to optimize the UPC implementation of three well-known benchmarks (HPC RandomAccess, HPC STREAMS and NAS CG). We present scaling and absolute performance numbers for these benchmarks on up to 131072 processors, the full BlueGene/L machine.

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, only allow 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. Several initiatives, such as the DARPA High Productivity Computer Systems (HPCS) program, are encouraging industry and academia to take a fresh look at the issue of programming large scale machines.

This paper explores the use of Unified Parallel C (UPC) [7, 18] on the BlueGene/L® machine. UPC is a shared memory programming extension of C that provides a few simple primitives to allow for parallelism. The programming model is Single Program Multiple Data (SPMD). In the UPC execution model all the threads are started before the user code begins. Threads are synchronized using barriers and locks. The memory model is that of a Partitioned Global Address Space (PGAS) with each thread having access to a private, a shared local, and a shared global section of memory. Threads have exclusive, low latency, access to the private section of memory. Typically the latency to access the shared local section is lower than the latency to access the shared global section.

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

BlueGene/L [19] is a distributed memory machine that features as many as 65,536 dual-processor compute nodes, each operating at a very low power, and hence at a relatively low frequency of 700 MHz. Designed for 360 Teraflops peak performance, the machine sustains 280 Teraflops when running the (optimized) HPL Linpack performance application [33]. 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 strong point of BlueGene/L is its network - a $64 \times 32 \times 32$ 3D torus that spans all compute nodes. The default software installation includes a port of the MPI library, which augments the standard Fortran, C and C++ compiler. It has been shown that careful programming and judicious use of the MPI primitives (and in some cases re-engineering of applications) allow scaling to the full extent 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 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 should increase software productivity and result in the generation of more efficient code.

This paper makes the following main contributions:

- describes the XL UPC compiler and the UPC Run-Time System (RTS);

- shows that, with the right kind of support from the compiler and run-time environment, scaling to hundreds of thousands of threads in a PGAS programming model is possible;
- presents a set of compiler optimizations that are essential for achieving high-performance on large-scale machines;
- discusses and illustrates design decisions in the run-time environment that allowed us to obtain the performance results presented in Section 4.

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 BG/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 and all the way to the code generator. By contrast, source-to-source translators have to rely on the native compiler and the runtime environment for many low level optimizations.

2.1 XL Compiler Framework

The XL Compiler framework 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 does 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 compiling 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-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 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 are treated as kill-sites. This reduces the scope of many data-flow optimizations such as copy propagation and common sub-expression elimination. Second, while the compiler can inline many of these 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 optimizations which could successfully optimize it.

As a result of these performance limitations, the translating of the UPC code to calls to the RTS is delayed until later in the compilation process. To facilitate this, 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. Without these extensions to the intermediate language a compiler has to translate the source-level parallel and data distribution annotations of UPC directly to calls to the UPC RTS.

When the XL UPC compiler uses the extended W-Code, the FE 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 path through the UPC TPO in Figure 1. The advantage is that now all the optimization passes in TPO see the shared array references as memory accesses and can apply all the classical optimizations. In addition we have implemented a set of optimizations specific to UPC that we discuss in Section 3.

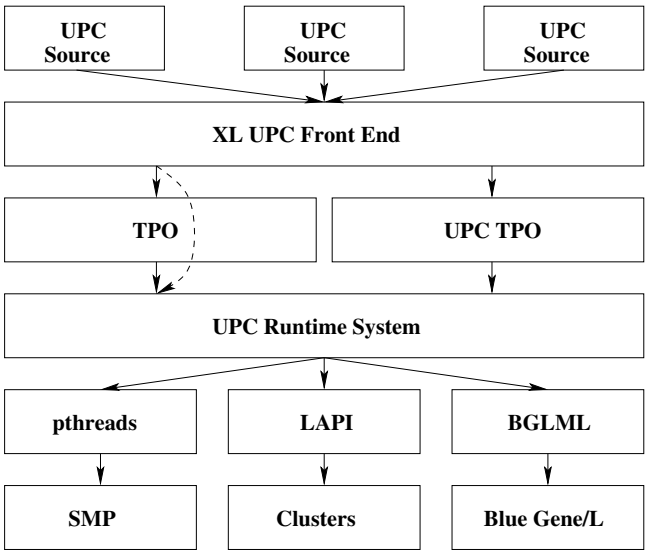

Figure 1. XL UPC Compiler and Runtime System

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 [4]. We have implemented the UPC RTS interface on three different platforms: (1) shared-memory multiprocessors (SMP); (2) clusters of workstations connected either through Ethernet or through specialized networks for which a Low-level Application Programming Interface (LAPI) [31] implementation was available; and (3) BlueGene/L. In this paper we will discuss results only on the BlueGene/L machine. Most of the optimizations presented here are applicable to the other implementations.

The UPC execution environment includes a threading and communication library. On an SMP platform our RTS implementation uses the Pthreads library [5]. The distributed memory version has two implementations, one using LAPI as the communication layer and one using the BlueGene/Lmessage layer [1].

2.2 Runtime System

The UPC RTS exposes a few simple abstractions to the compiler. This makes the translation of UPC programs relatively easy. In addition, to optimize for performance, we can rely on the exiting link time optimizations of TPO, because the entire UPC RTS code is available as a library to the compiler. These optimizations, while used in the reported results, are outside of the scope of this paper.

Shared objects are an important abstraction in UPC programs. The UPC RTS recognizes five kinds of shared objects: shared

scalars, shared structures/unions/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 also have *affinity* to a thread, i.e., they reside in the shared local memory of the thread. Affinity is important because shared data is typically accessed through what we call *fat pointers*. A fat pointer is a structure representation of a shared address that allows the program to reference shared object anywhere in the partitioned global address space. The cost of accessing shared data through these fat pointers is significantly higher than a simple dereference of a traditional C pointer. Therefore, if the compiler can determine statically the affinity of shared objects, it can convert these accesses from fat pointer to C pointer dereferences, 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 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. Rather, like Titanium [34], 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 shared variables independent of each other and keep the SVD consistent with a minimum 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 should keep pointers or references based on the number of processors; rather, if remote information about a variable is required, the requester should get the information through message exchange.

As far as we know, no other PGAS language implementation is able to scale to more than one or two thousand processors, in part because of limitations on their fat-pointer implementations.

The SVD implementation is presented in Figure 2. In this figure we assume a Partition Global Address Space in a distributed memory machine. Each thread owns a section of the memory (the shared local portion) and also has a private section of the memory. Logically, 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 that stores handles to shared variables that have affinity to the thread with `MYTHREAD` equal to the partition number. The partition with number `THREADS`, we call it the `ALL` partition, is used for all statically declared non-scalar variables. The reason for this separation is that the `ALL` partition has a fixed size, while the other partitions are resized when threads allocate shared data dynamically.

Each thread uses a mutually exclusive partition of the SVD. Each partition is an independent, resizable, array of pointers to control structures. If a thread declares a large number of shared variables, only its partition will grow.

Physically, in shared-memory machines the SVD is kept in shared memory. The partitions are stored with affinity to the owning thread. Since the SVD is not exposed to the compiler/user, the "atomic" access rules (see below) apply. For distributed-memory machines, the SVD is kept in the private memory of each thread and it is replicated across all threads. Because we expect most of the operations on the SVD to be global operations in which all threads participate, each thread's copy of the SVD can be updated without communication in a consistent manner (atomically). Communication is required in the case of non-global operations such as `upc_global_alloc` or `upc_local_alloc`. Even in these cases, the communication is non-blocking because our design of the SVD guarantees that only one thread has "write" access rights to its SVD partition.

Shared variable are accessed using handles, as shown by A_h in Figure 2. The handles address a variable by its partition number and the index in the partition. Depending on the type of the shared variable, we obtain the address of the variable or we need through another indirection level. When a thread in a distributed memory system request data from a remote thread it passes only the shared variable handle, and the remote thread will determine the local address. Optimizations, such as caching the values of shared variables and the addresses of shared objects, are outside the scope of this paper.

A major limiting factor of scalability for some implementations of the UPC RTS is the fact that threads are mapped to processes. Moreover, each thread has to map the entire memory space, at the same virtual address, such that static data are implicitly shared by virtue of being located at the same address on all the threads. We overcome this limitation by using the SVD.

Beside the statically declared shared arrays, UPC provides routines for dynamically allocate data, such as `upc_global_alloc`, `upc_all_alloc`, and `upc_local_alloc`. Some of these routines require synchronization/communication between threads. These interactions are non-trivial in distributed memory systems where messages are not guaranteed to arrive in order. Our implementation resolves this issue by partitioning the SVD. Essentially, there is no requirement on the message ordering because operations are "atomic". Each thread is responsible for managing its own partition, and a remote thread will not see the shared variables owned by another thread until its copy of the SVD is updated by the owning thread. All the variables stored in the all partition are allocated through collective operations, therefore guaranteed to be consistent.

2.3 Messaging Library

As mentioned before, the UPC runtime is written to leverage multiple types of hardware, e.g. shared-memory machines as well as the LAPI library; however, all measurements presented in this paper were made on BlueGene/L, using the BlueGene/L communication library.

The BlueGene/L communication library is designed to support both MPI as well as other, more light-weight, communication paradigms, such as UPC, Global Arrays [28] and Hierarchically Tiled Arrays [2].

The design decisions to support both communication paradigms are:

- The relatively low ratio of CPU speed versus network speed makes it imperative to send and process messages in as few CPU cycles as possible.

  Naive (un-optimized) UPC code typically performs individual remote variable dereferences. These result in very short (and hence, latency bound) network communication.

  Implementing the UPC runtime on top of the standard BlueGene/L MPI library would have caused unacceptably high latency because of the software overhead of starting, monitoring,

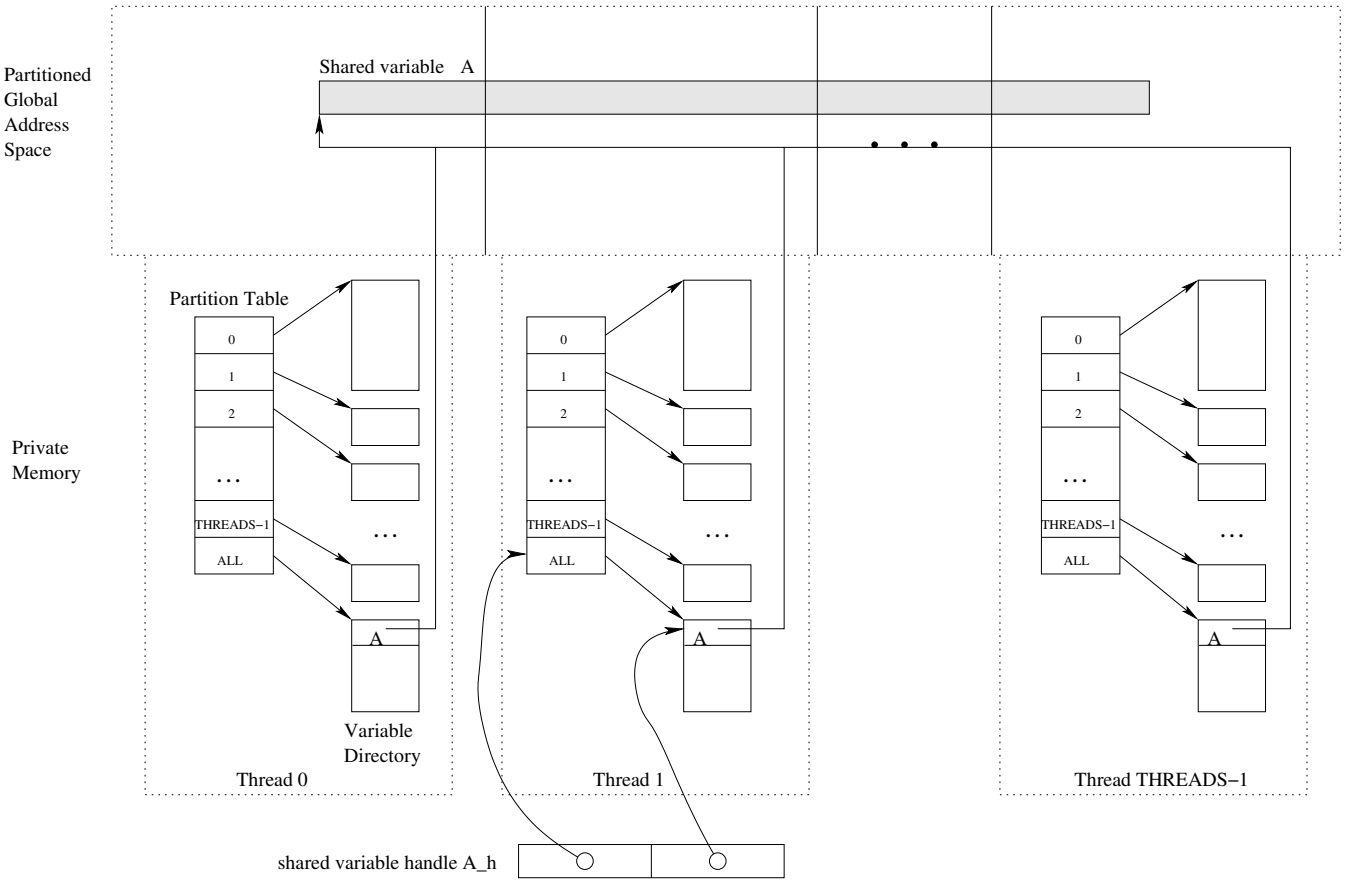

Figure 2. Shared Variable Directory in a PGAS distributed memory machine.

and receiving an MPI message. Many of our communication library optimizations address this problem.

- 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, decreasing overall throughput.

  In practical terms this means that 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 avoiding the need to hand-shake with the receiver for the transfer of a 4-byte value.

- The CPU-network interface is accessible only in 16 byte chunks, with each access being 16-byte aligned. This causes a problem when UPC buffers are arbitrarily aligned, forcing the messaging library to copy data to and from aligned buffers during send/receive. This, in turn, causes CPU overhead (memory copies are inordinately expensive on BlueGene/L), which translates into higher latencies for short transfers as well as decreased bandwidth when a node has to transmit and receive simultaneously.

  The method of dealing with this problem for medium and long messages has been documented in the context of implementing the MPI library [1]. For short data transfers we employed preallocated message objects that have alignment guarantees.

- The BlueGene/L network hardware does not directly support one-sided communication. All network packets have to be extracted from the network by the processor(s). The communication library is designed to be operated by polling, restricting the overlap of communication and computation. Moreover, the same set of double-wide floating point registers are used by the machine to perform computation and to talk to the network device, further restricting overlap.

  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 the fact that they do not have a coherent view of the memory. This makes low-latency communication using a dedicated communication processor virtually impossible.

  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, again 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.

- 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.

  Thus, a remote get operation involves the allocation of resources at the passive target. This causes two problems: first, memory allocation on the passive target constitutes overhead. We mitigate this by allocating and maintaining a pool of preallocated 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 we thus we followed the decision made in the BlueGene/L MPI implementation of shifting the burden on the programmer.

3. Compiler Optimizations

In this section we discuss several compiler transformations that are essential to optimize the performance of UPC codes. 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 communication library.

3.1 `upc_forall` Loop Simplification

The `upc_forall` statement is used in UPC programs to distribute iterations of a loop among all processors. Instead of each processor executing all iterations of the loop, an iteration is conditionally executed by a processor 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 will be 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 distributed among the processors. When the affinity parameter is an integer type, an iteration i will be executed by a thread j if and only if the integer value of the affinity parameter modulo by 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 guarantees that each thread will only execute 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 we optimize `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 most of the loops in the existing UPC benchmarks.

3.2 Local Memory Optimizations

Accesses of shared arrays are optimized using the OPTIMIZESHAREDARRAYINDEX algorithm seen in Figure 3. The optimization consists of converting *fat* pointers into *thin* pointers when the location of the reference allows. A fat pointer is an aggregated data

```
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 ← SVD handle for R_s
10.      L_i^Preheader.Add(R_address ← BaseAddress(R_handle))
11.      index ← COMPUTE INDEX
12.      if R_s is a def
13.        sym_data ← data to store to R_s
14.        if R_s.DataType is intrinsic
15.          L_i^Body.Add(store_ind(R_address, index, data))
16.        else
17.          L_i^Body.Add(memcpy(R_address,data,R_s.Size))
18.        endif
19.      else
20.        sym_dst ← location to store data from R_s
21.        if R_s.DataType is intrinsic
22.          L_i^Body.Add(dst ← load_ind(R_address, index))
23.        else
24.          L_i^Body.Add(memcpy(dst,R_address+index,R_s.Size))
25.        endif
26.      endif
27.      L_i^Body.Remove(R_s)
28.    endfor
29. endfor
```

Figure 3. Optimizing Shared Array Indexes (Local Memory)

structure, used by the UPC RTS, that identifies a shared variable, while its thin counterpart is a standard C pointer. Dereferencing a fat pointer requires several levels of indirection in the SVD and the shared variable control block. Thus thin pointer dereferences are much less costly. In a distributed memory architecture, pointers that are known to be non-local must remain a fat pointer because it is necessary to use functions defined in the UPC RTS to perform the memory access. Thus, when the algorithm is examining each shared reference in each loop in a given procedure, by traversing the loop table, non-local memory references in distributed memory architectures are not candidates for this optimization (step 6 of the algorithm). DIST_MEM_ARCH is an option passed to TPO through Wcode indicating that the target architecture has distributed memory. The detection of remote accesses in this algorithm relies on the affinity clause 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 the 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 statically.

For any array reference that satisfies this condition, which in many cases can be statically determined, we can transform the shared array access using fat pointers into a split operation: first, we calculate the base address of the array, which is common for all the array element accesses to the same array and can be hoisted out of the innermost loop; and second, the offset computation and the actual memory operation using traditional C pointers. Array references for which the affinity can not be determined statically will remain fat pointer accesses.

Step 9 obtains the handle used by the UPC Runtime System (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 UPC Shared Variable Directory (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 is used to identify the affinity block a given array element is located in. The phase indicates the element offset within the affinity block.

The algorithm then determines the type of reference 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 $R_{address} + index$ (Step 15). If the data type is not intrinsic, a call to `memcpy` is used to copy the value in data to $R_{address} + index$ (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, 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 $R_{address} + index$ 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. Even if Step 27 was not performed, the original program semantics would have been maintained because the Write-After-Write data dependence introduced by inserting the new statement uses the same value for the write.

3.3 Update Optimizations

The RandomAccess benchmark performs binary updates on random locations in a shared array. The UPC RTS has support for remote updating operations. TPO identifies updates of memory locations that use a binary logical operator (logical *and*, *or* and *xor*). The updates detected consist of instances of logical binary operators that both define and use the same shared reference, R_s (*i.e.* $R_s = R_s \texttt{OP} B$, where `OP` is a binary logical operator and B can be either shared or private). The statement containing R_s is replaced with a call to the appropriate runtime function passing in the fat pointer used to identify R_s and the data used in the logical operation. The runtime function sends a message to the processor owning R_s specifying the logical operation to perform and the data to use.

Given a processor *P* and a variable *v*, if *P* owns *v* then no communication takes place since the operation is performed locally. Otherwise, an asynchronous message is sent from *P* to the processor that owns *v*.

The message contains the necessary information to locate the variable in the SVD (partition, index, and offset), and the information to perform the operation: the data type, the binary logical operation that needs to be performed, and finally the data used in the operation.

When this message is first received, a handler is triggered, performing the operation locally and atomically (*i.e.* $R_s = R_s \texttt{OP} B$).

4. Experimental Results

In this section we present the environment we ran our experiments in, the benchmarks we used to evaluate the UPC compiler and the actual performance results we 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). All other runs had to be scheduled in advance, either on the `BG/W` machine at IBM TJ Watson (20 racks, 40960 processors), or at the LLNL installation (64 racks, 131072 processors).

In all the runs we scheduled one UPC thread for each BlueGene/L processor. Therefore we will use threads and processors interchangably in the following discussion.

4.2 Random Access Benchmark

RandomAccess is one of the four benchmarks that constitute the HPC Challenge Competition [22]. We implemented the UPC version of the benchmark from first principles, following instructions laid out on the HPC Challenge web site. We used the simplest possible algorithm, to keep source code simple; the UPC code has 111 lines.

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 a network packet; hence, in the current form of the UPC RandomAccess code performance is bounded by communication latency. Good runtime and communication library performance are crucial for this benchmark, as is the compiler's ability to generate remote update calls from a read-modify-write sequence in the source code.

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 2), 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, and restore the original content of the array when executed for the second time. Verification is part of our benchmark implementation.

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 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 of the machine for every machine size that we ran STREAM on.

Benchmark	Measure	FE trans	TPO trans						
			no opt	indexing	update	forall	pwr2	all opts - pwr2	all opts
Random Access	GUPS	0.00311	0.00270	0.00272	0.00561	0.00438	0.00270	0.01815	0.01918
	Time (sec)	172.681	198.492	197.033	95.729	122.673	198.661	29.580	27.987
	Speedup	1.15	1.00	1.01	2.07	1.62	.999	6.71	7.09
Stream	GB/s	0.2028	0.1343	0.1769	0.1343	0.2831	0.1343	0.5978	32.3609
	Time (sec)	23.665	35.730	27.129	35.730	16.952	35.730	8.029	0.148
	Speedup	1.51	1.00	1.32	1.00	2.11	1.00	4.45	240.77

Table 1. Compiler optimizations effects on Random Access and Stream Benchmarks, running on 64 threads. Speedups are measured relative to the TPO no opt case.

Verification: doing the verification on a single processor for an array of more than 366 billion elements 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 embarassingly parallel triad operation, in which each node operates on local data exclusively, the verification step involves communication across the machine.

4.4 NAS Conjugate Gradient Benchmark

For this benchmark we used the NAS CG code as implemented by [17], with a few changes – we eliminated multi-dimensional arrays because the XL UPC compiler does not support them yet; we also 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 what are really `Allreduce` operations, but 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.

4.5 Performance Evaluation

First we discuss the effect of the compiler optimizations presented in Section 3. Table 1 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 optimization, *indexing* – the indexing optimization discussed in Section 3.2, *update* – the update optimization presented in Section 3.3, *forall* the forall loop optimization shown in Section 3.1, *pwr2* specifies to the compiler that the number of threads is a power of 2, *all opts - pwr2* are all optimizations except *pwr2*, and finally *all opts* – all optimizations combined.

There are few observations that we make:

- 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 we are essentially replacing two messages (a get and a put) with one message (the update);
- the *forall* optimization benefits both benchmarks, slighly more STREAM because of the tighter loop;
- the *pwr2* optimization (which essentially enables the compiler to replace an integer division with a shift operation) has no effect on its own – there is far too much overhead in dereferencing shared structures for its effect to show up;

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. As we hinted before, the compiler is able to transform most of the fat pointers into standard C pointers (local references), enabling the code generation step to optimize the code as for a sequential program. This is illustrated by the effect that the *pwr2* optimization has on STREAM after all the other optimizations were performed.

Tables 2 and 3 show the absolute and scaling performance of our benchmarks as measured on up to 64 racks of BlueGene/L. To measure scaling performance we define efficiency for N processors as

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

Going from 1 processor to 2 processors in RandomAccess there is a 28% drop in efficiency, due to the non-locality of the updates on multiple processors. For large numbers of processors the efficiency drops steadily, with larger drops on machine topologies that are farther away from cubic. The network cross-section bandwidth of a booted BlueGene/L partition is gated by its longest torus dimension; hence, cubic partitions have the highest cross-section bandwidth. For example, note the drop in performance between 4096 and 8192 processors.

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
8912	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 2. Random Access performance results.

In terms of absolute performance, the UPC implementation of RandomAccess outperforms the best published MPI implementation results (0.454 GUPS for 2048 processors). Even on the full

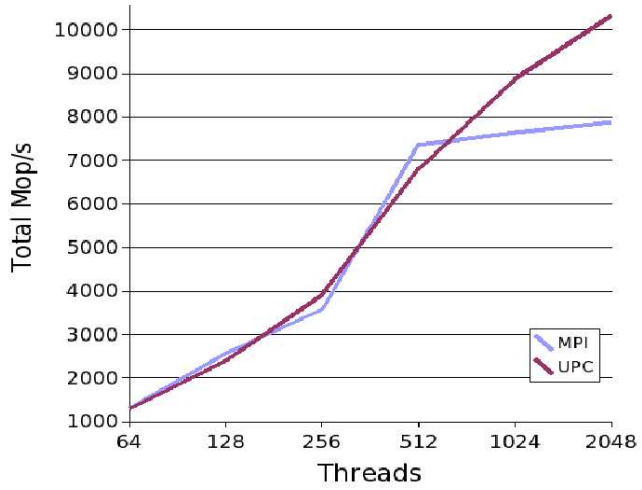

Figure 4. UPC vs MPI scaling on CG class C.

size machine (131072 processors), the simple UPC implementation achieves about 50% of the performance of a hand-coded best known implementation.

STREAM (Table 3) is embarrassingly parallel, and there is no scaling drop. In the table we left out the intermediate results because they contribute no information.

Threads	Performance (GB/s)	Memory TBytes	efficiency (%)
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 3. STREAM Triad performance results.

Figure 4 compares the scaling of the UPC version of the CG benchmark with the NAS NPB MPI version, on input size class C. 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 the Figure suggests that CG will not scale much beyond 2048 processors.

5. Related Work

In addition to UPC, there are a number of partitioned global address space (PGAS) language extensions available. Co-array Fortran [29] and Titanium [35] are the representatives for Fortran and Java, respectively. The big family of UPC implementations include Berkeley UPC [10], Cray UPC [14], HP UPC [23], GCC-based Intrepid UPC [20] and MTU UPC [30].

The Berkeley UPC compiler is a source-to-source (UPC-to-C) translator. Its companion runtime system is built on top of GASNet. While source-to-source translation scheme improves portability, it incurs optimization limitations for accommodating the impact to different back-end compilers. The shared address space in the Berkeley UPC runtime system is limited by the machine address space of a single node [16]. This is a serious limitation when scaling to large scale machines with 32 bit architectures, because the amount of memory on the machine is much larger than what a single node can address. The SVD design discussed in this paper allows us to overcome this limitation by addressing remote memory through indirection.

Chen *et al.* implemented redundancy elimination, split-phase communication and message coalescing in the Berkeley UPC Compiler [12]. When tested with GUPS 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 attempt to remove unnecessary communication for local shared pointer accesses however they did identify this as a potential optimization as 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 [36] use compiler analysis to select the "best" place for inserting communication, reduce redundant remote access and message aggregation. Other significant amount of prior research effort has been focused on communication optimizations for data parallel programs [9, 21, 32].

Iancu *et al.* optimize communication by demand driven synchronization [25]. 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) to reduce address translation overhead[6]. The BlueGene/L UPC runtime runs on top of a polling-based light-weight message layer. Therefore, we save the software overhead caused by interrupt handling.

There is a considerable amount of work evaluating the performance of UPC programs[17, 24, 3, 13]. However, in all these studies, scalability has been studied up to a few hundred processors. As far as we know, this is the first study evaluating the scalability of UPC to hundreds of thousandse of processors.

6. Conclusions

In this paper we have shown 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 have presented the essential compiler optimizations and the runtime features that contributed to high performance. We have illustrated our work with three benchmarks, two of which we 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 was 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. We have shown 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, Sid Chatterjee, 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 et al. 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. Almsi, L. D. Rose, B. B. Fraguela, J. Moreira, and D. A. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *16th International 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 *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, New York, NY, USA, 2004. ACM Press.

[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 *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2005.

[7] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[8] 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.

[9] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 1996. ACM Press.

[10] W. Chen. Building a source-to-source upc-to-c translator. In *Masters Report*, 2005.

[11] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in spmd programs with arrays. In *16th International Workshop on Languages and Compilers for Para llel Computing (LCPC)*, 2003.

[12] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[13] 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 *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM Press.

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

[15] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993.

[16] J. Duell. Memory management in the upc runtime (version 1.1). http://upc.lbl.gov/docs/system/runtime_notes/memory_mgmt.html.

[17] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[19] A. Gara et al. Overview of the Bluegene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195, 2005.

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

[21] 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 *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 71, New York, NY, USA, 1995. ACM Press.

[22] Hpc challenge award competition. `http://www.hpcchallenge.org`.

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

[24] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM Press.

[25] C. Iancu, P. Husbands, and P. Hargrove. Hunting the overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[26] IBM. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann, second edition, 1994.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[28] 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.

[29] R. Numrich and J. Reid. Co-array Fortran for parallel programming, 1998.

[30] 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.

[31] 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 *Proceedings of IPPS '98*.

[32] 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 *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 424–433, New York, NY, USA, 1995. ACM Press.

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

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

[35] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krish-

namurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[36] Y. Zhu and L. J. Hendren. Communication optimizations for parallel c programs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 199–211, New York, NY, USA, 1998. ACM Press.