A Performance Analysis of the Berkeley UPC Compiler

Wei-Yu Chen* Dan Bonachea* Jason Duell†
Parry Husbands† Costin Iancu† Katherine Yelick*†

†Computational Research Division, Lawrence Berkeley National Laboratory
*Computer Science Division, University of California at Berkeley

upc@lbl.gov http://upc.lbl.gov/

Abstract

Unified Parallel C (UPC) is a parallel language that uses a Single Program Multiple Data (SPMD) model of parallelism within a global address space. The global address space is used to simplify programming, especially on applications with irregular data structures that lead to fine-grained sharing between threads. Recent results have shown that the performance of UPC using a commercial compiler is comparable to that of MPI [7]. In this paper we describe a portable open source compiler for UPC. Our goal is to achieve a similar performance while enabling easy porting of the compiler and runtime, and also provide a framework that allows for extensive optimizations. We identify some of the challenges in compiling UPC and use a combination of micro-benchmarks and application kernels to show that our compiler has low overhead for basic operations on shared data and is competitive, and sometimes faster than, the commercial HP compiler. We also investigate several communication optimizations, and show significant benefits by hand-optimizing the generated code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – compilers

General Terms

Performance, Measurement, Languages

Keywords

UPC, performance, global address space, parallel

ICS'03, June 23–26, 2003, San Francisco, California, USA.

1. Introduction

One of the challenges faced by the application developers for high performance parallel systems is the relatively difficult programming environment. For large-scale parallel machines, the most common model is message passing, popularized in the MPI [20] standard interface. Relative to competing programming models such as OpenMP [19], threads, data parallelism or automatic parallelization, message passing has the advantage in scalability, since it runs on distributed memory multiprocessors, and performance tunability, since the programmer has full control over data layout, communication, and load balance. The ubiquity of MPI on parallel machines, enabled in part by the portable MPICH implementation from the Argonne National Laboratory [17], has been an important allure for application developers, who know that machines will change faster than the rate at which applications can be rewritten. The major drawback to message passing, however, is that reprogramming a serial or shared memory parallel application in MPI can be quite difficult, and anecdotal evidence suggests that some users have been "left behind" by the shift from vector shared memory machines to distributed memory and cluster computers.

To overcome these difficulties, some recent efforts have focused on explicitly parallel programming paradigms using a global address space (GAS) model, including UPC [8], Titanium (based on Java) [10], and Co-Array Fortran [18]. These languages are extensions of popular sequential programming languages, and the global address space model that they all share provides a middle ground between a shared and a distributed memory model. In GAS languages, a thread may directly access remote memory through global pointers and distributed arrays just as it would in a shared memory model. GAS languages thus offer a much more convenient programming style than message passing, and good performance can still be achieved because programmers retain explicit control over data layout, parallelism, and load balancing. This notion is well supported in a recent study by the UPC group at George Washington University; using the HP UPC compiler on HP AlphaServer, they demonstrated that the performance of UPC programs is comparable to that of MPI[7]. In addition to the HP compiler, several efforts are underway to implement UPC compilers for a variety of platforms, with compilers currently available for SGI shared memory systems, the Cray T3E, and the Cray X1. One common shortcoming of these compilers, however, is that they are specialized to the particular parallel machine for which they were built. Our goal is thus to build a portable compiler framework for

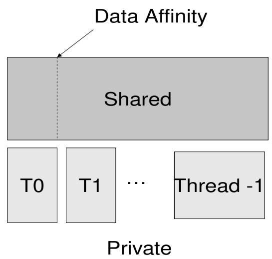

Figure 1. The UPC memory model.

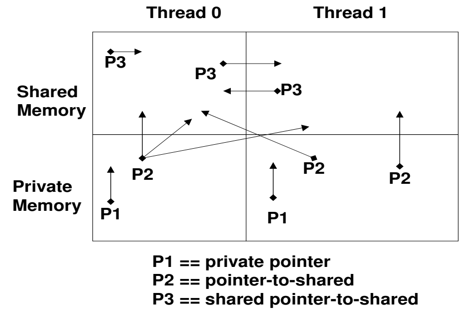

Figure 2. Types of pointers in UPC

UPC that also offers comparable performance to the commercial UPC compilers; UPC will then be able to match MPI in performance and portability, while maintaining a clear edge in ease of programming.

In this paper we describe the Berkeley UPC compiler, which is designed for portability and high performance. To support both of these goals, we use a layered design, so that the implementation can adapt to the functionality of different communication layers, global pointer representations, and processor architectures. Specifically, our compiler generates C code (which is then compiled by a native compiler) that contains calls to our UPC runtime layer, which is implemented atop a language-independent communication layer called GASNet [4]. We use a combination of micro-benchmarks and application kernels to demonstrate several key features of our compiler:

- The modular design of our runtime implementation makes it easy to change representations for pointers to shared data. We will describe and compare two of them.
- We use a novel optimization for pointers to shared data that eliminates most of the cost of blocked cyclic arrays for common special cases.
- For some programs, the performance of serial code from our compiler is comparable to that of a native compiler that generates assembly language directly.
- Despite its layered design, our compiler generates little overhead on shared address calculations, with performance rivaling that of a commercial UPC compiler.
- Our GASNet communication layer provides low-overhead access to the network, giving the UPC application programmer close to maximal performance achievable on the underlying network.

A second goal of our paper is to study the effects of communication optimizations on UPC performance. Because a thread can write and read shared memory directly, UPC encourages a programming style that may result in many small messages. A major challenge for a UPC compiler is thus to bridge the gap between fine- and coarse-grained styles by providing automatic communication optimizations. We explore several communication optimizations such as communication pipelining, aggregation, and overlap of communication with computation. Although these optimizations are not yet automatically performed in our compiler, we are able to measure their potential benefit by transforming the intermediate code by hand, and the results indicate that they are generally effective in reducing the communication cost.

The rest of this paper is organized as follows: In Section 2, we present a brief overview of the language and the design of the Berkeley UPC Compiler. In Section 3, we describe the experimental environment as well as the benchmarks used in the performance analysis. Section 4 discusses the potential compiler optimizations that could improve the performance of the benchmarks, while Section 5 presents the results. Section 6 concludes the paper.

2. Background

UPC (Unified Parallel C) is a parallel extension of the C programming language aimed at supporting high performance scientific applications. The language adopts the SPMD programming model, so that every thread runs the same program but keeps its own private local data; each thread has a unique integer identity expressed as the `MYTHREAD` variables, and the `THREADS` variable represents the total number of threads, which can either be a compile-time constant or specified at run-time. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads, and programmers can declare a shared object by specifying the `shared` type qualifier. While a private object may only be accessed by its owner thread, all threads can read or write to data in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local shared space are said to have "affinity" with the thread, and compilers can utilize this affinity information to exploit data locality in applications to reduce communication overhead. Figure 1 illustrates the UPC memory model.

Pointers in UPC can be classified based on the locations of the pointers and of the objects they point to. Accesses to the private area behave identically to regular C pointer operations, while accesses to shared data are made through a special pointer-to-shared construct. The speed of local shared memory accesses will be lower than that of private accesses due to the extra overhead of determining affinity, and remote accesses in turn are typically significantly slower because of the network overhead. Figure 2 illustrates three different kinds of UPC pointers: private pointers pointing to objects in the thread's own private space (P1 in the figure), private pointers pointing to the shared address space (P2), and pointers living in shared space that also point to shared objects (P3).

UPC gives the user direct control over data placement through local memory allocation and distributed arrays. When declaring a shared array, programmers can specify a block size in addition to the dimension and element type, and the system uses this value to distribute the array elements block by block in a round-robin fashion over all threads. For example, a declaration of `shared [2] int ar[10]` means that the compiler should allocate the first two elements of `ar` on thread 0, the next two on thread 1, and so on. If the block size is omitted the value defaults to one (cyclic layout), while a layout of `[]` or `[0]` indicates indefinite block size, i.e., that the entire array should be allocated on a single thread. A pointer-to-shared thus needs three logical fields to fully represent the address of a shared object: `address, thread_id,` and `phase`. The `thread_id` indicates the thread that the object has affinity to, the `address` field stores the object's "local" address on the thread, while the `phase` field gives the offset of the object within its block. Figure 3 demonstrates how the fields in a pointer-to-shared are used to access a shared value.

To simplify the task of parallelization, UPC also includes a builtin `upc_forall` loop that distributes iterations among the threads. The `upc_forall` loop behaves like a C `for` loop, except that the programmer can specify an affinity expression whose value is examined before every iteration of the loop. The affinity expression can be two different types: if it is an integer, the affinity test checks if its value modulo `THREADS` is the same as the id of the executing thread; otherwise, the expression must be a shared address, and the affinity test checks if the running thread has affinity to this address. The affinity expression can also be omitted, in which case the affinity test is vacuously true and the loop behaves as if it is a C `for` loop. A thread executes an iteration only if the affinity test succeeds, and the `upc_forall` language construct thus provides an easy to use syntax to distribute the computation load to match the data layout pattern. Other notable features of UPC language include dynamic allocation functions, synchronization constructs, and a choice between a strict or relaxed memory consistency model; consult the UPC language specification for more details [8].

2.1 The Berkeley UPC Compiler

The performance analyses and optimizations in this paper use the Berkeley UPC Compiler [3]. Figure 4 shows the overall structure of the compiler, which is divided into three main components: the UPC-to-C translator, the UPC runtime system, and the GASNet communication system.

During the first phase of compilation, the Berkeley UPC compiler preprocesses and translates UPC programs into ANSI-compliant C code in a platform-independent manner, with all of the UPC-related parallel features converted into calls to the runtime library. The translated C code is then compiled using the target system's C compiler and linked to the runtime system, which performs initialization tasks such as thread generation and shared data allocation. The Berkeley UPC runtime delegates communication operations such as remote memory accesses to the GASNet communication layer, which provides a uniform interface for low-level communication primitives on all networks.

We believe this three-layer design has several advantages. First, because of the choice of C as our intermediate representation, our compiler will be available on most commonly used hardware platforms that have an ANSI-compliant C compiler; the currently available UPC Compilers, such as the HP UPC or the SGI GnuUPC compiler, on the other hand, only support specific systems. The gain in portability does not mean that performance has to be sacrificed: the backend C compiler is free to aggressively optimize on the intermediate C output, and the UPC-to-C translator can utilize its UPC-specific knowledge about shared memory access patterns to perform standard communication optimizations. Moreover, the communication overhead is generally low since the GASNet system can directly access the networking hardware instead of going through another communication layer such as MPI, and many runtime and GASNet operations are implemented using macros or inline functions to eliminate function call overhead.

3. Experimental Setup

Portability is one of the major goals of the Berkeley UPC compiler; since the compiler translates UPC programs into C code, it should run on any networking architectures that GASNet communication interface supports (GASNet also has the ability to translate network accesses into MPI calls, so our compiler can run on any system with an MPI implementation; of course, doing so incurs a performance penalty). We perform our experiments on an HP supercomputer located at Pittsburgh Supercomputing Center, which consists of 750 HP AlphaServer ES45 nodes, with each node containing four 1-GHz processors and 4GBs of memory [13]. The nodes are connected by a Quadrics interconnect. We chose this architecture both because the GASNet implementation for the Quadrics network has been performance tuned, and also that we could directly compare our compiler's performance with that of the HP UPC compiler [6], one of the most mature compilers currently available for UPC. We used HP UPC 2.1-003, the latest release of their compiler. For simplicity, in all of our experiments we allocate exactly one thread per node, so that communications between different threads must go through the network. We use the HP C 6.4 compiler to compile the translated C code into object files; unless otherwise noted, all benchmarks are compiled with optimizations enabled ("-O3").

3.1 Micro-Benchmarks

One of the first steps in evaluating the effectiveness of our UPC implementation is to measure the execution costs of the various UPC language constructs in the Berkeley UPC compiler and compare them to the numbers collected from other UPC compilers; the availability of such information could be vital in helping us identify and optimize parts of our compiler that are implemented inefficiently. We thus have written a number of micro-benchmarks to time the overhead of the performance-critical UPC features:

Pointer-to-shared operations: Shared memory accesses are usually the performance bottleneck for communication-bound programs, as UPC encourages a programming style of using pointer dereferences, resulting in a large number of small messages. Pointer arithmetic on shared addresses will also be inevitably slower compared to their counterparts on regular C pointers, since a pointer-to-shared contains three fields, all of which may be updated during pointer manipulations. To combat this overhead, the Berkeley UPC Compiler performs two major optimizations: "phaseless pointers" and a compact pointer-to-shared representation. As mentioned in Section 2, a generic pointer-to-shared contains a `phase`

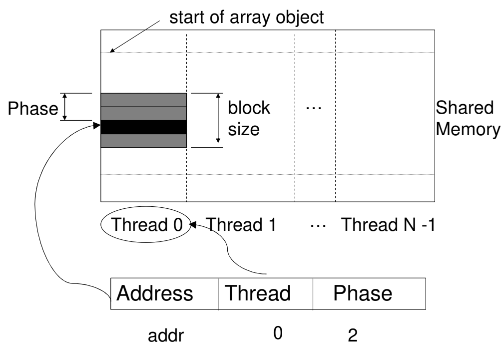

Figure 3. Pointer-to-shared components.

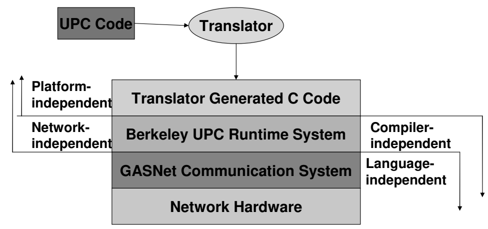

Figure 4. Architecture of the Berkeley UPC Compiler

field as index into the block that the object is located. For the frequently used cyclic pointers, which have block size one, however, the `phase` field can be eliminated since its value is always zero. Similarly, indefinite pointers (block size == 0) can omit their phase since all elements reside in a single block. Cyclic and indefinite pointers are therefore named “phaseless”, and our compiler exploits this knowledge to enable more efficient operations for them.

The other Berkeley UPC optimization is its support for both a C struct and a packed eight byte pointer-to-shared format. The packed representation is preferred for most applications because it enables more efficient implementations for pointer-to-shared operations and fits in the registers of many modern scientific platforms; only in the rare occasions where an application needs either a large amount of shared memory ($>$ 4GB per thread) or more than a few thousand threads does a programmer need to switch to the struct representation.

UPC forall loops: Generally affinity tests for `upc_forall` loops are performed on every loop iteration, and they could introduce a substantial overhead if implemented inefficiently. We have written a simple benchmark to measure the execution time of four loops, each with a different kind of affinity expression: none (effectively a C `for` loop), varying integer value, constant shared address (by taking the address of a shared scalar), and finally dynamic shared address (by taking the address of elements in a shared array). Since the loop body only consists of one variable update (necessary to ensure that the C compiler would not optimize away the loop), the running time of the loops should closely reflect the overhead of the affinity tests.

Dynamic allocation: UPC supports three dynamic shared memory allocation functions: `upc_alloc`, `upc_global_alloc`, and `upc_all_alloc`. The first function acts similar to `malloc()` and allocates a shared object that has affinity with the calling thread, while the other two take block-size and number of bytes per block as arguments, and create a distributed shared array (equivalent to `shared [b] char a[b * nblocks]`, where `b` is the array's block size). The difference between the latter two functions is that the first is meant to be called by a single thread, while `upc_all_alloc` is a collective operation. Although these functions are unlikely to be performance bottlenecks, they do appear with great frequency in UPC applications, and it's interesting to know if our compiler implements them efficiently, especially the most expensive `upc_all_alloc`.

3.2 Synthetic and Application Benchmarks

To further evaluate our Berkeley UPC compiler's performance, we have implemented several synthetic benchmarks and also examined a few UPC application benchmarks from [7]. The following is the list of benchmarks used in our experiments:

vector add This simple benchmark is intended to gauge the performance of local shared memory operations. The program uses a `upc_forall` loop to add two distributed arrays element by element, storing the result to another shared array. The arrays are aligned in ways such that no communication is required, and each thread receives a fixed number of elements. We also experimented with different block sizes for the arrays to see if the compiler could more effectively optimize certain block sizes.

gups pair This benchmark measures the overhead of remote shared memory operations. The program consists of a loop that processes a shared array of structs with two double fields: on every iteration, a number of random array indexes are generated and used to access elements in a shared array. Each random array index corresponds to two read operations and two floating point additions, and the percentage of reads that are made to remote shared memory can be adjusted.

scale This benchmark is similar to gups pair except that it also updates the shared variable, so each iteration now consists of one shared read, one constant multiplication on the value, and one shared write that stores back the value.

ep This is the UPC version of the embarrassingly parallel (EP) kernel from the NAS parallel benchmark suite [1]. As its name suggests, the program performs very little communication and should scale well with large number of processors. We used class A workload as the input size.

is This benchmark is the integer sort (IS) kernel from the NAS suite, and communications are done via the `upc_memget()` bulk memory transfer. Class A workload is used.

npbcg This benchmark is the Conjugate Gradient (CG) benchmark (with Class A input) from the NAS Suite. The UPC code is based on a parallel sparse matrix-vector multiplication in the Aztec[21] library and is not optimized for the

Benchmark	No. lines (UPC)	No. lines (translated C)	Communication Pattern
Vector Add	88	326	None
Gups Pair	92	210	Small Reads
Scale	92	208	Small Read + Write
EP	298	495	None
IS	1082	768	Bulk Memget
CG	757	1071	Mostly Small Reads, Some Writes
NPBCG	564	811	Bulk Memput
MG	1855	2596	Bulk Memput

Table 1. Characteristics of the benchmarks

NAS matrices with their random nonzero pattern. It provides a good test case for UPC codes that perform large `upc_memput` operations.

cg This is another instance of the Conjugate Gradient algorithm, but unlike npbcg, it uses fine-grained communication (single element reads) and matrices from the finite element domain. This represents the performance of a program written in a shared memory style without application-level optimizations.

mg This is the Multigrid (MG) benchmark from the NAS parallel benchmarks. Class B inputs were used in our tests. The communication in the application mainly consists of ghost region exchanges, implemented using `upc_memput` operations in UPC.

Table 1 summarizes the size and communication characteristics for each of the benchmarks used in the paper.

4. Compiler Optimizations

At the time of this writing, the Berkeley UPC compiler does not currently perform UPC language-specific optimizations. Nevertheless, the benchmark data presented in this paper indicates that the performance of the code produced by our compiler is comparable with the performance of the code produced by commercially available implementations of the UPC language.

The results reported in this paper represent an upper bound on the total running time of the benchmarks and we expect performance to improve significantly once we incorporate aggressive UPC-specific optimizations into the translator. In the rest of this section we will present several of these speculative optimizations and in Section 5.2 we analyze their impact on performance by manually applying the transformations to simple benchmarks.

4.1 Communication and Computation Overlap

The GASNet communication layer provides support for blocking and non-blocking one-sided remote memory operations (e.g., put and get) with a wide variety of synchronization options which are appropriate for different code generation situations. One straightforward translation of a remote memory access is to generate a non-blocking initiation call to the communication subsystem, followed immediately by a corresponding synchronization call to ensure its completion. In other words, a blocking remote memory operation `op()` can be translated into the sequence `init_op(); sync_op();`.

In order to hide communication latency, optimizing compilers for parallel languages can leverage the availability of an asynchronous communication interface by performing communication placement optimizations. The basic idea is to move the initiation and synchronization calls for a remote operation as far away from each other in the program as possible, while preserving data and control dependencies. This minimizes the chances that the synchronization call will waste time blocking for completion, and allows other communication and computation to be overlapped with the latency of the remote operation.

Several studies ([5], [22]) present compiler algorithms that perform possible-placement analysis, both on basic blocks and whole programs. Code motion of communication operations in UPC needs to be supplemented with an analysis to ensure that the new schedule of operations does not violate the memory consistency model of the language. Analyses which determine if shared memory operations in SPMD programs can be safely reordered while preserving the consistency model are described in ([11], [12]).

The combination of initiation/synchronization separation for remote operations and code motion optimizations produces a communication pattern known as message pipelining or communication overlap. In Section 5.2.1 we analyze the potential impact of these optimizations on performance.

4.2 Prefetching of Remote Data

An initial study [9] of UPC suggests that bulk prefetching of data is an important optimization for achieving good application performance. The study suggests that programmers and compilers should favor bulk transfers and replication of shared data over fine grained communication. This technique works well under programmer control for dense, array-based numerical codes with good spatial locality; however, compile time transformation of code with fine granularity communication into bulk transfers is likely to be less effective. It is difficult to automate this transformation for pointer-based programs with irregular access patterns, primarily because the compiler lacks the programmer's knowledge of the data usage patterns, and current data dependency and alias analyses are overly conservative or prohibitive in terms of compilation time.

Pointer-based prefetching techniques have been well investigated in the context of serial programs. The Berkeley UPC compiler inherits a working implementation of the pointer prefetching described in [15] from the Open64 code base, however our translator does not currently make use of the generated prefetch hints.

In the near future we plan to specialize this optimization for UPC programs by modifying the prefetch heuristics to handle remote memory fetches.

We analyze the potential benefit of data prefetching in Section 5.2.2.

4.3 Message Coalescing and Aggregation

The global-address space feature of the UPC language often leads to programs with a fine-grained communication pattern dominated by small message sends and receives generated by assignments and pointer dereferences. In the presence of a large number of small messages, the software overheads of communication and network gap can quickly become a performance bottleneck. Such programs could enjoy a significant performance boost if we can automate the packing of multiple messages with the same destination into a larger message to amortize communication overheads.

A recent study [2] shows that in most current high-performance networks, the cost of transmitting small messages is greatly dominated by fixed per-message overheads rather than bandwidth cost, and the aggregation of small messages into larger ones generally pays off for message sizes of up to a few hundred bytes. We evaluate the impact of message aggregation in Section 5.2.3.

4.4 Optimizing Access to Local Shared Data

The Berkeley UPC implementation currently performs all shared memory accesses through GASNet, and we are likely to see a substantial performance improvement if the compiler can statically identify shared data accesses which are local to the accessing thread; the thread can then perform such accesses directly through a private pointer, eliminating the overhead of runtime affinity detection. [14] proposes an efficient type inference system that can distinguish between local pointers and global pointers, with the former corresponding to pointers that address shared data with affinity to the current thread. The design of the UPC language makes it an ideal target for the local qualification inference algorithm, as the blocking factor of a statically declared shared array is guaranteed to be a compile-time constant. Other language features such as indefinitely blocked arrays and affinity expressions in loops can further assist the compiler in recognizing memory locations that have affinity to the accessing thread. Section 5.2.4 studies the effects of such optimizations in UPC.

5. Experimental Results

Figure 5 presents the cost of shared address calculations and underscores the effectiveness of our two major optimizations on pointer-to-shared: phaseless pointers and a packed representation. Note that the results are collected with variables declared `volatile`, so the C compiler would not attempt to optimize away the operation. From the figure it is clear that unoptimized pointer-to-shared manipulations incur a substantial overhead compared to regular C pointer arithmetic, especially when incrementing a struct-based generic pointer-to-shared by an integer offset.

The Berkeley UPC optimizations, however, significantly lower this performance penalty; cyclic pointers easily outperform generic ones thanks to the `phase` field not being adjusted. Indefinite pointers benefit even more from the optimization, since for them only the `address` field needs to be updated. Since cyclic and

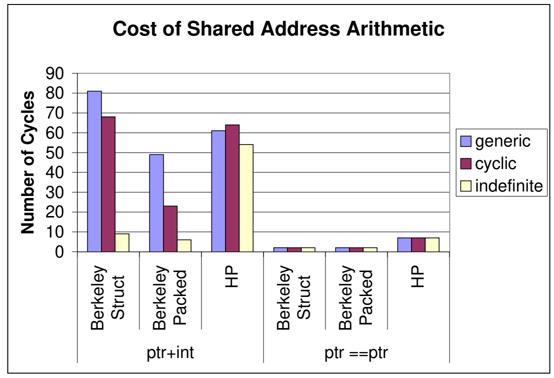

Figure 5. Cost of pointer-to-shared arithmetic (1 cycle = 1ns).

indefinite pointers both appear with great frequency in UPC programs (the former being the default block size and the latter introduced by the `upc_local_alloc` call), we conclude that the phaseless optimization is vital for good performance. Similarly, the packed representation substantially reduces the cost of shared address calculations, especially for generic and cyclic pointers. As a result, we use the packed representation for all benchmarks in the rest of the paper.

Also included in Figure 5 is the cost of pointer-to-shared arithmetic for the HP compiler. HP UPC outperforms the Berkeley UPC's struct-based pointer-to-shared format; this is expected since the HP compiler generates assembly code, while for portability Berkeley UPC implements the operations in C. The packed format, however, is so efficient that it beats HP UPC, again illustrating the effectiveness of our optimizations.

Figure 6 and 7 report the overhead of shared memory stores of a double for the two compilers. For Berkeley UPC accesses to local shared data are slightly slower than private load and stores (two cycles), due to the extra overhead of verifying the object's affinity. Remote accesses in turn are two orders of magnitude slower than local shared accesses due to network latencies. These figures suggest that in general compilers and programmers should attempt to make as much data private as possible by casting pointers-to-shared into private pointers when operating on local shared data; caching values for shared variables can also be a rewarding compiler optimization. When comparing the performance of the two compilers, we observe that HP UPC is faster for local accesses, while Berkeley UPC holds the edge in remote accesses.

Table 2 presents the overhead per iteration of performing affinity tests in a `upc_forall` loop. Not surprisingly, the presence of the affinity expressions introduces extra overhead for the `forall` loop: for an empty loop, an integer affinity expression causes the running time to be three times higher than when it's absent, while an affinity test on a shared address slows down the loop by about 70%. These measurements, however, represent the upper bounds on the real cost of affinity tests; in real programs the `forall` loops likely will contain many instructions, making the relative cost of performing affinity tests less noticeable. Also, if the shared address used as the affinity expression can be calculated at compile time, the compiler can statically determine the result of the affinity tests; similarly if the affinity expression is an integer, the compiler

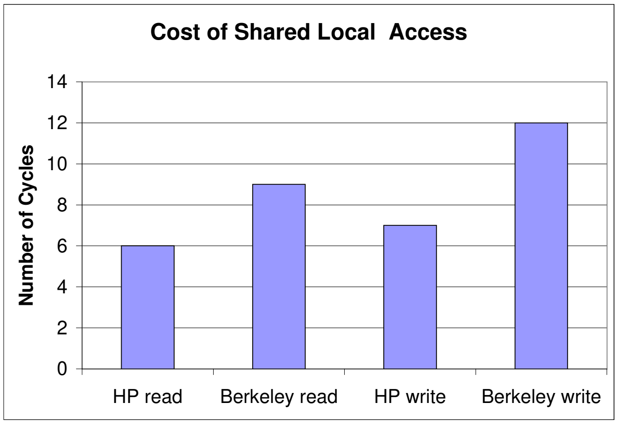

Figure 6. Performance of local shared memory accesses (1 cycle = 1ns).

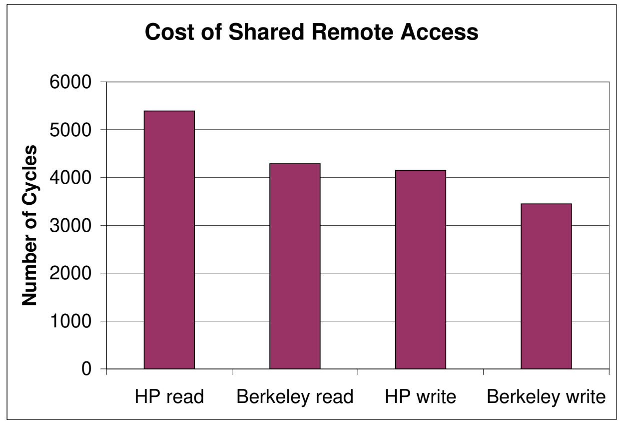

Figure 7. Performance of remote shared memory accesses (1 cycle = 1ns).

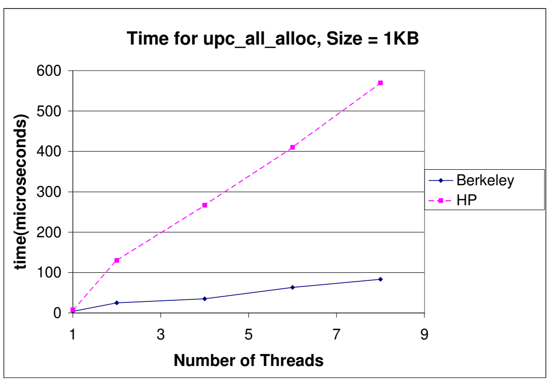

Figure 8. Time for upc_all_alloc

can unroll the loop and eliminate the affinity test in most iterations. We expect the overhead of the affinity tests to be substantially lowered once such optimizations are added to our UPC-to-C translator.

Figure 8 shows the execution time of calling `upc_all_alloc` to allocate a 1KB of data per thread; varying the allocation size has no noticeable effect on performance. Because `upc_all_alloc` is a collective call which broadcasts the result to all threads, its execution time naturally increases as there are more threads. The cost of the operation currently grows linearly with the number of threads; this is suboptimal as the global communication phase could be implemented by a $O(lgN)$ tree algorithm, which we expect to add in the near future. The surprising finding here, however, is the Berkeley UPC compiler outperforms HP UPC by more than an order of magnitude, and the performance gap is consistent despite the increasing number of threads. One possible explanation is that the HP implementation implicitly performs a barrier before all threads may return from the function, while under the Berkeley compiler each thread needs to only receive a broadcast from thread 0, with no barrier involved.

5.1 Performance of the Application Benchmarks

In this subsection, we compare the performance of Berkeley UPC compiler and HP UPC on larger benchmarks. The benchmarks exhibit different communication patterns: none (`vector_add` and `ep`), bulk reads(`is`), fine-grained (`cg`), and bulk writes (`npbcg` and `mg`). The difference between `vect_add` and `ep` is that local shared accesses dominate the former, while only private accesses are present in the latter.

Figure 9 shows the running time of the vector addition benchmark on the two compilers; for Berkeley UPC the running time is shown for both blocked and cyclic pointers (they have the same running time under HP). Since the Berkeley implementation optimizes for cyclic pointers, it runs faster than generic pointers as expected. HP UPC outperforms the Berkeley compiler, but neither scales well due to the overhead of affinity tests that are currently executed in every iteration of the loop. Optimizations for local shared memory accesses will be explored in Section 5.2.4.

Figure 10 reports the running time for the EP benchmark, which uses minimal shared data and requires nearly no remote accesses. The fact that the Berkeley UPC compiler performs as well as HP UPC indicates that the native C compiler can optimize the code output by our UPC-to-C translator just as it would on normal C code. This finding is encouraging because it validates that our approach of lowering UPC code into C code does not interfere with the C compiler's ability to optimize the sequential part of the code. Also, our compiler scales as well as HP UPC for growing number of threads, which is expected considering there are almost no communications or synchronizations.

Figure 11 illustrates the performance of the IS benchmark, for which bulk memory operations consume 95% of the communication time. The two compilers perform about the same for small number of threads, which is not surprising considering that the speed of bulk memory transfers is primarily limited by the network. HP UPC however appears to have difficulties scaling beyond 16 threads.

Figure 12 presents the performance of the conjugate gradient benchmark. We use a positive symmetric definite matrices from the Matrix Market [16], nos7.mtx (729 rows, 2673 nonzero ele-

Loop Type	C For Loop	UPC Forall,integer affinity	Forall, constant addr	Forall, dynamic addr
Berkeley UPC	6	17	10	10

Table 2. Overhead of affinity tests per iteration in UPC Forall loops, in cycles (1 cycle = 1ns)

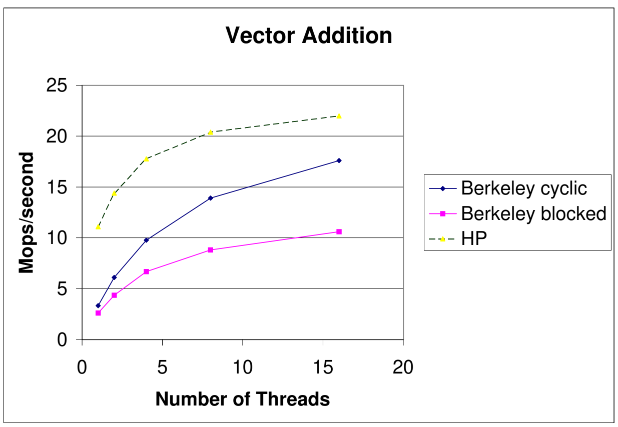

Figure 9. Performance for vector addition.

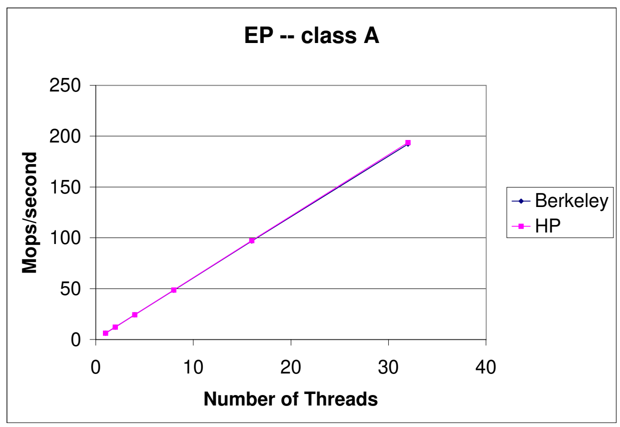

Figure 10. Performance for EP

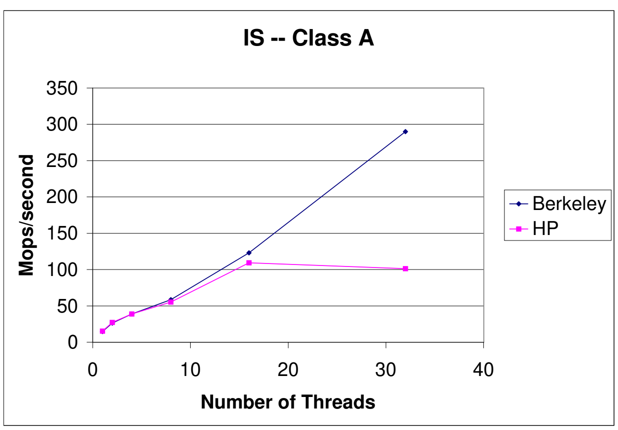

Figure 11. Performance for IS

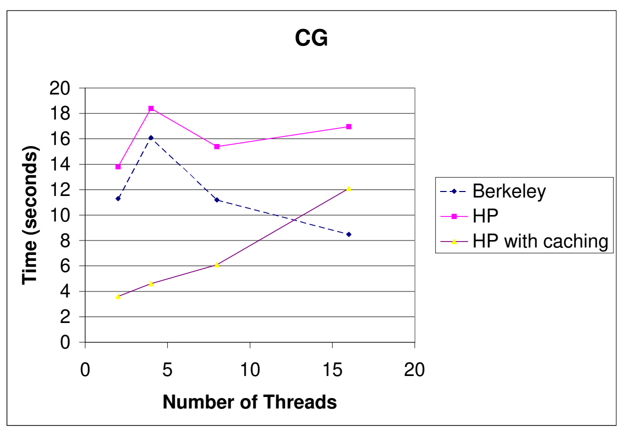

Figure 12. Running time for CG

ments, 1723 iterations to converge without a preconditioner). Here the Berkeley compiler outperforms HP UPC, with the performance gap growing with the increasing number of threads. HP UPC's running time, however, is lowered significantly once its software caching option is enabled, as the overhead of most remote array references is eliminated from cache hits. While redundant remote accesses are unlikely to arise in well-optimized code, caching could still improve the performance of fine-grained programs by prefetching data in the same cache block. Consequently, we are currently implementing a generic software caching scheme that can adapt to different memory consistency models in UPC.

Figures 13 and 14 show the performance of our compiler on code that involves many bulk writes (NPBCG and MG). the HP compiler again does not scale well*, while Berkeley UPC achieves near linear scale-up for MG.

5.2 Communication Optimizations

The previous subsection shows that even without implementing any parallel optimizations, the Berkeley UPC compiler is able to perform as well as other commercially available UPC compilers. The natural question this raises is how much performance gain one can expect once the missing communication optimizations are in place. To answer this question, we manually modify a subset of our benchmarks in ways that we expect a good optimizing compiler would be able to achieve.

5.2.1 Message Pipelining

We use the `scale` benchmark to illustrate the potentials of overlapping communication with more communication. Figure 15 reports the performance improvements due to message pipelining; instead of doing one read/write at a time, we issue five non-blocking requests at once and synchronize all of them together, so that one operation's network latencies can be overlapped with another's. The experiment was done with four threads. The figure shows that while message pipelining in general is helpful in reducing the communication cost of the benchmark, its benefits appear limited (around 5%) and do not seem to be affected by the number of outstanding sends and receives. The Quadrics network, however, is not an ideal target for communication overlapping, because the minimum gap required between every send is larger than the software overhead for each message; computation overlapping instead is more valuable because it will be able to obtain more CPU cycles [2]. this effect is larger.(e.g., Myrinet)

5.2.2 Data Prefetch and Overlapping Communication with Computation

To evaluate the potential of the UPC language and our compiler in overlapping communication with computation, we modified the scale benchmark so that each iteration now also performs a configurable amount of computation. Instead of a constant multiplication, the fetched value is now used to evaluate a polynomial, and the resulted value is stored back. To hide communication latencies by overlapping communication with computation, we manually performed software pipelining on the loop, so that the remote read of the current iteration can be processed simultaneously with the calculations of the previous iteration. Figure 16 shows the benefit of software pipelining on the scale benchmark for four threads (the optimization is also effective for different number of threads). As the figure shows, our optimizations are effective, providing a $15 - 20\%$ gain in performance when there is a substantial amount of remote accesses; the increase will likely be greater if we unroll the loop to provide more operations to be overlapped.

*The poor scaling of the HP compiler on our examples has been reported to their compiler team and may be due to a performance bug

5.2.3 Message Aggregation

Next we use the `gups pair` benchmark to study the effects of message aggregation. The naïve version of the program performs two remote eight-byte reads from consecutive addresses in the memory space of the same thread. We manually combine the two individual read operations into one operation that fetches a double amount of data. Figure 17 shows the resulted running time, and the benefit of message aggregation is fairly apparent. When the ratio of remote accesses is high, the running time of the optimized program is lower by about 50% than the original version, and the performance benefit remains constant as the number of threads grow. This improvement can be attributed to the reduction in communication overhead.

5.2.4 Efficient Access of Local Shared Data

As Figure 9 shows, the naïve vector addition benchmark (which accesses only local shared data) performs poorly due to the overhead of shared address calculation, whose cost gets magnified because the loop does no computation except for an addition. In particular, the speedup achieved in Berkeley UPC is not linear due to the cost of affinity tests, introduced by a parallel forall loop to ensure a thread can only perform addition on the elements it has affinity to. Because the block size of shared arrays is a static type property which is always known at compile time, the optimizer could utilize this information to perform optimizations specifically for local shared data. Figure 18 presents the potential benefit of such optimizations on vector addition with blocked pointers. `Opt1` indicates the elimination of affinity tests by transforming the `upc_forall` loop into an equivalent `for` loop; this is possible because the compiler can statically determine the affinity expressions' values and thus how the iterations are mapped to individual threads. `Opt2` builds upon `opt1` by using private pointers to access local shared data; the pointer-to-shared is cast into a regular C pointer before entering the loop. The figure shows a substantial improvement in performance with these optimizations, especially with `opt2` exhibiting an order of magnitude speedup (note the log scale on the y axis). While the results represents the best case scenario due to the simplicity of the benchmark, they still signify the importance of converting local shared accesses into private accesses where possible.

6. Conclusion

We presented a description of the Berkeley UPC compiler, a portable high-performance compiler for the UPC language. Our results show that, in spite of the modularity used to support portability, the compiler performs well in both absolute and relative terms. In an absolute sense, the communication performance is very close to that of the lowest level networking layer on the machine, with very little overhead from GASNet, the UPC runtime, or the translator. The serial performance is close to that of serial code compiled by a vendor compiler, even though our compiler targets C instead of machine code. In a relative sense, our compiler is competitive with a commercial UPC compiler, with neither

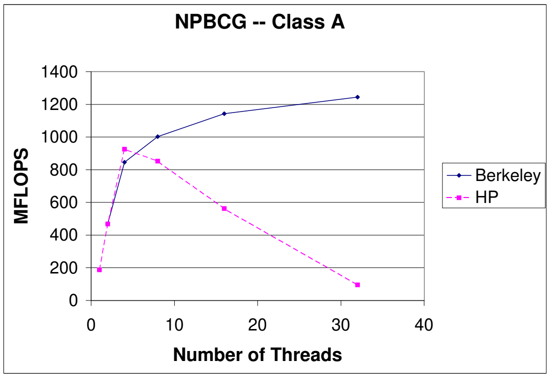

Figure 13. Performance for NPBCG

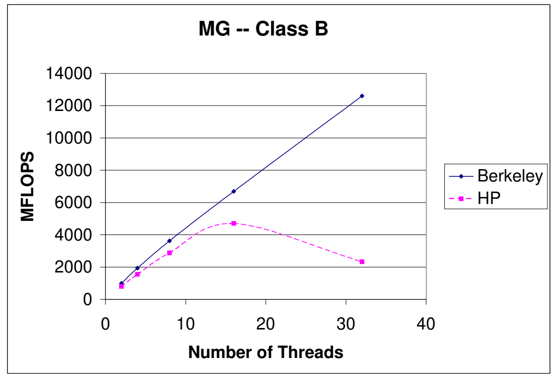

Figure 14. Performance for MG

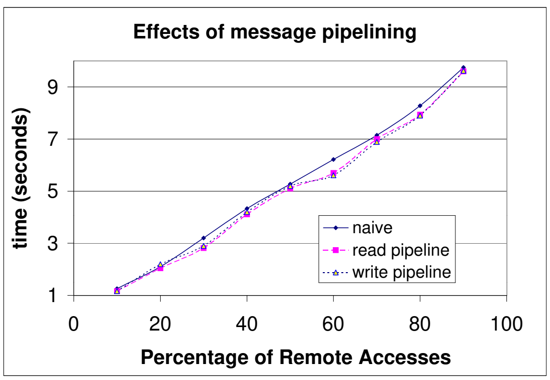

Figure 15. Message Pipelining for scale

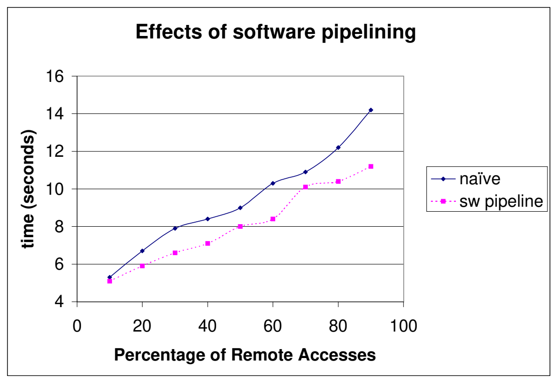

Figure 16. Software pipelining for scale

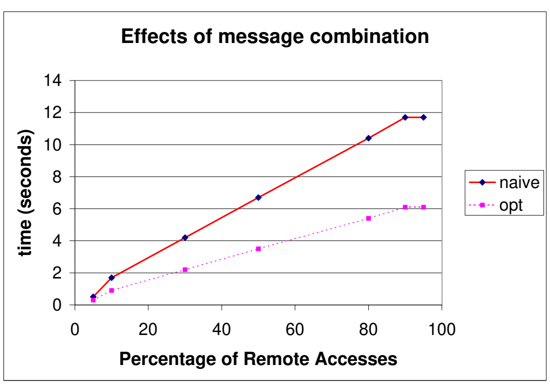

Figure 17. Message Combination for Gups_pair

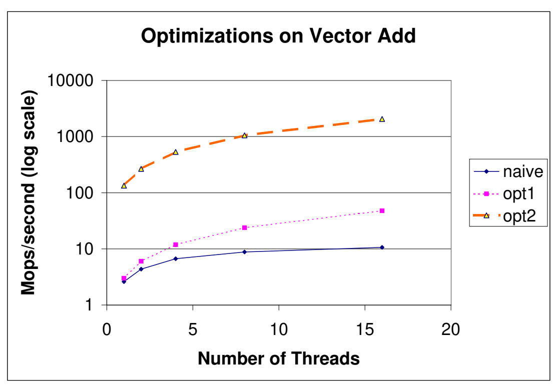

Figure 18. Optimizing Local Shared Accesses for Vector_add

clearly dominating the other across all of the performance measures. As both compilers are still maturing, we expect them to improve significantly over the next few months in code generation quality and in the use of more sophisticated optimizations.

Specific features of the Berkeley UPC compiler that we believe are important in achieving high performance include the use of "phaseless" pointers to lower the cost of general block-cyclic data layouts in important special cases, and a compact pointer representation that reduces the overhead of pointer-to-shared arithmetic and comparison. At the runtime level, GASNet's layered design allows for platform-specific implementations of basic primitives such as put and get when they are supported in the networking hardware. Our compiler performs as well as HP UPC for programs whose communication patterns consist of bulk memory transfers or few memory operations, and significantly better for applications that rely on small message traffic. This implies that applications written in a shared memory style are well-supported by our compiler, with their performance limited primarily by the underlying network. We thus conclude that for UPC portability is possible without making major concessions on performance.

Finally, we evaluated the potential of several compiler optimizations in reducing the communication overhead for UPC programs. Preliminary results from synthetic benchmarks suggest that optimizations such as message aggregation, privatizing local shared accesses, and overlapping computation and communication are promising in their ability to hide the latencies associated with remote shared accesses. Incorporating these optimizations into the compiler will directly improve the programmability of the language by moving some of the performance tuning transformations from the application programmers' hands into the compiler.

7. Acknowledgments

This work was supported in part by the Department of Energy under contracts DE-FC03-01ER25509 and DE-AC03-76SF00098, by the National Science Foundation under ACI-9619020 and EIA-9802069, and by the Department of Defense. The information presented here does not necessarily reflect the position or the policy of the United States Government and no official endorsement should be inferred.

We wish to thank Oak Ridge National Laboratory for the use of their Alpha-based Quadrics system ('falcon'), Pittsburgh Supercomputing Center for the use of the Lemieux supercomputer, and the National Energy Research Scientific Computing Center (NERSC) for the use of their IBM SP ('seaborg'), and Myrinet 2000 cluster ('alvarez'). Thanks also go to Brian Wibecan of HP, who provided very useful comments on the performance of our benchmarks.

8. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[3] The Berkeley UPC Compiler, 2002. http://upc.lbl.gov.

[4] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, Oct. 2002.

[5] S. Chakrabarti, M. Gupta, and J. Choi. Global communication analysis and optimization. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.

[6] Compaq UPC version 2.0 for Tru64 UNIX. http://www.tru64unix.compaq.com/upc.

[7] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing2002 (SC2002)*, Nov. 2002.

[8] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC Language Specifications, version 1.1*, 2003. http://www.gwu.edu/~upc/documentation.html.

[9] T. El-Ghazawi and S. Chauvin. UPC benchmarking issues. In *30th IEEE International Conference on Parallel Processing (ICPP01)*, 2001.

[10] P. Hilfinger et al. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, Nov. 2001.

[11] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[12] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *proceedings of The IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[13] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

[14] B. Liblit and A. Aiken. Type systems for distributed data structures. In *the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2000.

[15] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[16] Matrix market. http://gams.nist.gov/MatrixMarket/.

[17] MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[18] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[19] OpenMP: Simple, Portable, Scalable SMP Programming. http://www.openmp.org.

[20] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[21] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Official aztec user's guide version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.

[22] Y. Zhu and L. Hendren. Communication optimizations for parallel C programs. *Journal of Parallel and Distributed Computing*, 58(2):301–312, 1999.