Building a Source-to-Source UPC-to-C Translator

Wei-Yu Chen

Report No. UCB/CSD-4-1369

December 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Contents

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 Berkeley UPC Compiler is an open source and high-performance implementation of the language. The choice of C as the code generation target greatly enhances the compiler's portability, as evidenced by its support of a wide range of supercomputing platforms, including large-scale multiprocessors, vector machines, and network of workstations. In this paper I describe the translator component of the Berkeley UPC Compiler, which is responsible for performing UPC-to-C transformation and generating the necessary runtime calls for communication. The goal of the translator is to generate high quality C code while enabling easy porting of the compiler, and also provide a framework that allows for extensive high-level optimizations. We use a combination of micro-benchmarks and application kernels to show that our compiler can output C code that achieves good performance on both superscalar and vector environments, despite the source-to-source transformation process. We also investigate several communication optimizations, specifically targeting two optimizations can get significantly improve the performance of fine-grained programs: message coalescing and split-phase communication generation.

1 Introduction

Global Address Space (GAS) languages have recently emerged as a promising alternative to the traditional message passing model for parallel applications. Designed as parallel extensions for popular sequential programming languages, GAS languages such as Unified Parallel C [25], Titanium [48, 28], and Co-Array Fortran [39] 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. GAS languages thus offer a more convenient and productive programming style than explicit message passing (e.g., MPI [36]), and good performance can still be achieved because programmers retain explicit control of data placement and load balancing. Another virtue of GAS languages is their versatility; for example, UPC's flexible memory model is carefully designed so that it can operate in both shared and distributed memory environments. While it has not yet reached the level of MPI's ubiquity, UPC implementations are now available on a significant number of platforms [20, 26, 43], ranging from multiprocessors to the many flavors of networks of workstations.

The Berkeley UPC Project is a research effort aimed at increasing the visibility of the UPC language, by building a portable compiler framework that also offers comparable performance to other commercial UPC implementations. To achieve portability and high performance, the Berkeley UPC compiler uses a layered design, which can be tailored to adapt to the communication primitives and processor architectures offered by different platforms. Specifically, the compiler generates C code that contains calls to our UPC runtime interface [7], which is implemented atop a language-independent communication layer called GASNet [9].

In this report, we describe our experiences with designing and implementing the source-to-source translator in the Berkeley UPC Compiler [6]. The translator is derived from the Open64 Compiler Suite [40], an open-source collection of optimizing compiler tools that can compile C,C++, and Fortran programs. The translator helps achieve portability by both targeting C as its output format and employing a transformation process that is mostly platform independent with the exception of a few architecture specific parameters. Generating code for a new platform thus can be as simple as changing the particular values of such parameters in a configuration file. An equally important goal of the translator is to ensure that the sequential portion of an UPC program, which is written just like regular C code, does not experience performance slowdown after the source-to-source translation. By keeping the internal representation sufficiently high-level, our Berkeley UPC translator is able to generate C output that closely resembles the original source. Empirical evidence suggests that our translator can generate good-quality C output that performs well on both scalar architectures and vector machines.

The second part of this report concentrates on our efforts in using the translator as a framework for experimenting with high-level UPC specific optimizations. 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. Specifically, the optimizations should reduce both the number and the volumes of message traffic, as well as hide communication latencies by overlapping communication with computation. We have designed and implemented several optimizations in our translator that enable fine-grained UPC programs to be compiled more efficiently. The first optimization eliminates runtime affinity tests associated with UPC's parallel *forall* loop construct. The second optimization, *message coalescing*, transforms fine-grained memory accesses into bulk transfers to reduce communication overhead. Finally, the translator performs split-phase communication scheduling to exploit both communication-computation overlap and message pipelining. Preliminary results suggest that these optimizations are generally effective in reducing the communication costs.

The rest of the paper is organized as follows. Section 2 describes the UPC language and provides an overview of the Berkeley UPC Compiler. Section 3 presents the source-to-source transformation process of the Berkeley UPC-to-C translator. Section 4 discusses our strategies for dealing with the several challenges encountered while implementing the translator, while Section 5 examines the code generation quality of the translator, focusing on its ability to maintain the vectorizability of the program. Section 6 gives a summary of the Berkeley UPC translator's optimization framework. Section 7 presents our optimization work with the UPC *forall* loop, while Section 8 describes another loop optimization called message coalescing. Section 9 details the translator's strategy for split-phase communication generation. Section 10 lists the related work, and Section 11 concludes the paper.

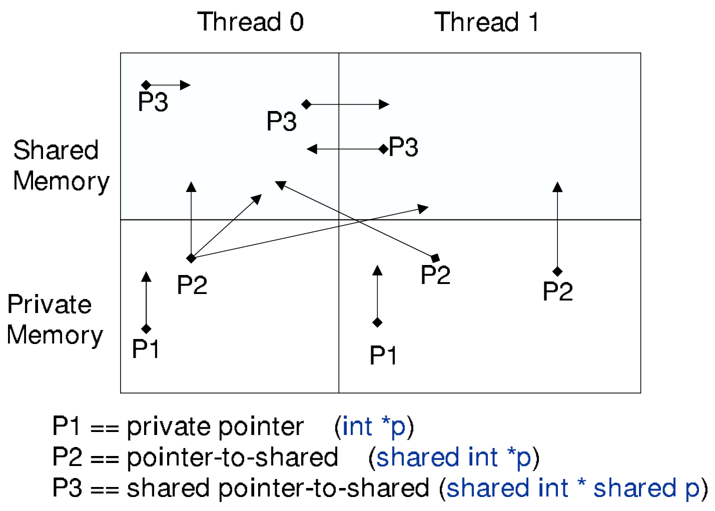

Thread 0
Thread 1
Shared Memory
Private Memory
P3
P3
P3
P2
P2
P2
P1
P1
P1 == private pointer (int *p)
P2 == pointer-to-shared (shared int *p)
P3 == shared pointer-to-shared (shared int * shared p)

Figure 1: Different type of UPC pointers.

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. Each thread has a unique integer identity expressed as the `MYTHREAD` variable, 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 data in the shared address space. The shared memory space is logically divided among all threads, so 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.

Pointers in UPC can be classified based on the locations of the pointers and

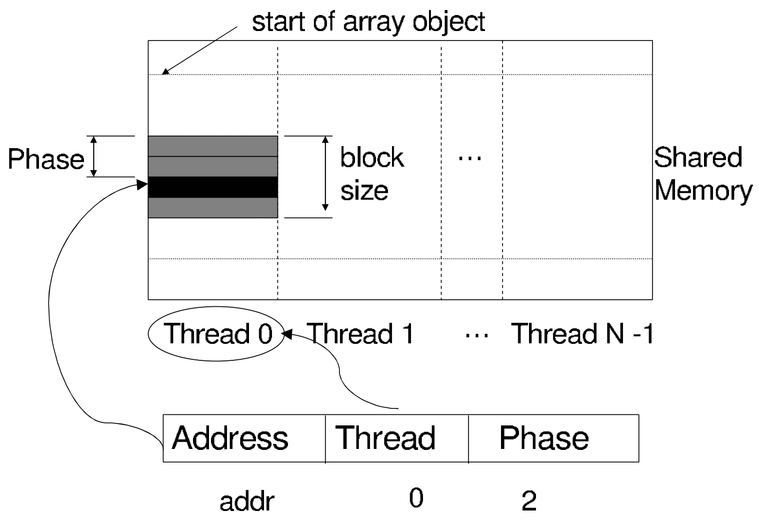


Figure 2: UPC pointer-to-shared components.

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 1 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

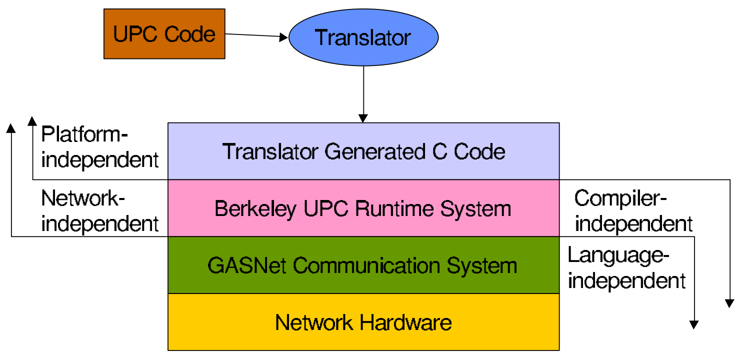


Figure 3: Architecture of the Berkeley UPC Compiler

object's "local" address on the thread, while the `phase` field gives the offset of the object within its block. Figure 2 demonstrates how the fields in a pointer-to-shared are used to access a shared value. In summary, a UPC pointer-to-shared thus can be classified into three categories based on the data layout: *block cyclic*, *cyclic*, and *indefinite*.

Another interesting UPC feature is its support for both a strict and a relaxed memory consistency model. Every shared variable access in UPC is type qualified as either "strict" or "relaxed", either explicitly or inferred from pragmas. The strict memory model is analogous to sequential consistency in that it requires the actual execution of the accesses on each thread to be consistent with program order, 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 [38]. The goal of the UPC memory model is to effectively exploit the tradeoff between programmability and performance; relaxed accesses offer better performance as they can be aggressively optimized by compilers as long as local data dependency on each thread is still preserved, but programmers are left with the burden of ensuring that their code is free of race conditions. Other notable features of UPC language include dynamic allocation functions, synchronization constructs, and a builtin parallel loop construct. The UPC language specification describes them in more details [25].

2.2 The Berkeley UPC Compiler

Figure 3 shows the overall structure of the Berkeley UPC Compiler [6], 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 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. 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. In addition to the portability benefits, the layered design also means that each component can be implemented and performance-tuned individually. The backend C compiler is free to aggressively optimize the intermediate C output, and the UPC-to-C translator can utilize its UPC-specific knowledge about shared memory access patterns to perform 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 The UPC-to-C Translator: An Overview

Like the Berkeley UPC Compiler, the UPC-to-C translator is also divided into three components: the front end, the back end, and whirl2c. Figure 4 depicts the translator's compilation process.

- Front end: Upon receiving a preprocessed UPC file, the translator's front end parses and type checks the input, and generates a high level WHIRL (Open64's intermediate representation) file. UPC-specific information such as shared types and block size for distributed arrays are preserved in the symbol table, so that the later translator phases can utilize the information in performing optimization and code generation. We have also extended the

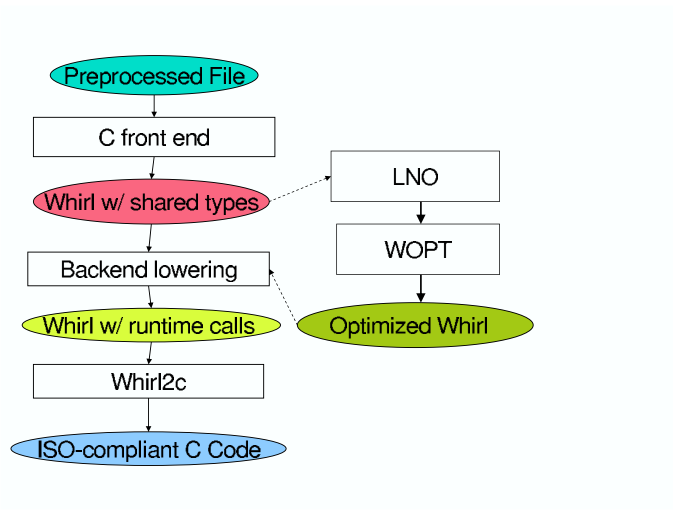


Figure 4: UPC-to-C Translation Process

front end code base with several features from ISO C99 standard [12] such as mid-block declarations and declaration expressions in a for loop header.

- Back end: The primary functionality of the back-end is to convert expressions involving a pointer-to-shared into the appropriate runtime library calls. Specifically, pointer arithmetic on a shared address is converted into function calls; the translator selects one of the runtime's three different functions for shared address calculation, based on the blocksize of the pointer-to-shared. Similarly, loads and stores of shared variables may require communication and are also transformed into runtime calls. The actual runtime function invoked again depends on a number of factors such as the type being loaded and whether the shared memory access is strict or relaxed. An optional optimization phase, which includes both a loop nest optimizer (LNO) and a general-purpose global scalar optimizer (WOPT), can be invoked before the lowering of shared expressions. Details about the optimization phase are presented in Section 6.

- Whirl2c: The final component's job is to convert the WHIRL representation into ISO-compliant C code, with shared pointers declared as opaque UPC pointer-to-shared types that are defined internally in the runtime system. This enables us to experiment with different pointer-to-shared representations in the runtime system without having to modify the translator. This capability has proved useful on platforms such as the Cray X1, where we can more efficiently implement pointer-to-shared operations by exploiting the hardware global address space support [5]. Whirl2c attempts to generate high-level C language constructs when possible (e.g., using struct member accesses in favor of pointer arithmetic), so that its output will bear sufficient resemblance to the source code. This stage also provides special support for static and global shared variables, which can not be initialized statically as their storage is not allocated until runtime. Finally, an indirect access scheme is adopted for applications running with POSIX threads so that each pthread gets its own private copy of thread-local variables, and whirl2c is responsible for generating the address translation macros when accessing such variables in the program.

4 Code Generation Issues

In this section, we describe several code generation issues encountered during our implementation of the Berkeley UPC translator, and our approaches for handling them.

4.1 Portable Code Generation

Since the Berkeley UPC translator outputs C code, it avoids difficulties associated with cross compilation that are often encountered when compiling for a variety of target systems. The infrastructure of the translator is flexible enough that it can perform code generation for both 32-bit and 64-bit platforms, and we have also extended the front end to obtain the values of architectural dependent parameters such as integral type width and struct alignment rules in a configuration file. In general, the generated code for different platforms will be identical (modulo differences introduced by the system headers) with the exception of parameters such as the size of pointers and primitive types that are either explicitly referenced in the program or implicitly used during the generation of runtime calls. Thus, while Open64 represents the scalar types internally by their type size (e.g., on IA-32 int and long would be considered equivalent since they are both four bytes), different integral types will remain distinct in the output to avoid both C compiler warnings and unsafe integer downcasts. Another important platform-specific parameter is the size of UPC pointer-to-shared object, which is usually different from that of regular C pointers. This leads to a subtle issue for structs containing pointer-to-shared, as their size needs to be adjusted as do the offsets of field accesses into such structs. The offset padding is performed in the backend, so that the preceding analysis and optimization phase of the translator can treat pointer-to-shared as regular pointers.

Another challenge introduced by the choice of C as the translation target concerns header file conflicts. The translated code must include the Berkeley UPC runtime header file to access the library function prototypes, and the runtime header in turn includes a variety of the standard C library headers to implement tasks such as I/O, timing, and communication. Consequently, duplicate type and variable declarations may occur if the UPC program itself also includes the standard C headers, and the translator emits their contents in the output. Our solution is to distinguish the files that contain UPC constructs and those that do not, and avoid outputting any variables and struct types for the latter, as such ordinary C files can be safely recompiled by the backend C compiler. Instead, the *#include* directives for these files are regenerated in the same relative order in the translated code, taking advantage of the fact that C library headers are guarded against reinclusion. Furthermore, only the toplevel inclusions (files explicitly included by the user) are reinserted into the output file. The Berkeley UPC compiler automatically recognizes files containing UPC constructs (whose contents thus must be emitted) by checking for the presence of shared expressions and declarations in a file and its recursively included contents, and no user intervention is therefore required. One code pattern that our "reinclude" scheme does not handle correctly arises when the behavior of a *#include* header depends upon the prior presence of macro variable definitions; a common example is the use of the `NDEBUG` variable to turn off assertions in a program. While this problem can potentially be fixed by also preserving macro definitions in the output, we have elected not to pursue this option, since such a fix would require the translator to have its own preprocessor and may thus have adverse effects on the compiler's portability.

4.2 Handling Shared Expressions

Pointer-to-shared variables in UPC are almost as expressive as normal C pointers (one exception is that pointers to shared functions have undefined behavior), and can generally appear anywhere in the program where it is legal for a C pointer. Shared expressions that must be converted into equivalent runtime functions by the Berkeley UPC translator include shared memory accesses, pointer-to-shared arithmetic, as well as equality tests and cast operations involving pointers-to-shared. Shared memory accesses can be easily recognized internally by the type of the operands. For accesses to shared objects with integral or floating point types that can fit inside a register, the translator generates memory-to-register runtime communication calls to avoid unnecessary local memory operations, while other accesses such as struct copying are translated into the more general memory-to-memory puts and gets, with the translator responsible for spilling out stack allocated temporaries to obtain a lvalue. Depending on whether the memory access is strict or relaxed, the translator must also choose between the blocking and non-blocking variants of the communication functions. Pointer-to-shared arithmetic expressions are always marked internally with a preceding type cast, so that they can be quickly identified and transformed into the appropriate runtime calls in the translator's lowering phase; explicit casts of pointer-to-shared into private types are handled in the same manner. The translator does not make any assumptions about the internal layout of the pointer-to-shared object while performing the above transformations, so the Berkeley UPC runtime layer is free to select the pointer-to-shared representation most suitable for the target platform.

4.3 Code Generation Example

Figure 5 provides an example on how UPC programs are translated into C code. The UPC code fragment performs matrix-vector multiplication between a distributed two-dimensional array and a shared vector located on thread 0. In the translated C code (for brevity, declaration and allocation of the shared arrays and temporary variables are omitted), both shared pointer arithmetic expressions and shared memory accesses are converted to runtime calls with the "upcr" prefix. Different runtime operations are chosen for the two shared variables, as mat is distributed block cyclically while vec is declared as an indefinite array. Since both shared arrays have relaxed type, the translator generates one-sided nonblocking communication calls to fetch the remote values. The nonblocking `GET_NB` call issues a request for the communication subsystem to begin the data transfer and return a handle, which later can be used in a synchronizing call (`WAIT_SYNCNB`) to wait for the completion of the request. Shared variable writes can be implemented analogously. The example shows but a small subset of functions available in our runtime layer, and the translator also generates code for other communication patterns such as bulk transfer and blocking operations.

```
#define ROW 10
#define N 10000
shared [N] double mat[ROW*THREADS][N];
shared [] double vec[N];

...

 double sum;
 sum = 0;
 for (int i = MYTHREAD;
      i < ROW*THREADS;
      i+=THREADS) {
   for (int j = 0; j < N; j++) {
     sum += mat[i][j] * vec[j];
   }
 }
```

Original UPC Code

```
sum = 0.0;
 i = ((int) upcr_mythread () );
 while(i <= 9)
 {
  j = 0;
  while(j <= 9999)
  {
    _bupc_Mptra0 = UPCR_ADD_PSHAREDI(vec, 8, j);
    _bupc_Msync2 = (upcr_handle_t) UPCR_GET_NB_PSHARED(
               &_bupc_spillld1, _bupc_Mptra0, 0, 8);
    UPCR_WAIT_SYNCNB(_bupc_Msync2);
    _bupc_Mptra3 = UPCR_ADD_SHARED(
                mat, 8, (_UINT32)(i) * 10000U, 10000);
    _bupc_Mptra4 = UPCR_ADD_SHARED(
               _bupc_Mptra3, 8, j, 10000);
    _bupc_Msync6 = (upcr_handle_t)
               UPCR_GET_NB_SHARED(
               &_bupc_spillld5, _bupc_Mptra4, 0, 8);
   UPCR_WAIT_SYNCNB(_bupc_Msync6);
    _2 :;
    j = j + 1;
  }
  _1 :;
  i = i + 1;
 }
```

Translator C Output (with one thread)

Figure 5: UPC-to-C Translation Process

5 Translator Output Performance – Vectorization

The popular GAS languages are designed as parallel extensions of sequential programming languages, and UPC is no exception. A thread's local computation in its private address space is generally written in a language very similar to ordinary C code, and therefore uniprocessor execution time is an important criteria in evaluating a UPC compiler's performance [24]. Although our translator preserves the semantics of the sequential portions of the program, it is infeasible to expect the translated output to be syntactically identical to the program source, due to optimizations performed by the translator and the lack of a one-to-one mapping between its intermediate representation and the C language. In previous work [16] we have discovered that despite a source-to-source translation from UPC to C, our compiler still delivers good serial performance on conventional superscalar architectures. It is less clear, however, whether such syntactic discrepancies will have a performance impact on vector platforms such as the Cray X1 [22], whose dramatically different architectural approach makes vectorization the dominant factor for achieving high performance. A common performance attribute of parallel vector systems is that the vector unit executes substantially faster than its scalar counterpart; for the Cray X1, in addition to operating at twice the clock speed, its ability to overlap memory operations with vector computation makes the vector unit significantly more powerful than the scalar pipeline. Furthermore, the Cray C compiler's vectorizer [21] is sensitive to changes in inner loop expressions; our experiments have identified several constructs that tend to inhibit a loop's vectorization, such as function calls, type casts, the address-of operator, and access to global variables in the presence of pointer arithmetic. One interesting metric of the translator's code generation quality is thus its serial performance on a vector architecture. Specifically, it is worth investigating whether the translator's code generation process can be extended to minimize interferences with the C compiler's ability to automatically vectorize application code.

5.1 Implementation Approach

Our goal is to evaluate the serial performance of the Berkeley UPC compiler, concentrating on its ability to maintain the vectorizability of the sequential portion of the program. With full optimizations enabled, the Cray C compiler [21] performs automatic vectorization on expressions inside a loop that it detects to be free of cycles of dependences, after applying vectorization-enabling transformations such as inlining, loop splitting, and loop interchange. The compiler also vectorizes certain special recurrences such as reductions and scatter/gather operations. Cray C provides two program-level techniques to assist the compiler's alias and dependence analysis: `restrict` pointers and the pragmas that declare a loop to be free of vector dependences or recurrences between array accesses. As such, our strategy is to keep the translated output as syntactically similar as possible to the original source. The level of the intermediate representation is kept sufficiently high such that C loops are preserved in their original form. Similarly, array expressions are recognized and handled specially by the translator, both to allow for more aggressive transformations by its optimizer and to provide the C compiler with more precise information. Multidimensional arrays are preserved in its original form instead of being linearized into one dimensional arrays. The Berkeley UPC compiler supports `restrict`-qualified pointers, and additionally UPC source-level vectorization pragmas are accepted by the translator and appear unchanged in the same relative location in the generated C output.

	Geo. Mean	Avg. Rate	Har. Mean	Max	Min
C	160	756	58.7	6561	9.0
UPC	161.9	762	59.6	6652	9.0

Table 1: Aggregate performance of the Livermore Loops (in MFLOPS)

5.2 Livermore Kernels

We use the C version of the Livermore Kernels [35] to evaluate the serial performance of our compiler. The Livermore Loops consist of 24 sequential computation loops extracted from common scientific applications, and should closely reflect the sequential computational performance offered by our compiler. In particular, the X1's reliance on the vector unit to achieve both fast computation and high memory bandwidth means that application performance will often hinge on whether the main computation loops can be efficiently vectorized. In this test, we do not supply any vectorization pragmas and do not perform any manual transformations, as our goal is to test if the translation process interferes with Cray C's automatic vectorization. Table 1 presents the aggregate performance for both the original C source and the translated output with the `-O3` flag, while Figure 6 displays the normalized performance of the individual kernels.

As Table 1 shows, Berkeley UPC's translated output performs almost identically to the original C source code. Performance results from the individual benchmarks confirm with this observation; the ratio of UPC running time versus C running time is within 5% for nearly all of the kernels, which can be attributed to measurment noises. One notable exception occurs in kernel 8, where Berkeley UPC's output surprisingly outperforms the C code by about 10%. Examination of the translated output suggests that its performance benefits from the Berkeley UPC translator recognizing several three dimensional array accesses in the loop as common subexpressions and replacing them with stack temporaries. The introduction of the stack variables does not affect vectorization, and saves three address calculations per iteration. Because the translated output exhibits similar performance to the C code for most of the kernels, we expect the Berkeley UPC compiler to offer competitive serial performance on a vector platform like the X1.

6 Optimization Framework

Having presented the source-to-source compilation strategy and evaluated the sequential performance of the Berkeley UPC-to-C translator, in the second part of the paper we focus on the translator's parallel performance. In particular, we de-

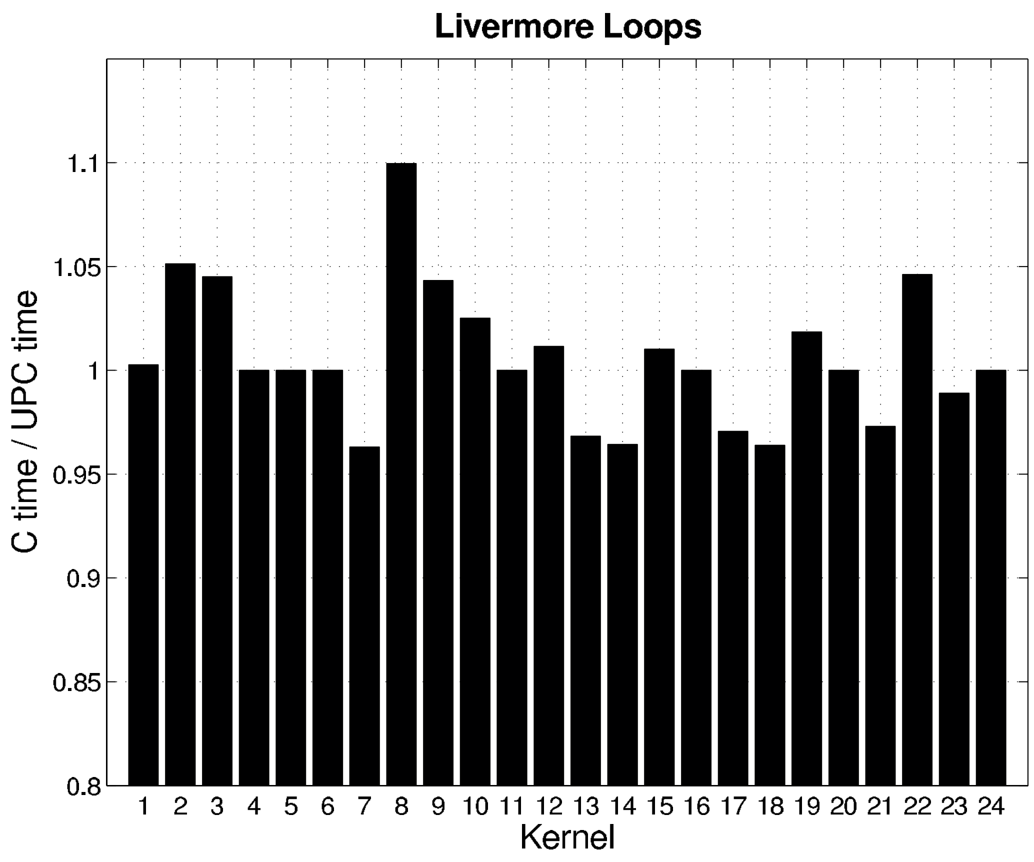


Figure 6: Performance of the individual Livermore kernels

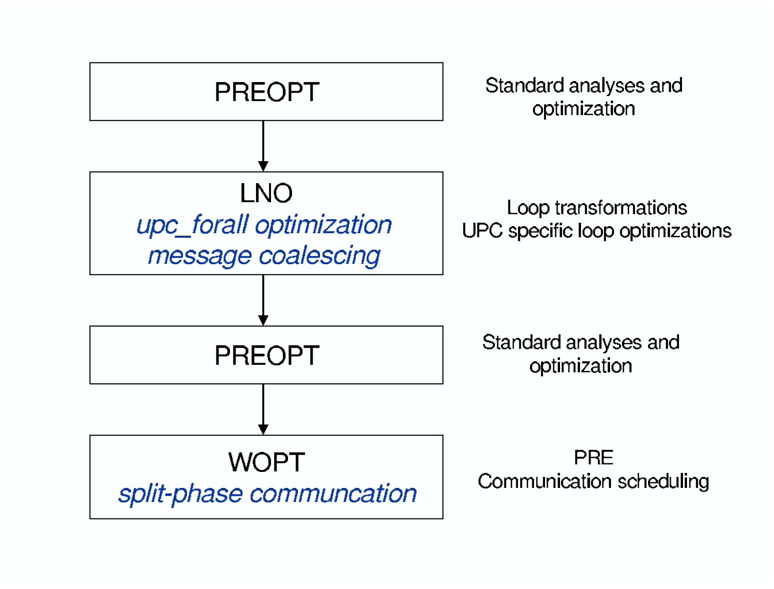


Figure 7: Optimization framework of the translator

scribe the preliminary experiences with designing and implementing a number of optimizations specifically targeted at improving UPC communication performance. One of the reasons Open64 was chosen as the code base of our translator is its extensive collection of optimizations. Not only are standard compiler optimizations such as copy propagation, partial redundancy elimination, and dead code elimination supported through its global scalar optimizer (WOPT), but Open64 also provides a high level loop nest optimizer (LNO) that recognizes and applies various loop transformations and optimizations. A framework for interprocedural analysis and optimizations is also available. For our source-to-source transformation several of Open64's optimizations are not directly applicable, since they either produce outputs that are too low-level to be expressed in C (e.g., prefetching individual loads), or can be performed equally well by the backend compiler. Therefore, while employing many of Open64's large repertoire of optimizations in the compilation process, we also have supplied several optimizations that require UPC specific knowledge and therefore could not be performed by a C compiler.

Figure 7 summarizes the overall structure of the Berkeley UPC-to-C translator's optimization framework. The process starts from Open64's Preopt optimizer [34], which serves as the front end of both the loop nest optimizer (LNO) and the global optimizer (WOPT). Accepting program input in WHIRL format, Preopt builds a control flow graph and a Static Single Assignment (SSA) represen-

Name	System	Network	CPU
Seaborg	IBM RS/6000 SP	SP Switch 2	375MHz Power 3+
Flyer	Compaq Alphaserver ES45	Quadrics Elan 3	1GHz Alpha
Ram	SGI Altix	shared memory	1.5 Itanium 2

Table 2: Machine summary

tation, and performs a number of high-level optimizations such as copy propagation and dead code elimination. After the optimizations are completed, it converts the program back into WHIRL form, and generates the def-use chain information for LNO. The LNO component performs transformations such as fusion, interchange, and tiling on loops that have the semantics of Fortran DO Loops [47]. Specifically, a DO loop contains a single index variable, the condition expression is a comparison on the value of the index variable, and the lower bound, upper bound, and stride of the loop are all loop-invariant. We have introduced two UPC-specific optimizations at the end of the LNO phase. The first optimization, described in Section 7, focuses on eliminating the runtime overhead of UPC's parallel `forall` loop, while the second optimization, message coalescing, is explained in detail in Section 8. WOPT is invoked after LNO, again with Preopt as the front end to perform the necessary analysis. Instead of running Open64's SSAPRE optimization [17], which produces low-level WHIRL nodes that can not be safely translated into C code, however, we have implemented our own algorithm that targets shared memory accesses and pointer arithmetic expressions in UPC. Also based on the SSA representation, our algorithm combines partial redundancy elimination with communication scheduling to automatically generate split-phase communication calls. The algorithm is presented in Section 9. Table 2 contains a summary of the machines where the experiments on the optimizations are performed.

7 Optimizing UPC Parallel Loop

To simplify the task of parallel programming, UPC 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 of 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.

UPC forall loops provide a convenient syntactic sugar for the purposes of thread coordination and preventing inadvertent remote accesses, but its primary drawback is the runtime overhead incurred by the affinity tests. Not only do these affinity tests have to be executed on each iteration by all threads, but the presence of the branches in the loop can also inhibit many useful loop optimizations. Fortunately, while their values naturally changes from iteration to iteration, affinity expressions can often be derived directly from loop induction variables; for such common special cases, we can eliminate the runtime affinity tests by incorporating their thread-iteration mapping constraints into the loop's bound and stride.

```
for(i = L; i < U; i ++; i+c)
  ...
```

```
int ofst = MOD(MYTHREAD - L - c, THREADS);
for(i = L+ofst; i < U; i += THREADS)
  ...
```

Figure 8: Forall Loop Affinity Test Removal

Our optimization operates on forall loops with unit stride (either 1 or -1); optimizing loops with non-unit stride is possible, but would involve more expensive operations such as greatest-common-divisor, and such forall loops occur relatively infrequently in practice. Another precondition is that the loop must be recognized by the loop nest optimizer as a DO loop. If the affinity expression is an integral expression of the form $i + c$, where i is the primary induction variable and c is loop-invariant, we apply the transformation illustrated in Figure 8, to yield an equivalent for loop with the affinity test eliminated. The MOD operation performs modular arithmetic, and $MOD(a, b)$ returns a value between 0 and $b - 1$ if b is positive. When the affinity expression is a shared address, we focus on the common special case of the form $\&a[i]$, where a is a shared array or pointer and i the induction variable. Once the base address a is established to be loop invariant, three transformations are available to eliminate the affinity tests and can be chosen depending on a's blocksize. In the trivial case when a is indefinite, all of its elements will be on the same thread, and we simply need to test the variable's affinity once before executing the loop. If a is cyclic, the affinity expression is equivalent to $i + threadof(a)$, where the second operand computes the thread that $a[0]$ locates on; the loop then is optimized with the transformation shown in Figure 8. Finally if a is block cyclic, a two-level nested loop can be used in place of the original forall loop. The outer loop will take a stride of $THREADS * BLOCKSIZE(a)$, while the inner loop iterates through the executing thread's block in unit stride.

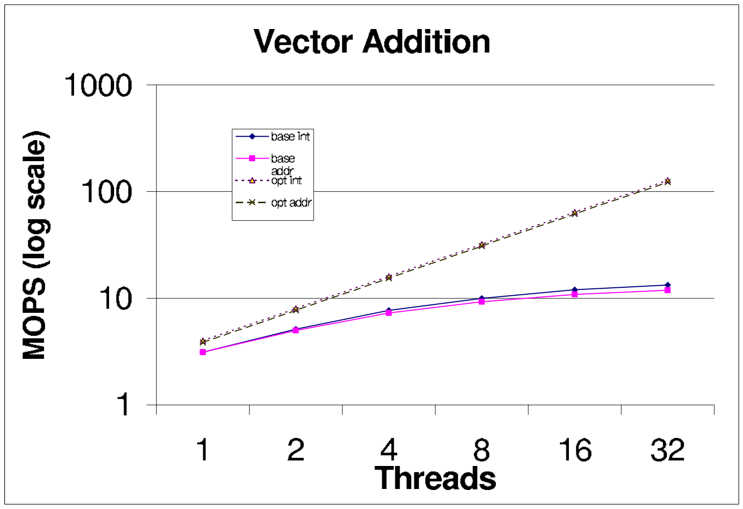


Figure 9: Vector Addition – Millions of Additions per Second

A vector addition benchmark is used to illustrate the performance gain resulting from the affinity test removal. The program uses a forall loop to add two cyclic arrays element by element, storing results to another shared cyclic array. As the benchmark contains a minimal amount of computation, affinity test overhead can contribute significantly to the forall loop's execution cost. We experiment with both integer and shared address affinity expressions for the loop. The results, presented in Figure 9, were collected on a 256-processor SGI Altix system with Itanium2 processors [14]. Removing runtime affinity tests substantially increases the performance of the benchmark, delivering a more than 20% speedup for the sequential case. More importantly, it significantly improves the program's scalability, since each thread no longer has to execute iterations that do no useful work other than the affinity tests. Whereas the unoptimized scales poorly even with a small number of threads, the optimized output achieves linear speedup.

Finally, since our optimization accepts only a few specific affinity expression patterns (namely $i + c$ and $\&a[i]$), one natural question is whether the technique may be too restrictive as to exclude a large number of forall loops found in common UPC programs. We seek the answer by analyzing the results of applying our optimizations to the Berkeley UPC Compiler's regression testsuite. The translator identifies 122 forall loops from the testsuite that contain an affinity expression, and the affinity tests are eliminated from 80% of them, suggesting that our optimization is powerful enough to capture most common usages of UPC forall loops.

8 Message Coalescing: Implementation and Results

```
shared [] int *r;
...
for(i = L; i < U; i += s)
  e1 = e2 + r[i];
```

Figure 10: Unoptimized loop.

```
int size = (U - L) * sizeof(int);
int *lr = malloc(size);
upc_memget(lr, r, size);
for(i = L; i < U; i += s)
  e1 = e2 + lr[i];
free(lr);
```

Figure 11: Loop after Message Coalescing.

Empirical data on the overhead and latency of today's high-performance networks speak volumes about the effectiveness of message coalescing and aggregation [4]; by combining small puts and gets into large messages, one can save significantly on the per-message startup overheads. The most common realization of this optimization, called *message coalescing*, significantly improves the performance of a fine-grained loop by fetching all the remote values it needs in a single bulk transfer outside the loop instead of issuing fine-grained read operations in every iteration. As a well known and extremely important optimization for amortizing the cost of small message traffic, message coalescing has been implemented in a number of compilers for Fortran-like languages [27, 46]. In this section, we describe the implementations of message coalescing in the Berkeley UPC translator.

8.1 Analysis for the Optimization

Our message coalescing optimization again operates on loops that have been marked by the loop nest optimizer as possessing the semantics of Fortran DO loops. To ensure that the resulting transformations do not violate the UPC memory model, the analysis further prohibits synchronization statements, function calls, and strict memory accesses from appearing in the loop body. Once a loop has been identified as a potential coalescing candidate, the analysis walks through the loop expressions and build for each distinct array symbol a (lo, up) bounding box on the range of its possible index values, which must be affine expressions of the index variable. For example, if $ar[i]$ and $ar[i + c]$ are both present in the loop body, a region is computed for the symbol ar by taking the union of the ranges projected by the two index expressions. The array symbols are not limited to variables with array types and can include pointers, provided that the pointer object is not modified in the loop. Coalescing-inhibiting array dependences are detected by intersecting the *def* and *use* sets of the loop.

8.2 Implementation Sketch

The choice of code generation for message coalescing varies significantly based on the layout and the access pattern of the shared array in question. At the language level, UPC provides several point-to-point bulk communication library functions that perform reads and writes on contiguous memory regions between the private and shared address space. To handle non-contiguous transfers, we have recently proposed library extensions that support both indexed and strided memory accesses [10], and reference implementations already exist in the Berkeley UPC runtime. Our message coalescing optimization targets both interfaces and is thus able to support a variety of array access patterns commonly observed in scientific applications.

In the simplest case, the shared array is one-dimensional and resides exclusively on one thread (called *indefinitely blocked* arrays in UPC), so that only one bulk communication call is required to fetch $up - lo + 1$ elements from the remote array.

Figure 10 and 11 illustrates the application of message coalescing for such a scenario[1]. Once the remote data is transfered into a local private buffer, the optimization can replace all references to the original shared array with equivalent accesses to the private array. When the lower bound is not a zero constant, the

[1]for brevity, some safety check code (e.g., checking loop is executed at least once) are omitted

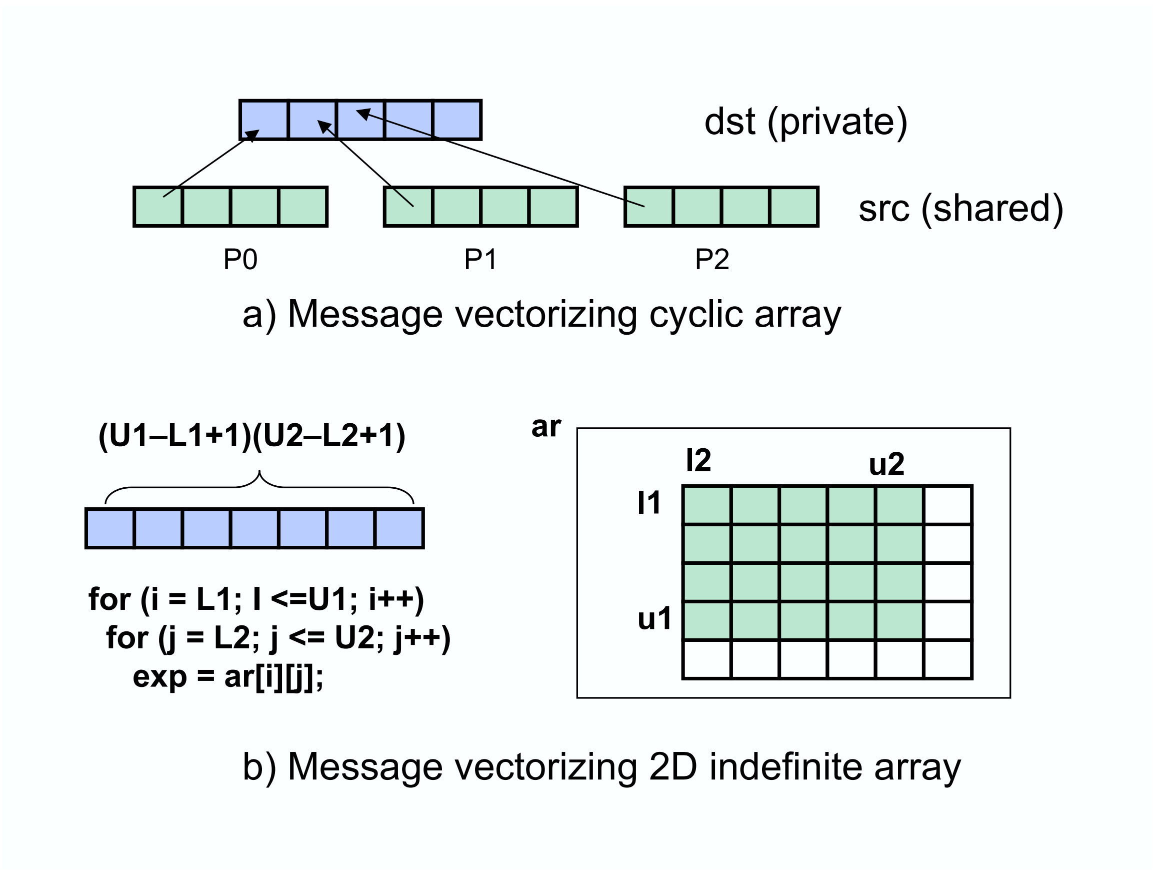

dst (private)
src (shared)
P0
P1
P2
a) Message vectorizing cyclic array
(U1–L1+1)(U2–L2+1)
for (i = L1; I <=U1; i++)
for (j = L2; j <= U2; j++)
exp = ar[i][j];
ar
l2
u2
l1
u1
b) Message vectorizing 2D indefinite array

Figure 12: Message Coalescing

index expressions used by the local array must also be adjusted by subtracting the lower bound's value, as $shared\_ar[i]$ refers to the same content as $local\_ar[i-lo]$. In the common special case where both the lower and upper bound of the loop are compile-time constants, a stack array is allocated instead of calling malloc() to save the heap management overhead. Finally, a unit-strided writes could also be vectorized in the same manner, except that upc_memput must be invoked at loop exit to store the new values of the shared array.

In a more advanced scenario, the shared array is still one-dimensional but is now (block) cyclically distributed, so that data may need to be obtained from multiple threads. Our optimization handles this case by generating a point-to-point bulk `upc_memget` call to each individual thread, copying remote data in units of blocks to simplify the shared address calculation. A temporary two-dimensional private array is allocated to serve as the destination buffer of the memget calls, with $ceil(number\_of\_blocks/threads) * blocksize$ elements for each thread. For maximal performance, the different memgets to the individual threads are overlapped to hide their communication latencies. After the bulk communication calls have completed, a final step copies the data from the two-dimensional temporary buffer into an one-dimensional local array following UPC shared pointer arithmetic rules (i.e., first block of thread 0 is copied first, followed by the first block of thread 1, and so on), so that the private array can be accessed with the same index expressions used by the original shared array. Figure 12a depicts graphically the code generation steps of vectorizing accesses to an one-dimensional block cyclic array.

For the previous two access patterns, the accessed elements on the individual threads are always contiguous, and it thus suffices to copy them with the existing UPC point-to-point memcpy library functions. For a two-dimensional array traversed inside a two-level loop nest, however, such code generation strategy may no longer be appropriate, since not all of the columns and rows of the array will be accessed. If the two-dimensional array is indefinite, our optimization can more efficiently vectorize the code by utilizing the strided memcpy functions in our runtime interface. These functions transfer a list of fixed sized regions with a single fixed stride separating them, and accept as arguments the starting address, region length, distance (stride) between the regions, and the number of regions of both the source and the destination array. This interface simplifies the code generation of vectorized two-dimensional indefinite array accesses: the region length is simply the number of columns to be copied, number of regions is the row count, and the stride is just the length of the inner dimension of the array. Thus, for the sample code shown in Figure 12b, our analysis will compute a bounding box for each dimension of the shared array, and use the strided memget function to transfer elements in the rectangular box into a private buffer. An alternative code generation scheme we considered is to combine message coalescing with pipelining; the inner loop is message coalesced by fetching one entire row in a single bulk call, while the outer loop is software pipelined so that the communication overhead of the bulk transfers can be overlapped. The performance tradeoff between the two methods will depend on factors such as the latency and bandwidth of the underlying networks, and thus vary from platform to platform.

Our current implementation could optimize most one-dimensional array access in a single loop nest, but more complicated communication patterns are still unsupported. On the top of our priority list is to generalize the analysis to handle multidimensional arrays that are either block cyclically distributed or have non-unit-stride accesses. Such access patterns are inappropriate for the *bound* methods mentioned above, which involve retrieving a bound box that contain the needed elements; a potentially large number of remote elements may be fetched but never used, resulting in a substantial performance penalty. Instead, we need to opt for a more general approach by passing a list of fixed-size (one element) regions to the Berkeley UPC runtime, which will be responsible for performing the required communications to the different threads. Another future plan is to support irregular array access patterns (e.g., $a[b[i]]$) found in sparse matrix vector multiply applications with the inspector-executor model [45].

8.3 Preliminary Evaluation

One advantage Global Address Space languages such as UPC have over message-passing based programming models is that communication can be conveniently expressed as reads and writes to the shared memory space, which allows programs to easily build shared data structures. Studies [8] have pointed out, however, the severe performance issues that shared-memory style UPC applications face on distributed memory platforms, due to the excessive amount of small message traffic generated. As a result, experienced UPC programmers usually code their applications with coarse-grained parallelism to ensure performance portability, even when the application logic can be expressed more naturally with fine-grained communication. By automatically transforming fine-grained shared reads and writes into bulk communication calls, message coalescing holds great potentials in narrowing the performance gap between the shared memory programming style and the coarse-grained communication paradigm for UPC programs. The effectiveness of message coalescing can thus be evaluated both in terms of performance and programmability. Not only do we seek to demonstrate that message coalescing of fine-grained code could perform as well as hand optimized code, but we also want to assess the strength of the analysis by examining UPC benchmarks to count the number of loops that can benefit from the optimization, thereby relieving application developers from the burden of manually converting fine-grained accesses into bulk communication calls.

8.3.1 Performance

To evaluate the performance of message coalescing, we present data from a simple dense matrix-vector multiply benchmark in UPC. The matrix is partitioned cyclically by row, and we experiment with both an indefinite (entirely located on thread 0) and a cyclic distribution of the vector. Three different code configurations were compared: a fine-grained version in which each thread has to repeatedly fetch the remote vector elements to perform its local computations, the compiler-optimized code of the fine-grained loop, and finally a coarse-grained version of the benchmark that fetches the entire vector once and saves the values in a local buffer.

Figure 13 presents the performance of the matrix-vector multiply under the different configurations. The results were collected on the machine Flyer and Seaborg in Table 2. As expected, the message coalesced code significantly outperforms the naive version by more than two orders of magnitude, as remote values may need to fetched in every iteration of the inner loop in the latter case. Also not surprisingly, the message coalesced version performs quite similarly to the manually optimized version, given the close resemblance of their code and the same bulk communication pattern they employ. One notable distinction between message coalescing and user's manual optimizations is that the former can target the nonblocking bulk communication calls provided by the Berkeley UPC runtime, a feature that is not yet available at the language level. When the vector is cyclically distributed, the message coalescing optimization is able to exploit this ability to overlap communications to different threads and thus substantially outperform bulk style UPC code on Flyer's Quadrics network. From the results of the benchmark, we therefore conclude that for fine-grained accesses to one dimensional shared arrays, message coalescing can produce code that performs as well as the manually optimized code, regardless of whether the array is cyclically or indefinitely distributed.

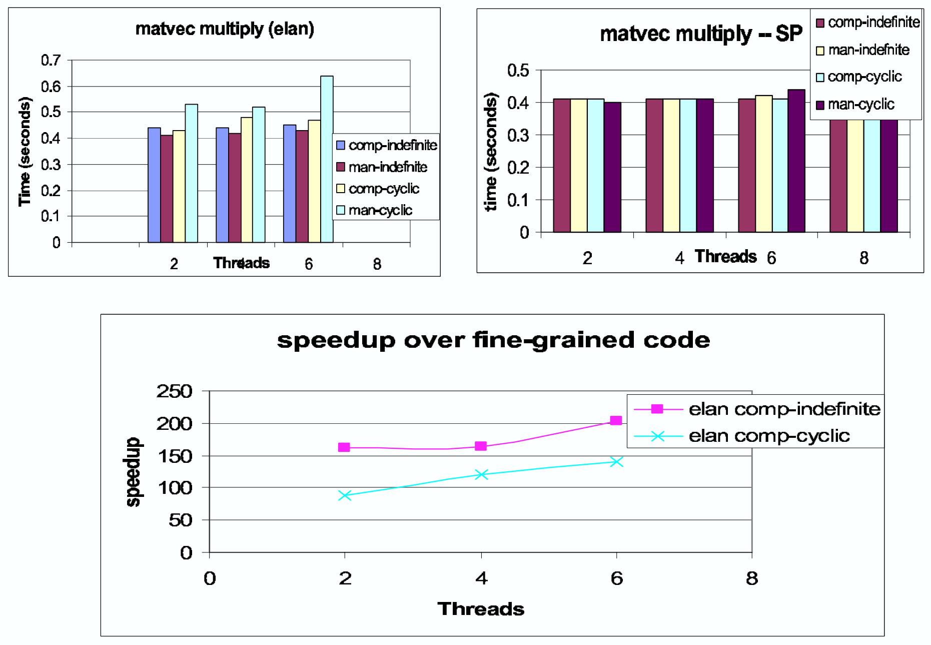


Figure 13: Performance Results of Matrix-Vector Multiply

8.3.2 Programmability

Good performance of the generated code alone does not make message coalescing a useful UPC optimization. If the analysis is too conservative, programmers will still be forced to manually optimize their fine-grained loops due to fears that they may not be recognized as message coalescing candidates. To evaluate the accuracy of the analysis, we examine several of the UPC NAS parallel benchmarks [3] (IS, MG, FT, CG) and report instances of fine-grained loops that can be successfully message coalesced by our optimizations. Since for performance reasons most of the benchmarks have been implemented using coarse-grained communication, we additionally manually convert the bulk communication calls (e.g., `upc_memget`) into fine-grained shared memory accesses in a loop, and see if they can be message coalesced.

```
shared[] dcomplex *shared reshuffle_arr_shd[THREADS];
...
// Broadcast the shared pointers
for (i = 0; i < THREADS; i++)
  reshuffle_arr[i] = reshuffle_arr_shd[i];
```

Figure 14: UPC code from FT benchmark that can be message coalesced

In the IS benchmark, which is written in bulk synchronous style, we discovered that all three of the `upc_memgets` in the program can be message-coalesced when converted back into loop form. Similarly, our analysis is strong enough to recognize all four memgets in the MG benchmark as message-vectorizable when the calls are rewritten as fine-grained accesses in a loop. While such transformations will not result in performance gains compared to the original benchmarks, it does mean that our optimization can relieve programmers the trouble of explicitly generating the bulk communication calls. More encouragingly, both the FT and the CG benchmark contain loops that perform fine-grained remote accesses to broadcast the elements of a (block) cyclic array to all threads (Figure 14); since UPC does not support multi-node communication routines at the language level, code patterns such as broadcast and reduction could not easily be expressed with bulk transfers. The Berkeley UPC translator, however, is able to automatically apply message coalescing on these fine-grained loops to hoist the required communication outside of the loops.

9 Automatic Split-phase Communication Generation

On machines with no hardware global address space support, remote data access in UPC eventually needs to be compiled into one-sided communication calls. Due to the asynchronous nature of message-passing networks, such one-sided communication routines are generally implemented in two phases. As the earlier example in Section 4.3 shows, one straightforward translation of a remote memory access is to have the `init_op()` call followed immediately by the corresponding `sync_op()` call. This code generation guarantees correctness, since all shared memory accesses will execute in program order. The disadvantage, however, is that remote memory accesses generally have very high latencies, and valuable processor cycles can be wasted while waiting for the remote transfer.

In order to hide communication latency, optimizing compilers need to leverage the 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 apart from each other in the program as possible, while preserving data and control dependencies. This minimizes the chance 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. We describe an algorithm that combines partial redundancy elimination and communication scheduling to automatically generate split-phase communication. Based on an SSA representation, the algorithm targets shared variable loads, reducing the number of messages and exploiting communication and computation overlap simultaneously. The optimization can be applied to both scalar and pointer accesses, and takes advantage of the SSA form in Open64 to efficiently place the split-phase communication operations.

9.1 Algorithm Sketch

The analyses is performed on Open64's Hashed SSA (HSSA) representation [18], which uses global value numbering to build a sparse representation of the entire program. A hash table is used to store all program expressions, and expressions with the same value number share the same hash entry. The HSSA representation also extends the original SSA form with the ability to support aliases, so that indirect memory operations (the C dereference operator *) can be modeled as if they are scalar variables. We further assume that control flow analysis has been performed on the program.

9.1.1 Optimize Shared Pointer Arithmetic

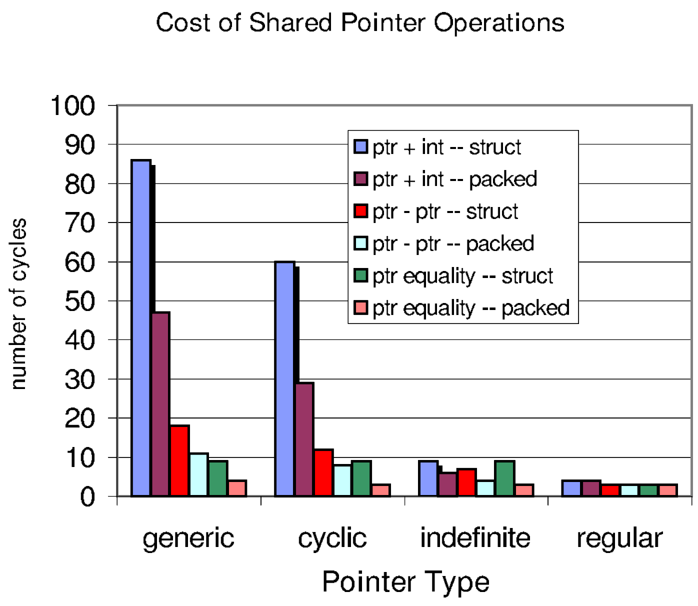


Figure 15: Cost of Shared Memory Operations

Before analyzing the split-phase communication placement, we perform partial redundancy elimination on shared pointer arithmetic expressions. 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. Figure 15 shows the cost of shared pointer arithmetic operations on a Compaq Alphaserver ES45 node, with a 1-GHz processor running the Tru64 operating system [32]. Two pointer-to-shared representations are included in the experiments; one declares the pointer-to-shared as a C struct, while the other more efficient format represents the pointer as a packed eight-byte integer. As Figure 15 shows, shared pointer arithmetic is an expensive operation even for the packed representation. Eliminating redundant shared address calculation is therefore an important optimization, especially when the C compiler likely will have difficulties performing it. Even if the runtime functions implementing shared pointer arithmetic operations are fully inlined, it is still unrealistic to expect the C compiler to optimize the function body the same way it could for an expression.

The analysis begins with a mark phase that iterates through all statements in a function and find nodes, or *use points* of shared pointer arithmetic expressions. If the expression is used more than once in the program (each static occurrence of the expression counts as a use point), we determine the earliest point in the function where the expression can be computed. This can be performed in two steps. First, we collect the *definition point* for all of the variables and indirect loads that appear in the expression; 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 will serve 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.

This use-def information is all that is necessary to perform optimization. At a shared pointer arithmetic expression's definition point, we compute the value of the optimized expression and assign it to a newly created variable. All occurrences of the expressions are then replaced with the temporary. While this optimization is not always profitable (e.g., the occurences of the expression may all be on different paths), the speculation is safe since pointer arithmetic operations will not raise exceptions.

9.1.2 Split-phase Communication for reads

Having eliminated redundant shared pointer arithmetic expressions, we next shift our attention to optimizing shared reads. The first step of the analysis is similar to the previous case, as we also compute the single definition point for every shared reads in the function. A major difference, however, is that we cannot simply place the nonblocking communication call at the definition point, since it may effectively place a shared load expression on a path that does not perform the read in the original code. This speculative code motion is incorrect because executing a load on an invalid address (for indirect loads) will generate an exception. Furthermore, the nonblocking remote read operation in our communication system performs RDMA (remote direct memory access) to copy the remote data directly into a stack-allocated temporary. This means all outstanding nonblocking reads must be synchronized before a function returns to avoid memory corruption, even if the return value is never used by the program. The spurious message traffic can have a significant performance penalty that outweighs the benefits of the optimization.

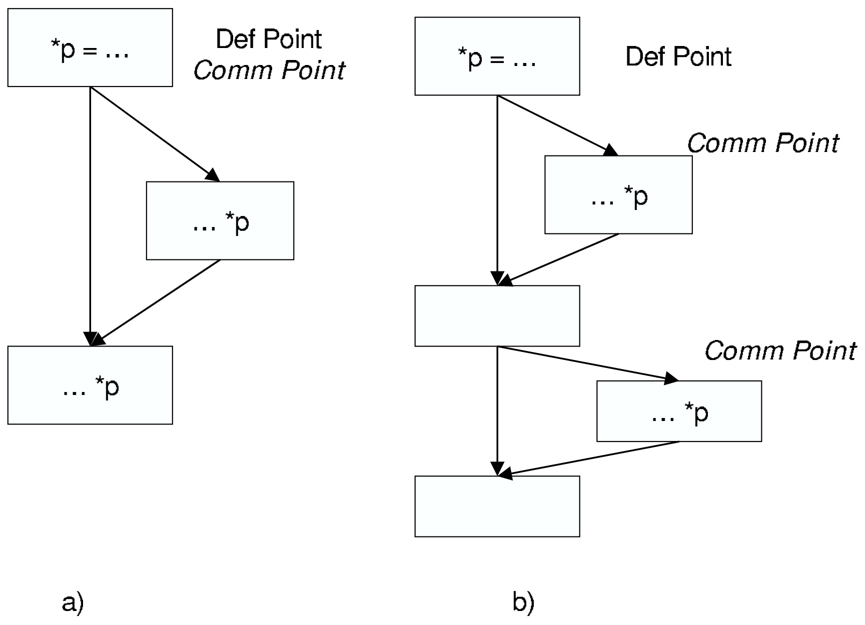


Figure 16: Split-phase Communication Analysis. Communication points correspond to init calls, actual use sync calls

To prevent speculative code motion, we rely on the concept of *anticipated expressions* [37]. An expression is said to be *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 load expression e must thus be anticipatible at the point where we insert the non-blocking communication call. To efficiently compute this information, we divide the use points of e into groups based on their basic blocks. We associate every use group with a *communication point* p, with the property that p is dominated by the definition point, and the expression is anticipatible after p. Since all uses in a group belong to the same basic block, the point immediately before the first use in the basic block trivially satisfies this property. To maximize the amount of overlap, though, we begin the search starting from the definition point of e, to identify the earliest program point that meets the requirement.

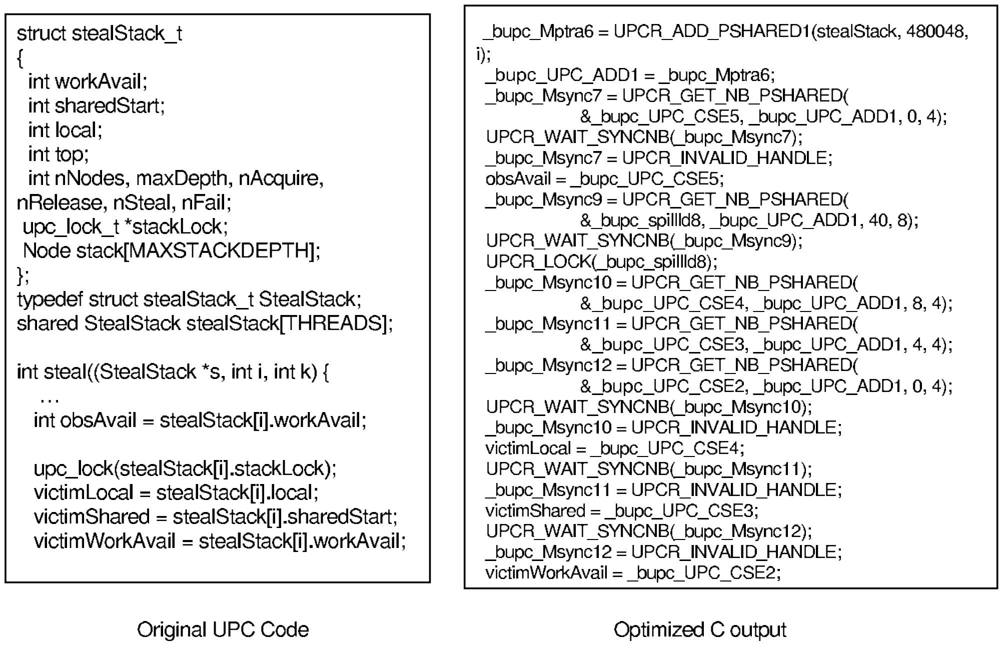
The communication points from each use group represent the locations where it is safe to insert the nonblocking operation. We can further reduce the number of redundant message by omitting communication at point a if a is dominated by another point from a different use group. The corresponding synchronizing calls are then inserted immediately before every use of the expression. To ensure that no nonblocking calls are synchronized more than once, we also invalidate the handle after each synchronization call. Figure 16 shows how the communication points are identified. In a), the load expression is anticipatible at the definition point, which thus also serves as the single communication point. For b), two non-blocking calls are required, since neither of the communication points dominate the other, and we want to prevent speculative code motion.

9.2 Preserving the UPC Consistency Model

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 employs barrier synchronization to divide a program into different phases, and accesses from different phases must execute in program order. Furthermore, UPC supports a strict memory consistency model, which requires all threads to agree on a total order over the strict operations. For the purposes of our optimizations, this prevents the compiler from performing code motion that would reorder strict reads and writes with any other memory accesses. To prevent accidental reordering caused by our optimization, we model strict accesses as barrier statements that may modify every shared variable in the program. This effectively inhibits any code motion that moves relaxed shared load expressions across strict accesses.

9.3 Optimization Example

Figure 17 provides a concrete example of how the optimization performs redundancy elimination and communication scheduling. The code is extracted from a fine-grained UPC benchmark that performs parallel unbalanced tree search [42]. The shared arithmetic expression $stealStack[t]$, which is computed five time in the original UPC program, has been replaced with a temporary variable, eliminating all redundant address computations. The three individual reads following the lock operation are also pipelined to reduce their communication overhead. The



```
struct stealStack_t
{
  int workAvail;
  int sharedStart;
  int local;
  int top;
  int nNodes, maxDepth, nAcquire,
nRelease, nSteal, nFail;
 upc_lock_t *stackLock;
 Node stack[MAXSTACKDEPTH];
};
typedef struct stealStack_t StealStack;
shared StealStack stealStack[THREADS];

int steal((StealStack *s, int i, int k) {
   ...
   int obsAvail = stealStack[i].workAvail;

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

Original UPC Code

```
 _bupc_Mptra6 = UPCR_ADD_PSHARED1(stealStack, 480048,
i);
 _bupc_UPC_ADD1 = _bupc_Mptra6;
 _bupc_Msync7 = UPCR_GET_NB_PSHARED(
              &_bupc_UPC_CSE5, _bupc_UPC_ADD1, 0, 4);
 UPCR_WAIT_SYNCNB(_bupc_Msync7);
 _bupc_Msync7 = UPCR_INVALID_HANDLE;
 obsAvail = _bupc_UPC_CSE5;
 _bupc_Msync9 = UPCR_GET_NB_PSHARED(
              &_bupc_spillld8, _bupc_UPC_ADD1, 40, 8);
 UPCR_WAIT_SYNCNB(_bupc_Msync9);
 UPCR_LOCK(_bupc_spillld8);
 _bupc_Msync10 = UPCR_GET_NB_PSHARED(
              &_bupc_UPC_CSE4, _bupc_UPC_ADD1, 8, 4);
 _bupc_Msync11 = UPCR_GET_NB_PSHARED(
              &_bupc_UPC_CSE3, _bupc_UPC_ADD1, 4, 4);
 _bupc_Msync12 = UPCR_GET_NB_PSHARED(
              &_bupc_UPC_CSE2, _bupc_UPC_ADD1, 0, 4);
 UPCR_WAIT_SYNCNB(_bupc_Msync10);
 _bupc_Msync10 = UPCR_INVALID_HANDLE;
 victimLocal = _bupc_UPC_CSE4;
 UPCR_WAIT_SYNCNB(_bupc_Msync11);
 _bupc_Msync11 = UPCR_INVALID_HANDLE;
 victimShared = _bupc_UPC_CSE3;
 UPCR_WAIT_SYNCNB(_bupc_Msync12);
 _bupc_Msync12 = UPCR_INVALID_HANDLE;
 victimWorkAvail = _bupc_UPC_CSE2;
```

Optimized C output

Figure 17: Sample Code from Optimized Programs

optimization also correctly conforms to the UPC memory model by not issuing any of the pipelined reads before the lock call.

9.4 Preliminary Results

Figure 18 presents the performance improvement achieved by our optimizations in two fine-grained UPC benchmarks. In the graph we refer to the unoptimized version as *base*, and the optimized version as *opt*. The results were collected on an HP machine called Flyer, described in Table 2.

Gups: This communication-intensive benchmark performs random access to a distributed shared array. For the optimized program, we manually unrolled the loop that performs the shared reads, so that it will benefit from the effects of read pipelining. When the loop is unrolled 4 times, performance improves by up to 20% due to the effects of message pipelining. When the loop is further unrolled so that 8 messages are now pipelined, performance increases by an additional

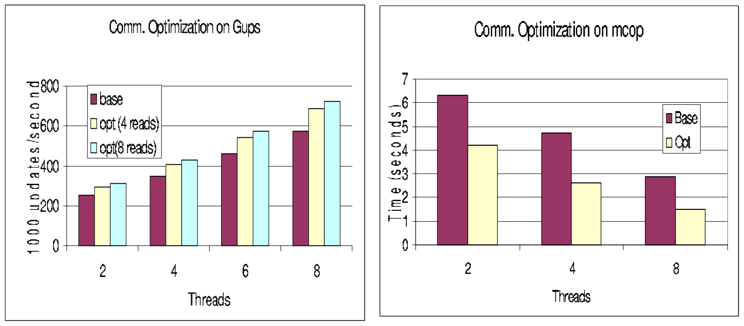


Figure 18: Performance Improvement

5%. This matches our expectation on the diminishing returns of unrolling loops to achieve message pipelining.

Mcop: This benchmark solves a problem called matrix chain ordering [11] in UPC. The optimized version again takes advantage of read pipelining by issuing four remote reads at the same time to overlap them. The performance speedup ranges from 50% to 90% for this benchmark, growing as the number of threads increases.

10 Related Work

In addition to the Berkeley UPC compiler, there are several UPC implementations available on a variety of platforms. These compilers include the HP UPC Compiler [20], the GCC-based Intrepid Compiler [26], the MuPC runtime system [43], and the Cray UPC Compiler [21]. El-Ghazawi et al. [13, 24] have evaluated the performance of some of the above compilers with NAS parallel benchmarks, and found that they can generally offer comparable performance to the MPI version of the NAS benchmarks. In particular, the HP UPC compiler offers a number of communication optimizations such as software caching, as does the MuPC runtime system.

Several research efforts [41, 31, 33] have used Open64 as an open source platform for compiler research and application development. Among these projects, the one most closely related to the research goals of the Berkeley UPC Compiler is the Co-Array Fortran Group at Rice University [19], which aims to provide a portable, retargetable, and high-quality implementation of the global address space language that is also based on Open64. To achieve portability, their compiler performs source-to-source translation from Co-Array Fortran to Fortran 90 with calls to runtime library primitives, while high performance is to be attained by employing optimization strategies with platform specific communication cost models.

For today's distributed memory machines, the overhead of accessing remote data is usually orders of magnitude higher than local memory accesses. This drastic performance gap has motivated numerous research works that aim to reduce communication overhead through automatic compiler optimizations. In general, communication optimization techniques can be classified into two categories: those that attempt to hide communication latencies through overlapping (e.g., prefetching, pipelined communication), and those that try to reduce the number and volume of message traffic (e.g., message coalescing). Traditionally the optimization problems have been studied along with communication code generation in parallelizing compilers. For example, Amarasinghe and Lam [2] use dataflow analysis on array elements to automatically parallelize a program and perform optimizations that eliminate redundant messages. Kandemir et al. [29] use a combination of dataflow analysis and linear algebra framework to perform optimizations such as message vectorization and message coalescing in the PARADIGM compiler. Similarly, a number of optimizations including communication vectorization and pipelining have been implemented in the HPF compiler [1].

In the context of communication optimizations that overlap communication and computation, perhaps the prior research that is most closely related to our techniques is Hendren and Zhu's work on parallel C programs [49]. Their analysis framework is based on the concept of possible-placement analysis, which identifies the earliest possible point to issue a remote read, and delays the issuing of a remote write to exploit opportunities for blocked communication. Chakrabarti et al. [15] have implemented a global communication scheduling algorithm for High Perfomrance Fortran that handles remote accesses in an interdependent manner. They have also explored using late placement to expose more opportunities for combining messages.

Krishnamurthy and Yelick [30] also presented compiler analysis and optimizations for explicitly parallel Split-C [23] programs with a global address space. Most of their work focuses on improving the accuracy and efficiency of the cycle detection [44] algorithm for SPMD programs, which enforces sequential consistency under reordering transformation. The optimizations presented in this report also guarantees that the consistency model of the language would not be violated, and our optimization framework can be augmented with their cycle detection algorithm to allow for more opportunities at communication optimization in the presence of strict accesses.

11 Conclusion

We have described in this report the design and implementation of the Berkeley UPC-to-C translator, which is an essential component in the Berkeley UPC compiler, a portable high-performance implementation of the UPC language. Our contributions in this paper can be summarized as the following:

- We have demonstrated that with a source-to-source translation strategy, we can build a portable compilation framework for UPC that works on a broad range of platforms, including shared memory machines, vector systems, clusters, and other hybrid architectures.

- We have also established that performance need not be sacrificed for the sake of portability. Despite the source-to-source translation, the Berkeley UPC compiler can achieve good serial performance even on a vector platform.

- We have implemented several communication optimizations for fine-grained UPC programs in the translator, and evaluated their effectiveness on a number of benchmarks. One optimization, message coalescing, improves performance by combining small messages into bulk transfers. Another optimization, split-phase communication, utilizes communication and computation overlapping to hide communication latency. Preliminary results suggest that the optimizations can be very effective at reducing the communication costs of fine-grained UPC applications, especially on distributed memory machines. Furthermore, we believe both optimizations can apply to any Global Address Space Languages that perform one-side communications.

Ultimately, the metric of success for the UPC language will be its degrees of acceptance in the parallel computing community. An open-source UPC compiler that offers both portability and good performance will contribute significantly toward the goal of promoting UPC. As the Berkeley UPC compiler matures, we believe it would eventually exert a positive influence on the development of other UPC compilers and on continuing language development, similar to GCC's role on C/C++ development. By offering an implementation that is freely available on a wide range architectures, the Berkeley UPC compiler can help set the bar on performance that other vendor compilers should meet or surpass on their supported platforms. As we continue to improve the robustness and performance of the Berkeley UPC compiler, we can also encourage other compilers with more incentives to enhance their implementations. This healthy competition will increase the qualities of UPC implementations overall, thereby attracting more users for UPC.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[2] S. Amarasinghe and S. Lam. Communicaton optimization and code generation for distributed memory machines. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, June 1993.

[3] 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.

[4] 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.

[5] C. Bell, W. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the cray x1. In *19th Annual International Conference on Supercomputing (ICS)*, June 2004.

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

[7] *The Berkeley UPC Runtime Specification*, 2003. http://upc.lbl.gov/docs/system/upcr.pdf.

[8] K. Berlin, J. Huan, M. Jacob, et al. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

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

[10] D. Bonachea. Proposal for extending the UPC memory copy library functions, 2004.

[11] P. Bradford, G. Rawlins, and G. Shannon. Efficient matrix chain ordering in polylog time. *SIAM Journal on Computing*, 27(2), 1998.

[12] Programming Languages – C, 1999. The ISO C Standard, ISO/IEC 9899:1999.

[13] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[14] CCS: SGI altix. http://www.ccs.ornl.gov/Ram/Ram.html.

[15] 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.

[16] 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.

[17] 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.

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

[19] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

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

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

[22] Cray X1 system overview. http://www.cray.com/craydoc/20/manuals/S-2346-23/html-S-2346-23/S-2346-23-toc.html.

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

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

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

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

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

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

[29] 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.

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

[31] Compiler research and the kylin project. http://www.capsl.udel.edu/kylin/.

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

[33] J. Lin et al. A compiler framework and speculative analysis and optimizations. In *SIGPLAN'03 Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[34] S. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, October 1996.

[35] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

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

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

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

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

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

[41] Open research compiler. http://ipf-orc.sourceforge.net/.

[42] 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.

[43] 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.

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

[45] J. Su and K. Yelick. Array prefetching for irregular array accesses in titanium. In *Sixth Annual Workshop on Java for Parallel and Distributed Computing*, 2004.

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

[47] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium (MICRO-29)*, December 1996.

[48] 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.

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