Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

Title
Communication Optimizations for Fine-Grained UPC Applications

Permalink
https://escholarship.org/uc/item/0j25j80n

Authors
Chen, Wei-Yu
Iancu, Costin
Yelick, Katherine

Publication Date
2005-07-08

eScholarship.org Powered by the California Digital Library
University of California

Communication Optimizations for Fine-grained UPC Applications

Wei-Yu Chen
University of California at Berkeley
Lawrence Berkeley National Laboratory
wychen@cs.berkeley.edu

Costin Iancu
Lawrence Berkeley National Laboratory
cciancu@lbl.gov

Katherine Yelick
University of California at Berkeley
Lawrence Berkeley National Laboratory
yelick@cs.berkeley.edu

Abstract

Global address space languages like UPC exhibit high performance and portability on a broad class of shared and distributed memory parallel architectures. The most scalable applications use bulk memory copies rather than individual reads and writes to the shared space, but finer-grained sharing can be useful for scenarios such as dynamic load balancing, event signaling, and distributed hash tables. In this paper we present three optimization techniques for global address space programs with fine-grained communication: redundancy elimination, use of split-phase communication, and communication coalescing. Parallel UPC programs are analyzed using static single assignment form and a dataflow graph, which are extended to handle the various shared and private pointer types that are available in UPC. The optimizations also take advantage of UPC's relaxed memory consistency model, which reduces the need for cross thread analysis. We demonstrate the effectiveness of the analysis and optimizations using several benchmarks, which were chosen to reflect the kinds of fine-grained, communication-intensive phases that exist in some larger applications. The optimizations show speedups of up to 70% on three parallel systems, which represent three different types of cluster network technologies.

1 Introduction

Partitioned Global Address Space (PGAS) languages have emerged as a promising alternative to the traditional message passing model for parallel applications. Designed as parallel extensions for popular sequential programming languages, PGAS languages such as UPC [12], Titanium [36], and Co-Array Fortran [25] provide better programmability through the support of a user-level global address space, leading to more flexible remote accesses through language-level one-sided communication. This hybrid approach can potentially achieve a good balance between programmability and performance; PGAS languages offer a more convenient and productive programming style compared to explicit message passing (e.g., MPI [22]), and achieve better performance than the pure shared memory model (e.g., OpenMP [27]) because programmers retain explicit control of data placement and load balancing. Another virtue of PGAS languages is their portability: UPC implementations are now available on most platforms, ranging from single processors, shared memory multiprocessors, generic clusters, and supercomputers [1, 29, 13, 9, 7].

Global address space languages offer a convenient programming style, especially for programs with fine-grained data sharing that can be cumbersome in a message passing style. Fine-grained remote accesses, however, are inherently expensive operations on distributed memory machines with high latency networks. Since the compiler assumes the responsibility of communication code generation for PGAS languages, the absence of efficient communication optimizations could result in poor performance for programs that employ fine-grained communication in cluster environments, where a remote memory access is orders of magnitude slower compared to a local memory operation. A naive code generation scheme turns every remote read and write into a blocking round-trip network transfer, which correctly implements a shared memory model but is not effective for fine-grained programs. To improve the performance of fine-grained programs, compilers must reduce the number, volume, and latencies of the message traffic on behalf of the programmer. For PGAS languages, the following optimizations are particularly important for lowering communication overhead:

- `Redundancy elimination`: A well-known sequential optimization, redundancy elimination is also critical for shared memory accesses in PGAS languages, since expensive communication operations may be disguised as such accesses. Redundancy elimination may also be used to avoid some of the bookkeeping overhead of a shared address space.

- `Split-phase communication`: On message-passing networks, communication routines are split-

phase by nature; an *init* call initiates the operation, and a subsequent *sync* call ensures the delivery of data on the remote side. By separating the initiation of a remote memory access as far away as possible from its completion, its latency can be hidden through the overlapping of communication and computation as well as message pipelining. This capability is especially relevant for UPC, which currently offers no nonblocking communication operations at the language level.

- `Message coalescing`: Coalescing small puts and gets into large messages can be a valuable optimization, due to the significant savings of the per-message startup overhead. The data referenced by two remote memory accesses may exhibit either temporal locality (overlap) or spatial locality (belong to the same array or struct), so that a single network transfer would suffice.

In this paper, we describe an optimization framework for fine-grained UPC applications that includes all three optimizations. Using the SSA representation from the Open64 compiler [26], our analysis can support both pointer and array-based shared memory accesses. First, we propose a simple SSA-based partial redundancy elimination (PRE) algorithm to optimize the expensive shared pointer arithmetic operations in UPC. The algorithm is next extended to generate split-phase communications for shared read expressions by propagating their *init* operations upwards in the control flow graph. The optimization achieves both communication latency hiding and elimination of redundant messages, as consecutive remote reads are merged. For remote writes the analysis applies a path-sensitive algorithm to propagate their *sync* operations downward. Finally, a coalescing optimization combines the nonblocking communication calls generated by the split-phase communication analysis to reduce the number of messages and thus save on message startup overhead. The correctness of these optimizations rely on the relaxed UPC memory consistency model, as they change the order of shared memory operations, which may be visible to other threads.

The communication optimizations have been implemented in the Berkeley UPC Compiler [4]. The optimizations were evaluated with a number of irregular UPC benchmarks that employ a variety of communication patterns, including nearest neighbor exchange, random table lookup, and dynamic access to sparse data structures. Experimental results collected on three popular network interconnects suggest that our optimization framework can achieve significant performance improvement for common fine-grained communication patterns, offering speedup as high as 70%.

The rest of the paper is organized as follows. Section 2 presents background information about UPC and the Berkeley UPC compiler. Section 3 describes the compiler algorithms for the three optimizations. Section 4 presents experimental results for a number of fine-grained UPC benchmarks on different network interconnects. Section 5 discusses related work while Section 6 concludes the paper.

2 Background

2.1 Unified Parallel C

UPC is a parallel extension of the ISO 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. 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 data in the shared address space.

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.

UPC gives the user direct control over data placement through 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. 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.

2.2 The Berkeley UPC Compiler

Figure 1 shows the overall structure of the Berkeley UPC Compiler [1], 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 translates UPC programs into C code in a platform-independent manner, with 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

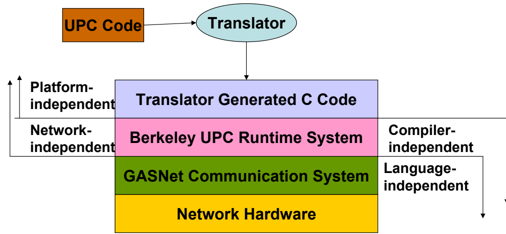


Figure 1. Architecture of the Berkeley UPC Compiler.

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.

The compiler algorithms described in this paper have been implemented in the Berkeley UPC-to-C Translator [1]. The translator is derived from the Open64 Compiler Suite [26], an open-source collection of optimizing compiler tools. Major components in Open64 include front ends for C/C++/Fortran 90, a loop-nest optimizer (LNO), a global scalar optimizer (WOPT), and a code generator for the Itanium architecture. Our optimizations utilize Open64 features such as the Static Single Assignment (SSA) representation and the pointer analysis.

2.3 Runtime Support and Hardware Testbed

The translator converts UPC shared memory accesses into runtime function calls that may require communication if the requested data are remote. For the purposes of this paper, our communication optimizations target the following simple one-sided communication interface:

```
sync_t get(void *dest, shared void *s,
                     size_t n);
sync_t put(shared void *dest, void *s,
                     size_t n);
void   sync(sync_t handle);
```

Both **get** and **put** are nonblocking memory-to-memory operations, transferring n bytes of data and returning an explicit synchronization handle. The **sync** function blocks until the operation corresponding to the supplied handle completes. Synchronization of a get operation implies the local $dest$ (usually a stack temporary) now contains the value of the remote address; synchronization of a put means that the remote $dest$ has been updated with the content of the local source. This generic interface can be implemented on top of any of today's high-performance networks, but also means that compiler has the responsibility of managing handles and issuing synchronization calls at the right place. Specifically, every **get** and **put** must be explicitly synchronized exactly once in the function it appears in.

Performance results were collected on the supercomputer clusters listed in Table 1, covering the three popular high-performance networks. *Get* and *Put* in Table 1 refer to the cost of executing an 8 byte blocking access in our runtime. *Add* refers to the overhead of pointer arithmetic on a generic UPC pointer-to-shared. From the table it is clear that shared address calculation is another source of performance slowdown for UPC code. Since UPC pointers-to-shared are represented as opaque types internally in the translator to achieve maximal portability, shared pointer arithmetic is implemented as a runtime function. Pointer arithmetic on shared addresses is inevitably slower than regular C pointer operations, since a pointer-to-shared contains three fields, all of which may be updated during pointer manipulations. Furthermore, since such expressions are converted into function calls in the translated output, the back-end C compiler likely will have difficulties optimizing them.

3 Compiler Algorithms

UPC extends the ISO-C99 type system with the notion of shared data types, which encapsulate information about data layout. Pointer-to-shared variables in UPC are thus almost as expressive as normal C pointers, and can generally appear anywhere in the code where it is legal for a C pointer. Within the Open64 framework, our translator extends only the type system and uses the same internal program representation for shared expressions, which are distinguished based on their types. This design decision allows us to transparently reuse some of the analyses and the optimizations already present in the framework.

The analyses presented in this paper are performed on the Hashed SSA (HSSA) program representation from [6]. This representation uniformly handles both scalar variables and indirect memory references, and allows a transparent extension of optimization passes developed for scalar variables to handle indirect accesses, e.g.,`*p`, `*(p+1)`, `**p`, `p->x`. HSSA uses a global value numbering approach to build a sparse program representation that captures the aliasing information for both scalar and indirect memory reference expressions. Our work builds on prior HSSA results [37, 2] by extending HSSA to handle both pointer and array expressions.

We present the following analyses: 1) partial redundancy elimination (PRE) for pointer arithmetic on shared types

System	Processor	Network	Software	Get	Put	Add
Alpha/Elan	1GHz Alpha	Quadrics Elan 3	Tru64 V5.1, gcc 3.4.0	7us	6.5us	110ns
Opteron/VAPI	2.2GHz Opteron	Infiniband 4X	Linux 2.6.5, gcc 3.3.3	11.6us	8.4us	72ns
Itanium2/GM	900MHz Itanium2	Myrinet GM 1.6.5	Linux 2.4.18, gcc 3.4.3	26us	16us	220ns

Table 1. Machine summary

and shared memory accesses; 2) split-phase optimizations to separate the initiation of a memory access from its completion; and 3) a coalescing optimization that combines individual messages.

While Open64 includes a powerful PRE optimization [5] (SSAPRE) in its global optimizer, for practical reasons our optimization is implemented as a separate pass. Since the cost of a remote load/store is orders of magnitude slower than a local one, our analysis must go beyond redundancy elimination and apply communication optimizations such as split-phase communication and coalescing. Second, the SSAPRE implementation does not correctly preserve the type information associated with the expressions it eliminates. This type information is needed in a later compiler pass that generates runtime calls. While our optimization is not as powerful as SSAPRE and might miss some optimization opportunities, it handles uniformly both pointer-to-shared arithmetic and load/store operations on shared data. We have found it to work well in practice.

The goal of the split-phase optimizations is to separate the initiation of a communication operation (**get**,**put**) as far apart from its synchronization (**sync**) as possible, while preserving data and control dependencies. This minimizes the chance that the sync call will waste time blocking for completion, and allows other communication and computation to be overlapped with the latency of the remote access. In the case of remote reads, downward code motion of **sync**s is limited by the fact that the value will immediately be needed in the absence of code scheduling, an optimization generally ineffective at the source level. Upward code motion of the **get** operations is not subject to such constraint; we can "prefetch" the remote value by issuing the initiation earlier in the program. A reverse situation applies for remote writes. While the upward movement of the initiation is limited by the availability of the *rhs* value, we can still generate split phase communication by moving the synchronization operation later in the program.

3.1 Optimizing Shared Pointer Arithmetic

The analysis begins with a mark phase that iterates through all statements in a function and finds distinct shared pointer arithmetic expressions. HSSA's global value numbering uses a single node to represent expressions that compute the same value, which makes it trivial to identify the static occurrences of an expression. If the expression is

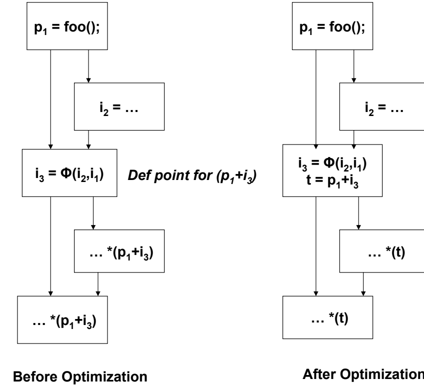


Figure 2. Redundancy Elimination for Shared Pointer Arithmetic.

computed more than once in the program, we consider it to be potentially redundant and determine the earliest point in the function where the expression can be computed[1]. This can be done in two steps. First, we collect the *definition point* for all variables and indirect loads that appear in the expression; a definition point can either be an assignment that explicitly redefines the variable, a statement that may redefine a variable (e.g., function calls), or a ϕ-statement in SSA. Because the program is in SSA form, every variable and indirect load is guaranteed to have a single definition that must dominate it. If a variable is never defined inside the function, we set its definition point to be the function entry point. In the second step, we perform a merge operation on the collection of definition points to find the one that is dominated by all of the rest (i.e., it occurs last). This point serves as the single definition for the shared pointer arithmetic expression, since at this point the values of all variables used by the expression have become available.

The *use-def* information extracted from the SSA form is all that is needed for our optimization. Figure 2 illustrates the transformations of our algorithm. In the example we are working on the shared pointer arithmetic expression $p + i$, which can be computed immediately after the ϕ-assignment to i. We thus place the original expression there and assign its value to a newly created variable. All occurrences of the expression are then replaced with the temporary. While this

[1] In SSA, each use of a variable has the same value.

optimization is not always profitable (e.g., the occurrences of the expression may all be on different paths), the speculation is safe since pointer arithmetic operations will not raise exceptions. Note also that our approach can handle generic multi-term pointer arithmetic expressions such as `p+i+j`.

3.2 Split-phase Communication for Reads

The first step of the analysis is similar to the previous case, as we also compute the single definition point for every shared load. A major difference, however, is that we cannot simply place the dereference operation at the "address" definition point, since it may effectively place communication on a path that does not perform it in the original code. In a cluster environment, communication operations on invalid addresses will generate runtime exceptions and speculative communication movement is unsafe. Furthermore, **get** in our communication system uses RDMA (remote direct memory access) to copy remote data directly into a stack-allocated temporary. All outstanding nonblocking reads must thus be synchronized before a function returns to avoid memory corruption, even if the value is never used. The spurious message traffic can have a significant performance penalty that outweighs the benefits of the optimization.

To prevent speculative code motion, we rely on the concept of *anticipated expressions* [23]. An expression is *anticipatible* at program point p if every path from p to exit evaluates the expression, with nothing in between that could alter the value of the expression. To achieve safe code placement, a shared read e must be anticipatible at a *communication point*, or the program point where a **get** call is inserted. We start by inserting exactly one communication point *cp* in every basic block that contains uses of e. A communication point is either located at the top of its basic block, or right after the definition point if it is contained in the block. It is clear to see that correctness is guaranteed if a **get** is placed at each *cp*, as every *cp* has the property that it is dominated by the definition point and that e is anticipatible at *cp*. Such code generation, however, does not maximize the amount of overlap; for example, in part a) of Figure 3 it is safe to place the communication before the branch, as $*p$ is anticipatible at this point. We solve this problem with a breadth-first postorder traversal of the basic blocks that propagates communication points upward using the following rule: if all of a basic block *bb*'s successors have a communication point, the communication points are merged into one and moved to *bb*. Thus, part a) of figure 3 requires only one communication point in $BB1$, while part b) needs two since not all children of $BB1$ and $BB2$ have communication points.

Once the locations of the **gets** are determined, the corresponding **syncs** are inserted immediately before every use of the expression. Synchronization generation is suppressed

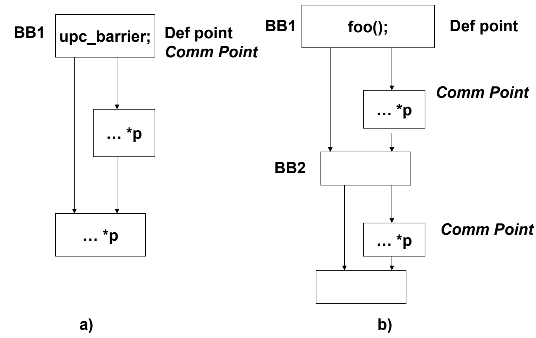


Figure 3. Split-phase Analysis for Reads. Communication points correspond to gets, actual use syncs

if the sync can be statically proven to be redundant (e.g., it follows another sync for the get in the same basic block). To ensure that no nonblocking calls are synchronized more than once, the handle is invalidated after each **sync** call.

3.3 Split-phase Communication for Writes

A different algorithm is needed for remote writes, primarily because the HSSA representation does not provide the *def-use* relation that associates a definition with all of its uses and killing definition. Instead, we employ a path-sensitive analysis that minimizes the number of **syncs** inserted. For each shared write w, our analysis examines paths leading from the statement to function exit, using the rules shown in Algorithm 1. If a statement that may reference or modify the shared location is encountered, we place a **sync** before the statement and terminate the analysis for the current path, to prevent the insertion of redundant syncs in subsequent blocks. For example, in part b) of Figure 4, no **sync** is needed in $BB1$, since the shared write can never reach the block without encountering an earlier sync (marked by the *sync point*). Since for each write a basic block will be examined at most once, the analysis has a $O(n^2)$ running time, where n is the number of expressions. Finally if a path reaches the exit node, a **sync** will be issued at function exit (see Figure 4(a) for example). Once the sync points are determined, a **put** is inserted to replace the shared write.

In general, our algorithm will attempt to push the synchronization as far away as possible from the initiation of the write, with the exception of loops. Our analysis stops the forward code motion when encountering a loop, to avoid executing a **sync** in every iteration. Similarly, the algorithm avoids propagating **sync** along the loop back edge, to pre-

Input: a shared write w of the form $exp = ...$
for *every statement* s *after* w *in the same block* **do**
  if s *uses or modifies* exp **then**
    insert sync for w before s ;
    return ;
  end
end
set : a set of basic blocks;
add to set the successors of w's basic block;
while set *is not empty* **do**
  Remove a basic block bb from set;
  if bb *is seen* **then**
    continue;
  end
  for *every statement* s *in* bb **do**
    if s *uses or modifies* exp **then**
      insert sync for w before s;
      goto while loop;
    end
  end
  Add bb's successors to set;
end

Algorithm 1: Optimizing Shared Writes

clude the error of issuing the operation prior to the corresponding **put**. Loop-invariant code motion is applied instead in the cases where the shared write can be hoisted out of the loop. Redundant writes can be optimized by the standard dead-store elimination algorithm in Open64.

3.4 Coalescing Communication Calls

The split-phase placement analysis optimizes shared accesses individually to hide communication latencies through overlapping. Message pipelining and communication and computation overlapping, however, are not the only ways to reduce communication overhead; by combining the small gets and puts into larger messages, one can save significantly on the per-message startup overhead. Therefore, following the split-phase analysis, we perform another optimization pass to coalesce communication operations.

For remote reads our analysis considers as coalescing candidates **get** calls that share the same communication point; in other words, the gets appear consecutively (pipelined) in the program without other intervening statements. For each communication point, the algorithm coalesces pair-wisely the accesses $get(addr1)$ and $get(addr2)$, where $addr1$ and $addr2$ are the shared source addresses. Since our framework handles both pointer based ($p{\rightarrow}x$) and array based ($a[i]$) accesses, it is not always possible to determine whether two reads can benefit from coalescing. For

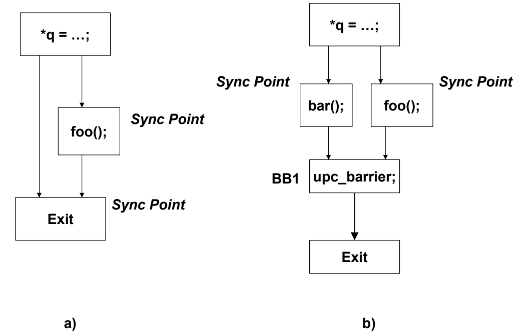


Figure 4. Split-phase Analysis for Writes.

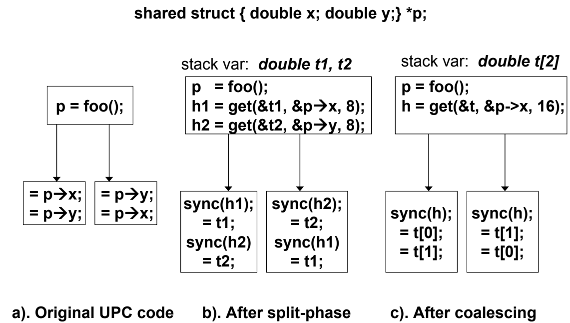
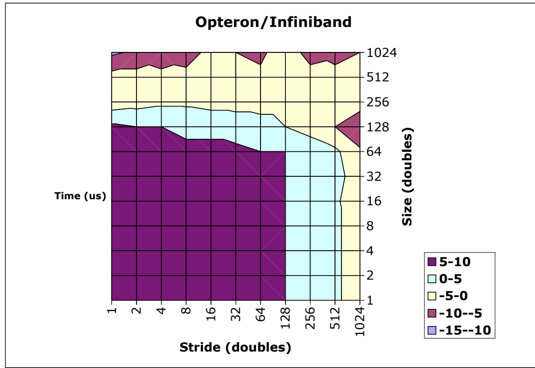


Figure 5. Compiler directed coalescing.

situations where the accesses are to consecutive locations on the same processor (e.g., $p{\rightarrow}x$ and $p{\rightarrow}y$ with x and y next to each other), coalescing is always profitable. Our analysis thus merges the two reads into one nonblocking **get** that fetches both elements and stores the results into a stack temporary. The corresponding **syncs** are also updated to use the new handle. Figure 5 illustrates the transformation.

When $addr1$ and $addr2$ are non-contiguous but still belong to the same processor, profitability for coalescing depends on both network performance characteristics and the distance between the two addresses. Specifically, if the two accesses are combined into one contiguous transfer, the amount of extraneous data between the addresses that must be copied may outweigh the benefit of coalescing. We use microbenchmarks to determine the tradeoffs between coalescing and pipelining on a network; coalescing is favored when the distance is below a certain threshold value, and pipelined communication is used otherwise.

Figures 6 and 7 show the performance difference between pipelining and coalescing communication for the Infiniband and Quadrics networks. The microbenchmark used



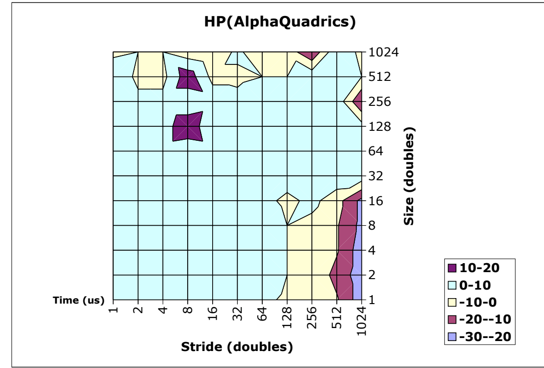


Figure 6. Coalescing versus pipelining performance trends for Infiniband.

Figure 7. Coalescing versus pipelining performance trends for Quadrics.

performs either two pipelined communication calls with a variable size (axis *y*), or a single call to transfer a bounding box containing all the data of interest. The *x* axis shows the distance in bytes between the transferred data areas. For example, the point at (128,1) in the figure shows the performance difference between performing two pipelined 8 byte transfers and a single transfer with size 1024+2*8 bytes. In the two figures, the series labeled with positive values indicate the scenarios where coalescing is faster than pipelining. Based on the results, we use a threshold value of 1024 bytes for the filler space within the bounding box corresponding to a coalesced transfer.

Such a *bounding box* method, of course, is just one way that non-contiguous transfers can be coalesced. An alternative *pack* method that communicates only the needed elements by packing them into a buffer prior to sending may be preferable depending on network and application characteristics. Our communication interface currently supports only contiguous transfers, and we are studying the tradeoffs involved between the two methods.

If the locations of $addr1$ and $addr2$ can not be resolved at compile time, we delay the decision of whether to coalesce them until runtime with a different code generation strategy. A general multi-message call capable of gathering data from arbitrary source memory regions is introduced to our communication interface. The function takes as input an array of the source and destination addresses of the **gets**, as well as their corresponding synchronization handles. The multiple **gets** associated to a *communication point* thus can be collected and issued using the multi-message call. **Syncs** that correspond to the original get operations remain in their original locations. Based on the same coalescing profitability model established from microbenchmarks, the runtime dynamically chooses whether to perform coalescing or not. If coalescing is not desirable, nonblocking communication is initiated for each transfer. If coalescing is profitable, the transfers are coalesced and the runtime is in charge of providing the extra buffer space required and managing the correct synchronization of the operations involved.

The algorithm for coalescing remote writes is similar to that of reads, except that coalescing is only performed on **puts** that access contiguous memory locations (i.e., consecutive struct fields or array elements). The reason is that coalescing individual writes into a single contiguous store may cause spurious updates to memory locations that should not be modified. In [37], Zhu and Hendren present an algorithm capable of coalescing non-contiguous write operations to struct fields, by first fetching the "filler" fields in between that are not written in the original program, then writing the entire struct back. This transformation is unsafe for SPMD programs, since other threads may be simultaneously updating the filler fields in the struct.

3.5 Preserving the UPC Consistency Model

One interesting UPC feature is its support for both a strict and a relaxed memory consistency model [18, 35]. Every shared access in UPC is type qualified as either "strict" or "relaxed", either explicitly or by inferrence from pragmas. The strict memory model ensures *sequential consistency* in that it requires the actual execution of the accesses on each thread to be consistent with program order [19], while relaxed accesses only need to preserve local data dependencies. The difference between the two models is visible only in a program with a *data race*, which occurs when two threads access the same memory location with no ordering

constraints between them, and at least one of the accesses is a write [24]. UPC provides both memory models and allows them to be mixed within a single program. Strict operations are not only sequentially consistent with respect to each other, but they also prevent the movement of certain relaxed operations past strict accesses. Thus, built-in UPC synchronization primitives are strict, and programmers may use strict variables to implement their own synchronization operations such as full-empty bits.

The compiler transformations presented so far maintain safety by preserving local data dependencies. Such a notion of correctness, however, is inadequate for parallel programs, as it does not take into account the restrictions imposed by the synchronization constructs on the ordering of memory accesses. For example, UPC supports barrier synchronization to divide a program into different phases, and accesses from different phases must execute in program order. In our analysis, we model barriers as black box function calls that could modify every shared memory location, so that any code motion across barriers is prevented. For lock-based synchronization, we ensure that code executed inside the critical section will not be moved outside the lock protected region. Therefore, the `upc_lock` operation is modeled as a backward code motion barrier that must complete before subsequent shared memory accesses; similarly, `upc_unlock` will inhibit forward code motion. UPC's memory consistency model also prohibits reordering of strict reads and writes if they could be observed. To prevent illegal reordering caused by our optimizations, we model all strict accesses as we do barriers, i.e., they are repreanted as if they may modify every shared variable in the program. This is conservative, but sufficient for UPC programs in practice, which rarely contain strict accesses other than the built-in synchronization primitives.

3.6 Example

Figure 8 provides a concrete example of how our compiler automatically performs the communication optimizations. The code is extracted from a fine-grained UPC benchmark that performs parallel unbalanced tree search [28], except that variable names were shortened to fit in the figure. The shared arithmetic expression $pool[i]$, which is computed five time in the original program, has been replaced with a temporary variable, eliminating all redundant address computations. The three individual reads following the lock operation are coalesced to reduce their communication overhead, as they access fields in the same struct. The optimization also correctly conforms to the UPC memory model by not issuing any of the pipelined reads before the lock call.

```
struct node_t {
  int workAvail;
  int local;
  int sharedStart;
  ...
};

typedef struct node_t Node;
shared Node pool[THREADS];

int steal(Node*s, int i, int k) {

  int obsAvail = pool[i].workAvail;
  upc_lock(pool[i].stackLock);
  victimLocal = pool[i].local;
  victimShared =
          pool[i].sharedStart;
  victimWorkAvail =
          pool[i].workAvail;
  ...
```

Original UPC Code

```
/* local storage for coalesced gets */
char _CSE4[12];

/* _ADD1 <- pool[i] */
_ADD1 = UPCR_ADD (pool, 480048, i);
_sync7 = UPCR_GET(
          &_CSE5, _ADD1, 0, 4);
UPCR_SYNC (_sync7);
obsAvail = _CSE5;
_sync9 = UPCR_GET (
          &_lock8, _ADD1, 40, 8);
UPCR_SYNC (_sync9);
UPCR_LOCK (_lock8);
_sync10 = UPCR_GET (
          &_CSE4, _ADD1, 0, 12);
UPCR_SYNC(_sync10);
victimLocal = *(int*) (_CSE4 + 4);
victimShared = *(int*) (_CSE4 + 8);
victimWorkAvail = *((int*) _CSE4);
```

Optimized C output

Figure 8. Sample Code from Optimized Programs

4 Experimental Results

We evaluate the effectiveness of the optimizations on communication-intensive UPC code with fine-grained accesses. The benchmarks were written by researchers outside of our group and reflect the kinds of fine-grained communication that is present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling.

4.1 Performance Improvements

Figure 9 presents the speedups achieved by our optimizations over the unoptimized version of the benchmarks. The benchmarks were executed with one thread per node. Each benchmark is discussed in more details below.

Gups: The benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction. The amount of work is static and evenly distributed among threads at execution time. The read/modify/write loops in the benchmark are unrolled to allow for communication overlap[2]. Due to the presence of indirect memory accesses, message coalescing cannot be applied to this benchmark. Our optimizations improve the benchmark running time by an average of 13%, across all networks and processor configurations. The least improvement is observed on the Quadrics network where execution speeds up by at most 3%. The best improvement is observed on the Infiniband network, with an improvement of 27%.

[2] We are currently exploring the heuristics of automatic unrolling in the presence of communication operations.

Mcop: The benchmark solves the matrix chain multiplication problem [8]. This is a classical combinatorial problem that captures the characteristics of a large class of parallel dynamic programming algorithms. The matrix data is distributed along columns, and communication occurs in the form of accesses to elements on the same row. For this benchmark coalescing is not applicable. Our optimizations improve the execution time by an average of 32%, across all networks and processor configurations. Least improvement is observed on the Quadrics network with a value of 26%. Best improvement is observed on the Infiniband network with a value of 60%.

Sobel: The benchmark, whose implementation is described in [11], performs edge detection with Sobel operators (3X3 filters). The image is divided horizontally into rows of bytes and distributed by block rows, so that communication is required only when the thread is processing its first and last row of data. Our optimizations are able to perform read pipelining as well as redundant communication and pointer arithmetic elimination. This benchmark exhibits the highest speedup under the optimizations, as its performance bottleneck is in a short inner loop that is effectively optimized by our compiler. Our techniques improve execution time by an average of 40%, with a maximum of 76% for the Quadrics network.

Psearch: This benchmark performs parallel unbalanced tree search [28]. The benchmark is designed to be used as an evaluation tool for dynamic load balancing strategies. The code fragment in Figure 8 is extracted from this benchmark and shows the effect of our optimizations. This benchmark benefits the least from optimizations, since communication occurs only for the work stealing part of the implementation. The trees are replicated across processors and the benchmark spends only a small fraction of the total running time performing work stealing. Our optimizations improve the execution time by an average of 3% across all platforms and processor configurations. Maximum improvement is observed on the Quadrics network at 7%.

Barnes: The application [34] simulates the interaction of a system of bodies (e.g. galaxies or particles) in three dimensions. It uses the Barnes-Hut hierarchical N-body method. The program performs one tree traversal per body in order to compute the interactions. Written in shared memory style, communication in this program is unstructured and involves many locking operations that can limit the effectiveness of communication optimizations. The most effective optimization for this case is coalescing, and in several instances accesses to fields within a particle were combined into a single call. The optimizations are effective on all platforms, achieving 17% speedup on average and a maximum speedup of 24%. There is a noticeable slowdown for one processor runs on two systems, caused by the extra stack temporaries generated during the split-phase optimization.

Benchmark	Quadrics	Myrinet	Infiniband
gups	29.3	22.8	39.1
mcop	<1 (9.8)	<1 (10)	<1 (10.5)
sobel	46	1.3	0.53
psearch	23	6.5	16.5
barnes	<1 (3.8)	<1 (2.3)	<1 (4.3)

Table 2. Speedups of 32 processor runs relative to 1 processor runs.

4.2 Scalability and Breakdown of Execution Times

Table 2 summarizes the scaling results by computing $t_b(1)/t_o(32)$, where $t_b(1)$ is the running time of serial unoptimized code and $t_o(32)$ that of 32 processor optimized code; a speedup of 32 thus represents linear speedup. Since our benchmarks are communication-intensive, some exhibit slowdowns going from one to two processors, as the introduction of communication outweighs the benefits of parallelism. For such cases, we instead show their parallel scalability (relative to 2 processor unoptimized code, with 16 being linear speedup) in parenthesis.

The results underscore the importance of data distribution in achieving good performance on clusters. For example, *psearch*'s data layout results in a balanced communication/computation ratio and thus scales well, especially on the faster Quadrics network. *Gups* also shows excellent speedup, since the benchmark is constructed such that each processor performs the same amount of work independent of other processors. *Mcop*, however, scales poorly from one to two processors, as its cyclic layout dictates that four reads to different processors may be performed in a critical loop. While our optimizations are able to overlap those reads, it could not overcome the inefficiencies of the data distribution strategy. Similarly, our *barnes* implementation is written in a shared memory style, performing only fine grained memory accesses and does not scale well. Although coalescing reduces the costs of accessing fields in the same node, it provides only incremental improvements, and no compiler optimization short of changing the application's parallel layout is likely to remedy the poor scalability on distributed memory machines. Shan et al [30] show that a bulk implementation of the same algorithm using MPI outperforms a shared memory implementation even for SMP systems.

Figure 10 presents a breakdown of the benefits of each individual optimization. Four threads were used in the execution. In the figure, "Add" represents the results obtained with only redundancy elimination enabled for pointer

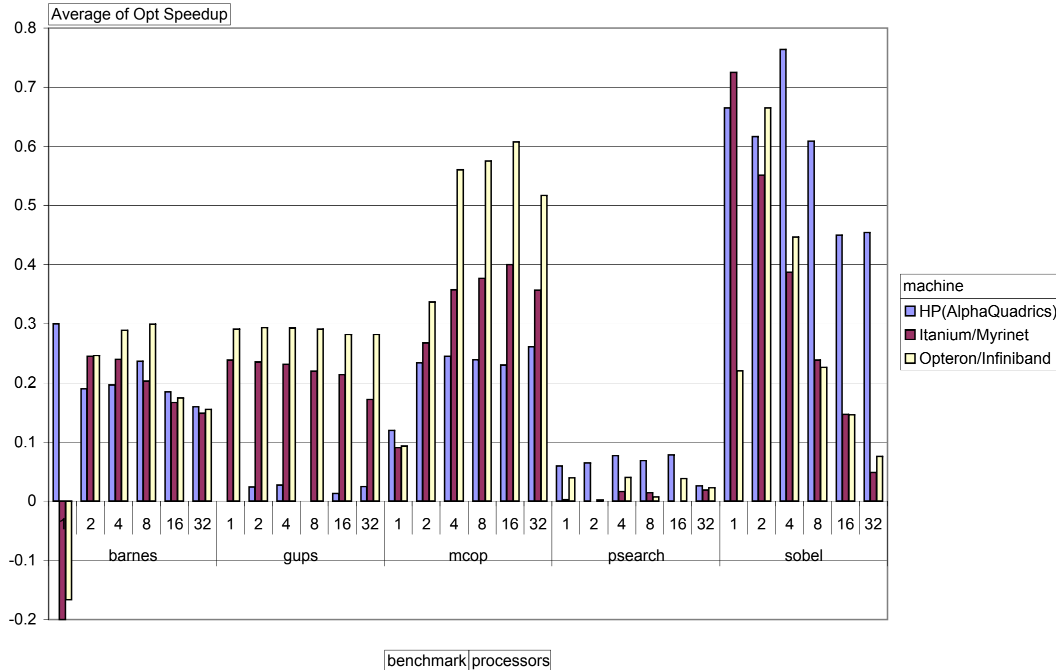


Figure 9. Optimization speedup, measured as percentage over unoptimized version

arithmetic. "SP+Add" optimizes the program further with nonblocking communication, while "Full" is the most optimized version that also performs coalescing. As the figure shows, all three optimizations contribute to the reduction of the program's execution time; while redundancy elimination produces a substantial performance gain on several benchmarks, applications like *barnes* that read and write fields from a node will profit more from coalescing. The split-phase optimization as expected is more effective on networks with higher latencies (e.g. Myrinet).

5 Related Work

In the context of communication optimizations for explicitly parallel languages, the prior research that is most closely related to our techniques is Hendren and Zhu's work on parallel C programs [37]. Their analysis identifies the earliest point that a remote read can be issued, and applies either pipelined or blocking (coalesced) communication based on heuristics. Remote writes are propagated forward in the program and might also be subject to coalescing. Our work is different from theirs in several ways. They consider only pure pointer based programs, while the HSSA representation in our framework allows us to optimize both pointer and array based programs. More importantly, their analysis is assisted by runtime support present in the Earth MANNA system: 1) non-abortive speculative communication and 2) runtime managed synchronization for communication operations. The first allows for speculative loads on possibly invalid addresses, while the second will automatically suspend execution if a thread attempts to use a value for which a remote load has not completed. The presence of both features significantly simplifies the compiler analyses required. Our target systems (commodity networks and processors) lack the hardware support required for an efficient implementation of these features.

Krishnamurthy and Yelick [17] studied compiler analysis and optimizations for explicitly parallel programs with a global address space. Most of their work focuses on improving the accuracy and efficiency of the cycle detection [31] algorithm for SPMD programs, which enforces sequential consistency under reordering transformations. Our work instead uses UPC's relaxed memory model to avoid this analysis, although it could be augmented with cycle de-

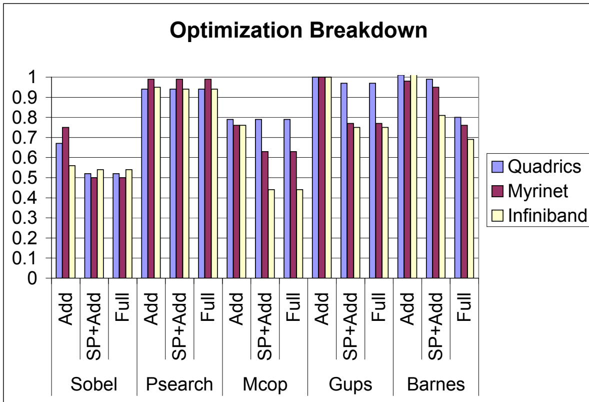


Figure 10. Breakdown of the benefits of the optimizations. Execution times are normalized so that the time of the unoptimized code is 1.

tection so that UPC's strict accesses would become candidates for optimization. Their work was done on a subset of the Split-C [10], but with the important simplifying assumption that pointers are never aliased. Our compiler performs the necessary analysis to handle aliasing. Another difference is that they do not consider the opportunities for redundancy elimination or communication coalescing.

Lee et al. [21, 20] also describe an approach for compiling explicitly parallel programming languages. They present a concurrent static single assignment (CSSA) form that can represent parallel programs with `cobegin/coend` and `post/wait` synchronization. They propose several optimizations based on the CSSA form, including global value numbering, common subexpression elimination, and redundant load/store elimination. Their optimizations maintain correctness by enforcing sequential consistency, while our method takes advantage of UPC's relaxed memory model to preserve sequential consistency only for strict accesses. Their work addresses a more general parallelism model than ours but a more restricted class of shared memory architectures where explicit non-blocking communication optimizations are not relevant.

Communication optimizations of parallel programs have been studied extensively in the context of data parallel languages [32, 14, 15, 33, 16, 3]. Although some of the optimizations are similar to our work, the analysis problems are quite different: data parallel languages have a serial semantics without race conditions or memory consistency models, but they have an added burden of managing the mapping of fine-grained parallelism onto coarse-grained architectures. These projects tend to focus on array optimizations, such as communication vectorization, an ongoing effort in our compiler as well. In addition, the work by Chakrabarti et al. [2] uses redundancy elimination and a global communication optimization algorithm for reads, which identifies the earliest and latest safe position to issue the communication. Their analysis is different as it starts from a data parallel language and adds an additional greedy algorithm to attempt to optimally place communication between the earliest and latest possible points.

6 Conclusion and Future Work

In this paper, we presented an optimization framework that effectively improves the performance of fine-grained UPC programs. The framework consists of three optimizations: a PRE optimization that eliminates redundant shared pointer arithmetic and memory accesses, a split-phase optimization that increases communication and computation overlap, and a coalescing optimization that reduces the message startup overhead. Experimental results on a number of benchmarks suggest that our optimization framework can achieve impressive speedups for common fine-grained communication patterns on today's high performance networks. While our optimizations do not overcome the problem that fine-grained accesses perform poorly on clusters, they do reduce a substantial portion of their communication overhead. Combined with programmer efforts such as more sophisticated choices of algorithms and data decomposition that reduce the amount of communication necessary, our framework has the potential of achieving reasonable performance on clusters for UPC applications written using fine-grained accesses.

There are several areas where our analysis can be further improved. One useful extension is to incorporate a locality analysis that attempts to determine statically whether a shared access is local or remote, so that the communication optimizations can concentrate only on remote accesses. The analysis presented in Section 3 does not perform speculative communication movement, because of both correctness constraint (loading incorrect address causes exceptions) and performance considerations (speculation may worsen performance by creating additional communication). Although it is unclear whether speculative code motion will help fine-grained UPC programs, the former problem may be solved by extending our runtime system to support speculative remote accesses that suppress the exception on invalid address until the value is actually used. The latter problem may be addressed by utilizing profiling information to issue speculative accesses only when they appear profitable.

Runtime based coalescing is another area where we can

improve the performance of our approach. In particular, when coalescing non-contiguous accesses, we need to understand better the performance tradeoffs between the different methods for fetching remote data. For example, a gather operation that packs the selected elements on the remote side may be appropriate for small number of accesses, while for large number of accesses retrieving a contiguous bounding box with the needed elements may be more efficient.

References

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

[2] 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.

[3] D. Chavarria-Miranda and J. Mellor-Crummey. Effective communication coalescing for data parallel applications. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, June 2005.

[4] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[5] F. Chow, S. Chan, R. Kennedy, et al. A new algorithm for partial redundancy elimination based on SSA form. In *Proc. of SIGPLAN '97 Conf. on Programming Language Design and Implementation (PLDI)*, May 1997.

[6] F. Chow, S. Chan, S. Liu, et al. Effective representation of aliases and indirect memory operations in ssa form. In *Proc. of 6th Int'l Conf. on Compiler Construction (CC)*, April 1996.

[7] Compaq UPC version 2.0 for Tru64 UNIX. http://h30097.www3.hp.com/upc/.

[8] T. Cormen, C. Leiserson, and R. Rivset. *Introduction to Algorithms*. The MIT Press, 1994.

[9] Cray C/C++ reference manual. http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003/.

[10] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing (SC1993)*, 1993.

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

[12] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC specification*, 2003. http://upc.gwu.edu/documentation.html.

[13] GCC Unified Parallel C. http://www.intrepid.com/upc/.

[14] M. Gupta, S. Midkiff, E. Schonberg, et al. A HPF compiler for the IBM SP2. In *Supercomputing 1995*, November 1995.

[15] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1996.

[16] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, 1999.

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

[18] W. Kuchera and C. Wallace. The UPC memory model: Problems and prospects. In *the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, January 1976.

[20] J. Lee, S. Midkiff, and D. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *10th International Workshop on Languages and Compilere and Parallel Computing (LCPC)*, August 1997.

[21] J. Lee, P. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 1999.

[22] The Message Passing Interface (MPI) standard. http://www.mpi-forum.org/.

[23] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[24] R. Netzer and B. Miller. What are race conditions? some issues and formalization. *ACM Letters on Programmming Languages and Systems*, I(1), March 1992.

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

[26] Open64 compiler tools. http://open64.sourceforge.net.

[27] OpenMP specifications. http://www.openmp.org.

[28] J. Prins, J. Huan, W. Pugh, et al. UPC implementation of an unbalanced tree search benchmark. Technical Report 03-034, Department of Computer Science, University of North Carolina, 2003.

[29] 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 Techincal University, September 2002.

[30] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. Message passing vs. shared address space on a cluster of smps. In *Proceedings of The International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[31] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, April 1988.

[32] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. H. IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *9th ACM International Conference on Supercomputing*, pages 424–433, July 1995.

[33] C.-W. Tseng. *An optimizing Fortran D compiler for MIMD distributed-memory machines*. PhD thesis, 1993.

[34] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[35] K. Yelick, D. Bonachea, and C. Wallace. A proposal for a UPC memory consistency model. Technical Report LBNL-54983, Lawrence Berkeley National Laboratory, May 2004.

[36] K. Yelick et al. Titanium: a high performance Java dialect. In *proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.

[37] Y. Zhu and L. J. Hendren. Communication optimizations for parallel c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, pages 199–211, 1998.