Optimizing Partitioned Global Address Space Programs for Cluster Architectures

Wei-Yu Chen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-140
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-140.html

December 4, 2007

Optimizing Partitioned Global Address Space Programs for Cluster Architectures

by

Wei-Yu Chen

B.S. (University of California, Berkeley) 2000
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine A. Yelick, Chair
Professor Rastislav Bodik
Professor Gregory Fenves

Fall 2007

The dissertation of Wei-Yu Chen is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Optimizing Partitioned Global Address Space Programs for Cluster Architectures

by

Wei-Yu Chen

Abstract

Optimizing Partitioned Global Address Space Programs for Cluster Architectures

by

Wei-Yu Chen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine A. Yelick, Chair

Unified Parallel C (UPC) is an example of a *partitioned global address space* language for high performance parallel computing. This programming model enables application to be written in a shared memory style, but still allows the programmer to control data layout and the assignment of work to processors. An open question is whether programs written in simple style, with fine-grained accesses and blocking communication, can achieve performance approaching that of hand-optimized code, especially for cluster environments with high network latencies.

This dissertation proposes an optimization framework for UPC that automates the transformations currently performed manually by programmers. The goals of the optimizations are twofold: not only do we seek to aggregate fine-grained remote accesses to reduce the number and volume of message traffic, but we also want to overlap communication with computation to hide network latency. The framework first applies communication vectorization and strip-mining to optimize regular array accesses in loops. For irregular fine-grained accesses, we apply a partial redundancy elimination framework that also generates

split-phase communication. The last phase targets the blocking bulk transfers in the program by utilizing runtime support to automatically schedule them and achieve overlap. Message aggregation is performed as part of the scheduling to further reduce the communication overhead. Finally, we present several techniques for optimizing the serial performance of a UPC program, in order to reduce both the overhead of UPC-specific constructs and our source-to-source translation.

The optimization framework has been implemented in the Berkeley UPC compiler, and has been tested on a number of supercomputer clusters. The optimizations are validated on a variety of benchmarks exhibiting different communication patterns, from bulk synchronous benchmarks to dynamic shared data structure code. Experimental results reveal that our framework offers comparable performance to aggressive manual optimization, and can achieve significant speedup compared to the fine-grained and blocking communication code that programmers find much easier to implement. Our framework is completely transparent to the user, and therefore improves productivity by freeing programmers from the details of communication management.

Chair Date

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I am deeply grateful to my advisor Kathy Yelick, as this dissertation would not have been completed without her guidance for the past five years. She introduced me to the area of parallel computing, taught me how to write papers and do research, and always managed to find time in her busy schedule to guide and mentor my work.

I am also thankful for the interactions with other members of my dissertation committee. Ras Bodik was kind enough to read both my Master's thesis and later this dissertation, and provided many valuable suggestions from the PL angle. Greg Fenves offered big-picture insights on how my optimizations could be applicable to read world scientific applications.

I would also like to thank my colleagues in the Berkeley UPC group, including Christian Bell, Dan Bonachea, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Rajesh Nishtala, and Mike Welcome. My research has benefited tremendously from their work, and they have been important co-authors on several papers that form the basis of this dissertation. Costin Iancu deserves special thanks for his crucial contributions on the Berkeley UPC translator. I really enjoyed the friendship of my office mates over the years (Christian Bell, Dan Bonachea, Kaushik Datta, Rajesh Nishtala, and Jimmy Su), and I can always count on them to answer my questions and read paper drafts.

Last but definitely not least, I would like to thank my family and friends for their continuous support. My parents gave me the opportunity to come to the United States, and their love and encouragements were the fuel that propelled me through six years of graduate school. Finally thanks go to my sister Irene, who despite having endured years of my whining is still someone that I can tell anything and everything to.

Chapter 1

Introduction

Today clusters are the dominant architecture in high performance computing arena due to their excellent cost/performance ratio. Clusters represent 75% of the systems in the most recent (June 2007) ranking on the world's top 500 fastest computers [101], with several boasting a processor count of more than 10,000. In the meantime, chip multiprocessors (CMP) have become part of mainstream computing, with Intel and AMD both transitioning their product line to quad-core processors. The prevailing hardware trends indicate that tomorrow's applications must embrace parallelism in order to leverage the power of the massively parallel computer systems in the future.

Writing parallel software is a notoriously difficult task, however, due to the correctness issues that arise from nondeterminism and the subtle performance characteristics of locality and parallelism. Today, most parallel software is written in one of two programming models: Two-sided message passing (i.e., MPI [79]) is used on large-scale machines and clusters, while shared memory programming (often with dynamically created threads

or OpenMP [85]) is used on smaller machines with hardware support for shared memory. Both models have their problems. Message passing gives programmers control over performance-critical features, such as locality and load balance, but the need to pack and unpack messages and coordinate between senders and receivers is tedious and sometimes awkward. Traditional shared memory programming offers flexible communication by allowing one thread to directly read and write into shared data structures, but lacks control over data layout or other forms of locality management that are essential to performance. The contrast between the two models reflects the challenges of finding the appropriate levels of abstraction for the underlying hardware. Exposing low level details about the architecture delivers performance, but hurts productivity by requiring programmers to make explicit decisions about communication management and data decomposition. On the other hand, a high level abstraction simplifies the task of parallel programming, but makes it more difficult for programmers to overcome the abstraction and achieve performance.

In this dissertation, we will examine a relatively new class of languages called Partitioned Global Address Space (PGAS) languages that offer some advantages of both models. The global address space abstraction supports shared data structures, but the space is partitioned into processor domains so that programmers can control data layout for performance. Each of the current PGAS languages is based on a popular sequential programming language, so that their syntax will appear familiar to many programmers. Unified Parallel C (UPC) [100] is an extension of ISO C, Co-Array Fortran [83] is an extension of Fortran 90, and Titanium [51, 111] is primarily an extension of Java, but with Java's thread features omitted.

Previous work has demonstrated that these languages offer significant advantages in

both programmability and performance relative to MPI [25, 31, 40, 43]. The programmability advantages come primarily from the global address space, as the communication code is much simpler when implemented as memory accesses. Additional advantages accrue from high level language support, such as distributed array abstractions. More surprising is the performance advantage of PGAS languages for some applications, which comes from their use of a *one-sided* communication model [12]. In one-sided communication, data transfers are decoupled from inter-thread synchronization, and this allows for more efficient use of cluster network hardware. For small and medium size messages, one-sided communication can outperform two-sided message passing by 20%-80% on networks with remote direct memory access (RDMA) support. PGAS languages thus offer a more convenient and productive programming style than explicit message passing, and good performance can still be achieved because programmers retain explicit control of data placement and load balancing. Another virtue of PGAS languages is their versatility, since they can run well on both shared and distributed memory machines.

In spite of the positive performance and programmability results for the PGAS languages, these studies have confirmed the need for hand-tuning of code. Programs that perform single-word remote reads and writes, while very simple to program, are not always practical on machines with high communication latency. Due to the loose coupling of the network interconnects and the microprocessors, fine-grained remote accesses are inherently expensive operations on cluster architectures; the cost of sending an eight-byte message is typically between $4 - 20us$ on today's high-performance networks [11]. Consequently, PGAS programmers sometimes need to invest substantial efforts to make their applications performance portable, by manually scheduling and combining remote accesses to reduce communication overhead. Such manual transformations can often be tedious and

error-prone, and negate some of the productivity advantages offered by PGAS languages.

1.1 An Optimizations Framework for PGAS Programs

The premise of this dissertation is that using sophisticated program analyses and optimizations, UPC programs written in a straightforward manner can attain performance approaching that of hand-optimized code. Our approach is to look at the optimizations that have been the most successful in hand-optimized code and devise techniques for automating them in the UPC compiler and runtime system. While some of these optimizations have enormous performance impacts, this work remains a significant departure from automatic parallelization, which is widely viewed as intractable for large-scale machines. In our setting, the application programmer is responsible for identifying parallelism and distributing data structures, while the compiler is responsible for selecting communication mechanisms and optimizing single node performance for UPC constructs.

This research in PGAS language analysis and optimizations uses the UPC language and the Berkeley UPC compiler as the vehicle for study, although many of the results are applicable to the other languages and compilers. The Berkeley UPC Project is a joint Lawrence Berkeley National Laboratory and UC Berkeley research effort aimed at increasing the portability of UPC programs by building a portable, open source compiler framework that also offers comparable performance to 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 a UPC runtime interface [13], which is implemented atop a language-independent communication layer called GASNet [16].

The cost of a message transfer on a cluster interconnect can be divided into two components: a per-message cost influenced by the latency of the network, and a per-byte cost influenced by the communication bandwidth of the network. This suggests that aggregating small messages into large ones can be an effective optimization, since it amortizes the fixed per-message costs and utilizes the high bandwidth provided by the network for large messages. Making the transfers nonblocking and overlapping them is another common optimization that is especially useful for PGAS programs, as the software overheads for one-sided accesses tend to be considerably lower compared to MPI messages. Our framework uses a combination of compiler and runtime support to implement both approaches for optimizing communication:

- *Communication aggregation*: Our framework primarily targets programs with fine-grained accesses and transforms them into coarser-grained transfers, but bulk communication calls to non-contiguous memory regions on the same processor can also benefit from our optimizations. This optimization reduces the number of remote accesses and amortizes communication latency.

- *Communication overlapping*: Our framework automatically transforms blocking remote accesses (either fine-grained or bulk) into nonblocking communication to overlap communication with computation and with other communication events. This optimization enables the processor to perform useful computation work instead of sitting idle while waiting for the remote transfer to complete. Overlap with computation may also reduce the effects of network contention by spreading communication

over a longer period of time.

The two principles of communication optimizations, aggregation and overlapping, may be incompatible in some situations. For example, aggregating small accesses into a single bulk transfer achieves better bandwidth, but could decrease the amount of communication and computation overlap available. Similarly, separating the initiation of a nonblocking access as far as possible from its completion maximizes the amount of overlap, but could reduce the opportunities for communication aggregation. The tradeoffs between the two approaches can be highly dependent on the particular system (e.g., send overhead, bandwidth, queue depth) and application (e.g., communication pattern, message size) in question. Balancing the two optimization goals is thus a challenge for our optimizations, as there is no complete solution that can guarantee optimal performance. Instead, we apply heuristics based on performance models to ensure that our framework can achieve performance that approaches and sometimes exceeds that of hand optimized code.

1.2 Overview of the Optimizations

Our framework consists of three communication optimization phases, plus miscellaneous optimizations that improve uniprocessor performance. The correctness of these optimizations relies on the relaxed UPC memory consistency model, as they change the order of shared memory operations, which may be visible to other threads.

- *Serial optimizations:* Although communication overhead tends to receive more attention when optimizing parallel programs, uniprocessor performance is equally important for achieving good overall performance, since scalable applications will spend

most of their execution time in local computation code. To achieve good serial performance, we carefully tune the compiler framework to generate good quality C output that can be optimized by the backend compiler. We also develop several optimization techniques to reduce the overhead of UPC specific constructs, including shared pointer manipulation functions and the **upc_forall** parallel loop. Chapter 4 summarizes these optimizations.

- *Optimizing fine-grained regular accesses:* In the first phase of our framework, we apply *message vectorization* to hoist individual array accesses out of loops and aggregate them into a single transfer. Message vectorization has been studied extensively in the context of data parallel languages, and traditional techniques try to aggregate as much as possible to reduce the number of messages. While minimizing message count is an important optimization goal, it can lead to missed opportunities for overlap. Our approach applies a transformation called *message strip-mining* that is more effective for PGAS programs. Message strip-mining attempts to reduce the overall communication costs by breaking up large message transfers into smaller ones that can be overlapped with computation. By deriving a performance model using synthetic benchmarks, we develop heuristics that enable strip-mining to significantly outperform the vectorized loop. The optimizations are described in Chapter 5.

- *Optimizing fine-grained irregular accesses:* For irregular pointer-based accesses that are not amenable to our array-based optimizations, we develop a separate static analysis framework that performs partial redundancy elimination (PRE) to both shared fine-grained accesses as well as shared pointer arithmetic. Code motion is applied as part of the framework to automatically generate split-phase communication, by sep-

arating the initiation of a remote access from its completion. During communication scheduling, small reads and writes that exhibit spatial locality (i.e., they belong to the same array or struct) may also be coalesced into a single transfer. Chapter 6 presents these optimizations.

- *Optimizing bulk memory transfers:* While our PRE framework is effective for fine-grained irregular accesses, it can have difficulties optimizing bulk communication routines due to the limitations in static array range and shape analysis, especially for C-based languages. Therefore, in the third phase of the framework, we develop runtime techniques to automatically transform blocking bulk transfers into nonblocking communication calls. Correctness is preserved via runtime conflict checks and double buffering, and the system also recognizes special access patterns and aggregates them to further improve performance. Chapter 7 presents our optimizations.

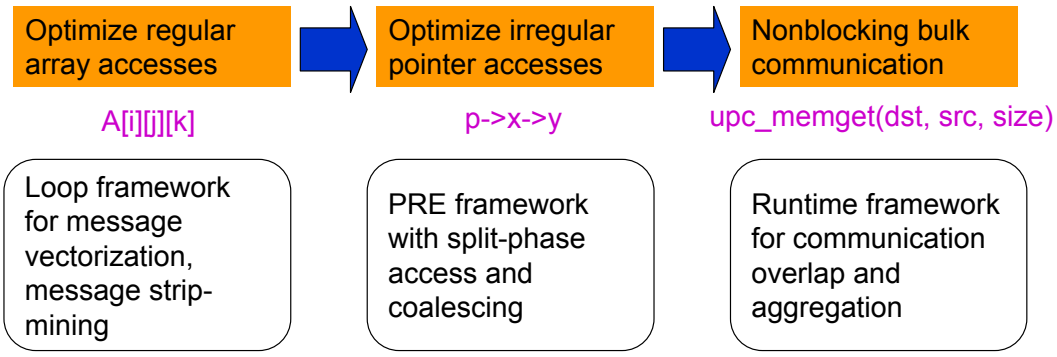


Figure 1.1: Summary of communication optimizations in our framework

Figure 1.1 summarizes the communication optimizations applied by our framework. Together the framework covers a wide range of communication patterns, from fine-grained

accesses to bulk synchronous transfers. All three phases combine the use of communication aggregation with communication and computation overlap in order to achieve the best performance. Each of the optimization phases operates independently, so that programmers can selectively enable the individual phases based on their application needs. More importantly, all of the optimizations are transparent to the user. Thus, while our optimization framework is not always as effective as aggressive manual transformations, it can improve productivity by freeing programmers from the details of communication management, which is also one of the major design goals for PGAS languages.

The optimizations are evaluated on a variety of benchmarks, ranging from the bulk synchronous benchmarks to dynamic shared data structure code with irregular accesses. Several supercomputer clusters were used in the experiments, covering the most popular system architectures and network interconnects. Experimental results suggest that all of the optimization phases are needed to maximize the framework's effectiveness. In terms of serial performance, despite the source-to-source code generation, the Berkeley UPC compiler suffers a less than 2.5% performance loss compared to sequential C code. Our optimizations are also effective at reducing the serial overhead associated with UPC-specific constructs such as shared pointer arithmetic and forall parallel loops.

In terms of communication optimizations, the message strip-mining transformation in Chapter 5 can bring up to a 40% speedup on micro-benchmarks and a 10-20% speedup on two of the NAS benchmarks compared to the reference implementation that uses blocking communication. The static optimization framework in Chapter 6 can achieve significant performance improvement for common fine-grained communication patterns, offering speedup as high as 80%. On a full computational fluid dynamics (CFD) application with

a lot of irregular accesses, our framework achieves a 10% speedup overall and a 30% speedup on the communication phase of the application. Finally, our runtime-based non-blocking transformation of bulk transfers delivers a near 30% performance improvement over the original blocking code, and is only slightly worse than manual optimizations.

Chapter 2

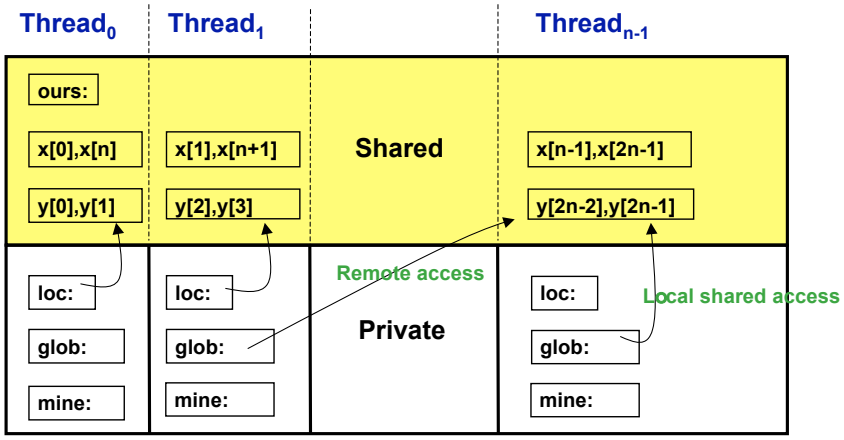
Background

In this chapter, we present a high level overview of the UPC language and the Berkeley UPC compiler, focusing on aspects relevant to the optimizations described in this dissertation. More details on the compiler can be found in [25] and [27], and the language specification provides a comprehensive definition of the UPC language [100].

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 Single-Program-Multiple-Data (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 runtime.

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. Figure 2.1 illustrates UPC's Partitioned Global Address Space model. While a thread can only access its own copy of the private variable `mine`, a single copy of the shared variable `ours` exists on thread zero and can be used by all threads.




```
#define n THREADS
shared int ours;
int mine;
shared double x[2*n];  //cyclic array
shared [2] double y[2*n];  //block cyclic array
double *loc;   //local pointer
shared double *glob; //gobal pointer (pointer-to-shared)
```

Figure 2.1: UPC memory model with sample variable declarations.

Figure 2.1 also shows how the pointers in UPC can be classified based on the locations of the pointers and of the objects they point to. A local pointer such as `loc` may only reference local data (i.e., the pointer and the pointed object belong to the same thread), but the data may live in either the private or the shared space of the thread. A global pointer `glob` may only used to reference data in the shared address space, which may be either local (a shared local access) or remote (a shared remote access). A global pointer is therefore also named a pointer-to-shared. Dereferencing local shared data with a global pointer is slower than using a local pointer due to the extra overhead in determining affinity, and shared remote accesses in turn are typically significantly slower because of the network overhead. Communication in UPC can be either implicit through pointer dereferences and shared array accesses, or explicit through the **upc_memget**, **upc_memput**, and **upc_memcpy** calls.

UPC gives the user direct control over data placement through shared memory allocation and distributed arrays. When declaring a shared array, programmers 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. Revisiting Figure 2.1, the `y` array is declared with a block size of two, which means that the compiler should allocate the first two elements of `y` 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 like the array `x`), while a layout of `[]` or `[0]` indicates indefinite block size, i.e., that the entire array should be allocated on a single thread. As is the case with C pointers and arrays, a point-to-shared can be used to access elements in a shared array.

Figure 2.2 presents a simple parallel vector addition benchmark in UPC. The three shared arrays are distributed cyclically, and each thread follows the owner-computes rule

```
/* vadd.c */
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];
int main() {
    int i;
    for(i=0; i<N; i++)
      if (MYTHREAD == i%THREADS)
          sum[i]=v1[i]+v2[i];
}
```

Figure 2.2: Vector addition benchmark in UPC

and processes its own local shared data. Other notable features of UPC language include dynamic allocation functions, synchronization constructs, and a builtin **upc_forall** parallel loop.

2.2 The Berkeley UPC Compiler

Figure 2.3 shows the overall structure of the Berkeley UPC compiler, which is divided into three components: the UPC-to-C translator, the UPC runtime system, and the GASNet communication system [16]. 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 runtime library calls. The translated C code is next 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.

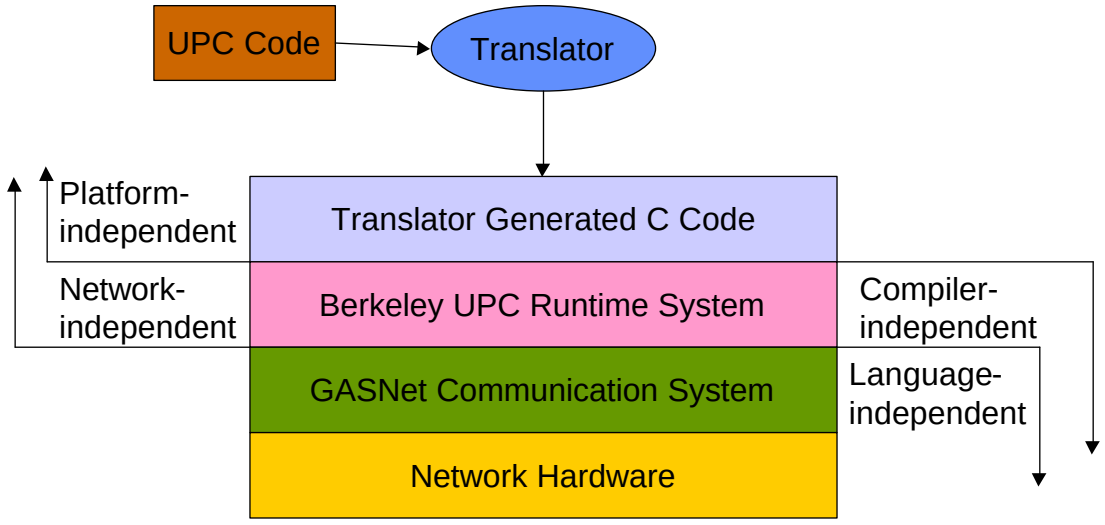


Figure 2.3: The Berkeley UPC compiler

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. GASNet also provides high level non-contiguous remote access methods called the VIS (vector/indexed/strided) functions [17]. The VIS calls accept a list of non-contiguous put/get as arguments, and the communication algorithm is selected at runtime based on network characteristics and transfer parameters. The VIS calls perform message aggregation using GASNet Active Messages, packing non-contiguous data at the source into large packets and unpacking it at the destination.

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 ISO-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 inter-

mediate C output, and the UPC-to-C translator can utilize its UPC-specific knowledge to perform communication optimizations. High performance is achieved in the GASNet system by directly targeting the low-level communication interfaces, while still abstracting away the network specific features from the UPC runtime. Finally, most runtime and GASNet operations are implemented using macros or inline functions to minimize the overhead introduced by the layered design.

2.3 Translation Framework Overview

The analyses and optimizations in this dissertation are primarily implemented in the Berkeley UPC translator [27]. The translator is derived from the Open64 compiler suite [84], an open source collection of optimizing compiler tools. Major components in Open64 include front ends for C/C++/FORTRAN, a loop-nest optimizer (LNO), a global scalar optimizer (WOPT), and an interprocedural analysis framework. Figure 2.4 describes the UPC-to-C translation process applied by the translator. The basic translation process consists of front end processing, backend lowering, and whirl2c transformation:

- 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 [107]) file. UPC-specific information such as shared types and block size for distributed arrays is preserved in the symbol table, so that the later translator phases can utilize the information in performing optimization and code generation.
- Back end: The primary functionality of the backend is to convert expressions involving a pointer-to-shared into the appropriate runtime library calls. Specifically,

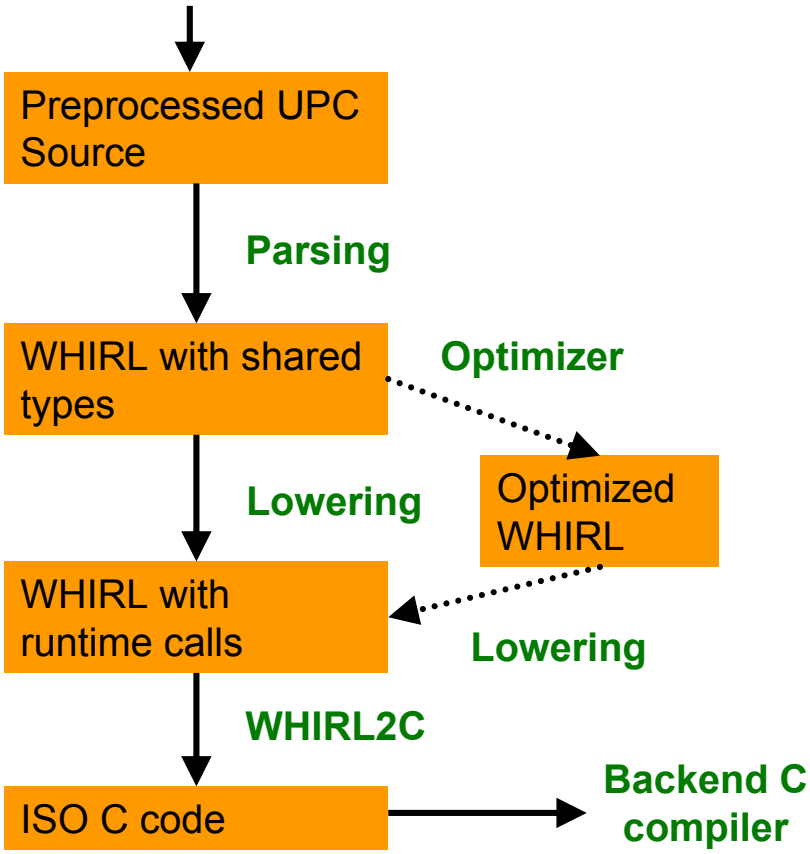


Figure 2.4: UPC-to-C translation process

pointer arithmetic on a shared address is converted into function calls based on the block size 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.

- Whirl2c: The final component's job is to convert the WHIRL intermediate 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 en-

ables us to experiment with different pointer-to-shared representations in the runtime system without having to modify the translator. As Section 4.1 shows, whirl2c generates high-level C language constructs when possible (e.g., using struct field accesses rather than 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, whose storage can not be allocated until runtime to ensure that they are addressable by the network. These variables are instead dynamically allocated and initialized during program startup [41]. 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 [42]; whirl2c is responsible for generating the address translation macros when accessing such variables in the program.

When compiling for clusters, conceptually the translator targets the following 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. In particular, messages are not guaranteed to be delivered in order; if two puts are made to overlapping memory locations without a sync in between, the resulting value is undefined.

When the "–opt" flag is passed to the Berkeley UPC compiler, the translator invokes the optimization phase, which includes both the LNO and the WOPT, before the backend lowering of shared expressions. The goal of LNO is to improve the memory performance of a program's loop nests; it includes an extensive data dependence analysis framework and supports well-known loop transformations such as loop fission/fusion, unroll and jam, loop tiling, and vector data prefetching [108]. WOPT operates on individual functions and performs a number of standard optimizations such as copy propagation, partial redundancy elimination, and dead code elimination. At the heart of WOPT is its Hashed Static Single Assignment (HSSA) representation [30], which extends SSA to support pointer aliases and indirect memory operations. The HSSA form is used to implement most of the optimizations, including a partial redundancy elimination framework (SSAPRE) [29, 66].

For our source-to-source transformation, several of Open64's optimizations are not directly applicable, since they produce outputs that are too low-level to be expressed in C (e.g., register promotion [75] and automatic parallelization). Such optimizations are therefore disabled by the Berkeley UPC translator, with the hope that they can be performed equally well by the backend compiler. In addition to employing many of Open64's large repertoire of optimizations in the compilation process, we have also supplied several optimizations that require UPC specific knowledge and could not be performed by a C compiler.

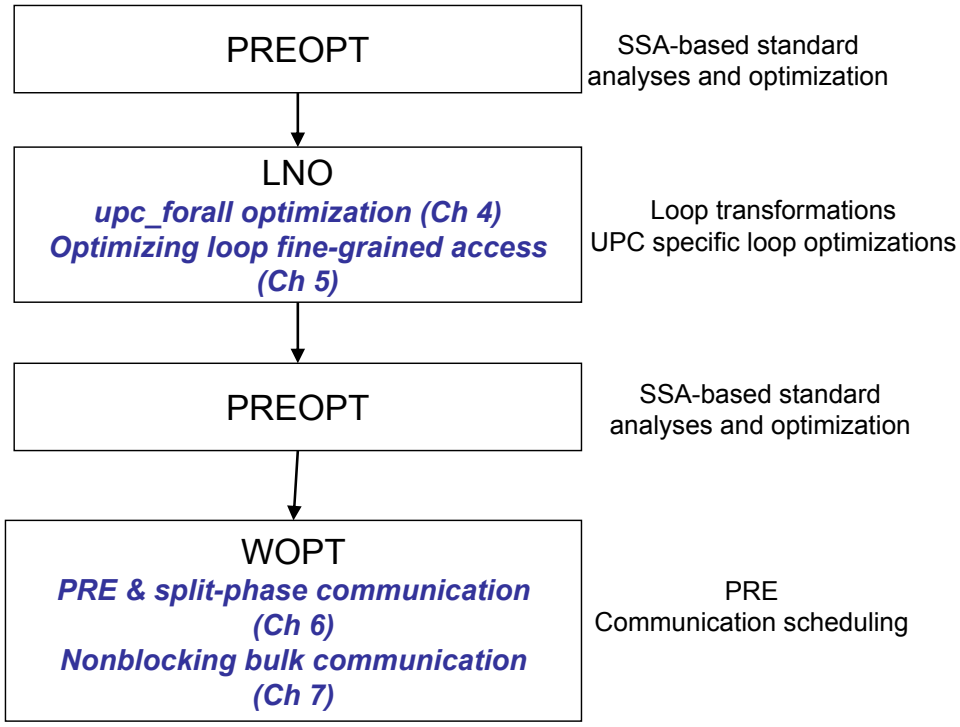


Figure 2.5: Optimization framework for the translator.

Figure 2.5 summarizes the overall structure of the Berkeley UPC translator's optimization framework. At the end of the LNO phase, we add an optimization for the **upc_forall** parallel loop that helps remove the overhead of distributing the iterations to the executing threads (Section 4.3). Also in the LNO is a loop communication optimization framework, described in Chapter 5, that hoists fine-grained remote accesses out of a loop nest and combines and schedules them to significantly reduce communication overhead. Both optimizations utilize LNO's analysis information, and operate on loops that have the semantics of Fortran DO Loops. Specifically, a DO loop contains a single index variable, and the

condition expression is a comparison on the value of the index variable; the lower bound, upper bound, and stride of the loop are all loop-invariant.

A separate optimization at the end of the WOPT phase is designed for fine-grained accesses that do not benefit from our loop optimization framework (e.g., they do not appear inside loops, or have dynamic access patterns). The optimizer performs PRE on both shared pointer arithmetic and shared memory accesses, applies split-phase communication to separate the initiation of an access from its completion, and coalesces individual accesses when appropriate. For the bulk communication routines that are not amenable to static analysis, Chapter 7 presents a runtime-based optimization framework that automatically converts them into nonblocking transfers. The candidates for this optimization (**upc_memget** and **upc_memput** calls) are also identified at the end of WOPT.

2.4 Memory Consistency Models

All of the optimizations presented in this dissertation require the ability to reorder remote accesses in a program, either through aggregation or nonblocking communication. In a uniprocessor environment, such compiler transformations must adhere to a simple data dependency constraint: the orders of all pairs of conflicting accesses (accesses to the same memory location, with at least one a write) must be preserved. The execution model for parallel programs is considerably more complicated, since each thread executes its own portion of the program asynchronously, and there is no predetermined ordering among accesses issued by different threads to shared memory locations. A memory consistency model defines the memory semantics and restricts the possible execution orders of mem-

ory operations. More precisely, it determines whether accesses from one thread may be observed to be out of order by another thread, and whether such reordering is legal.

An 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 "strict" or "relaxed" either explicitly or inferred from pragmas. The relaxed model permits reordering of memory accesses as long as local data dependency is preserved, while the strict model further requires that the effects of reordering can not be observed by other threads. 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 [81]. For example, reordering on either thread in Figure 2.6 is forbidden if either variable x or y is declared strict, since doing so may produce results that would not happen if execution follows the original program order. If both variables are declared relaxed, however, the compiler can freely reorder the memory operations due to the absence of local data dependencies.

The goal of the UPC memory model is to exploit the tradeoff between programmability and performance; relaxed accesses offer better performance since 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. While race conditions are likely to be programming errors even in a strict memory model, debugging them in a relaxed model is more difficult due to potential reordering

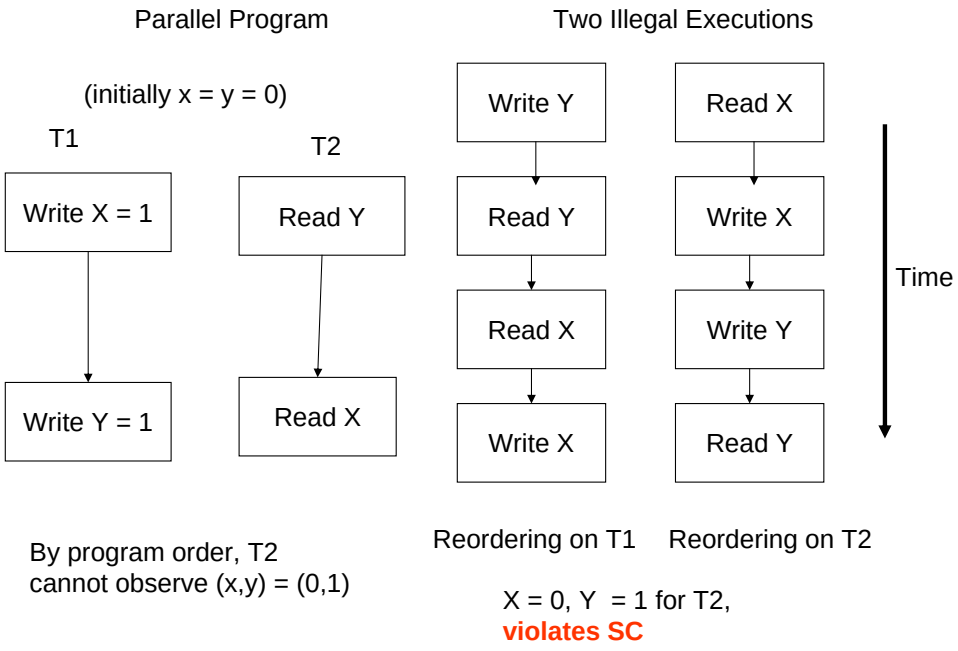


Figure 2.6: Example of reordering that violates the strict memory model.

performed by the compiler. Optimizations described in this dissertation take advantage of UPC's relaxed model to aggressively reorder and combine shared accesses. Synchronization operations such as barriers and lock/unlock operations are treated as code motion barriers. In a static setting, they are represented as black box function calls that could modify every shared memory location, and in a dynamic setting we simply stop the optimization when encountering a synchronization event. Strict accesses are also modeled as synchronization operations to prevent illegal reordering caused by our optimizations. This is conservative, and more sophisticated parallel analysis techniques could be applied to

enable high-level optimizations for strict accesses [26]. We have found our strategy to be sufficient for UPC programs in practice, however, since they rarely contain strict accesses other than the built-in synchronization primitives.

Chapter 3

Experimental Setup

Portability is an important goal for the Berkeley UPC compiler; since the compiler translates UPC program into C code, it runs on any networking architectures that GASNet communication interface supports, which includes most of today's popular interconnects[1]. Our optimizations should thus be evaluated on different cluster platforms to ensure that they are performance portable. Performance results in this dissertation are collected on the supercomputer clusters listed in Table 3.1, covering a variety of network interconnects and processor architectures. The *Get* and *Put* times in Table 3.1 refer to the cost of executing an eight byte blocking remote access in our runtime, i.e., the roundtrip latency.

We use a number of benchmarks to evaluate the effectiveness of our optimization framework. Most of the benchmarks are written by researchers outside our group, and reflect the kind of UPC programs that our compiler is likely to encounter in the real world. Since each phase of our optimization concentrates on different communication patterns, we also

[1]The full list of supported platforms may be found at http://upc.lbl.gov/download/.

System	Processor	Network	Software	Get(us)	Put(us)
POWER5/LAPI (Bassi)	8-way 1.9GHz POWER5 (111 nodes)	IBM Federation	AIX 5.3, IBM LAPI 2.3.3.3, IBM xlc v7.0	9.2	8.2
Itanium/GM (Citris)	Dual 1.3GHz Itanium2 (42 nodes)	Myricom Myrinet LANai XP PCI-X	Linux 2.6.18, icc v9.0	23.3	19.7
Opteron/VAPI (Jacquard)	Dual 2.2GHz Opteron (320 nodes)	Mellanox Cougar InfiniBand 4X	Linux 2.6.5, Pathscale cc 2.4	11.6	8.4

Table 3.1: Machine summary

use different benchmarks to evaluate them. For the framework in Chapter 6, we use benchmarks containing fine-grained irregular accesses that can not be readily vectorized with our loop optimization. Examples include distributed hashtables and dynamic work stealing algorithms. For the automatic nonblocking communication optimization in Chapter 7, we choose programs that use bulk memory copies for communication.

Table 3.2 summarizes the benchmarks used in this dissertation. The Get/Put column refers to the dominant communication type in the program, and LOC represents the number of lines of UPC source code (including comments) for each benchmark. The Get/Put count column refers to the dynamic count of the remote accesses, while the last column reports the average size of each access. The first five benchmarks are communication intensive application kernels with irregular fine-grained accesses.

Gups: Gups is a benchmark that performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction. Communication in the benchmark takes the form $a[b[i]]$.

Mcop: The Mcop benchmark solves the matrix chain ordering problem using clas-

Name	Get or Put	LOC(1000s)	Get/Put Count(1000s)	Avg. Size(KB)
Gups	both	0.3	490	0.01
Mcop	get	0.4	178	0.004
Sobel	get	0.6	153	0.001
Psearch	get	0.8	2.1	0.05
Barnes Hut	get	2.2	223	0.004
BT	both	6.7	2100	0.15
CG	get	1.5	34	33
FT	put	2.5	5	131
FT-pencils	put	2.5	164	4
IS	get	1.1	0.352	239
MG	put	2	19	2.6
SP	put	6.1	636	1.7
Gups-bulk	put	0.6	8	0.5
CFD	get	14.1	123	0.1

Table 3.2: Benchmark summary. Results for the last two columns were collected on 16-thread runs, using data from thread zero.

sic dynamic programming algorithms [35]. The shared matrix is distributed cyclically by columns, and communication occurs as part of solving the recurrence relations over all subproblems.

Sobel: The Sobel benchmark, whose implementation is described in [43], performs edge detection with Sobel operators (3X3 filters). The image is divided into equal contiguous slices of rows and distributed among the threads, so that communication occurs only when processing the last row of data.

Psearch: The Psearch benchmark performs parallel unbalanced tree search [89]. The benchmark is designed to be used as an evaluation tool for dynamic load balancing strategies. Communication takes place when a thread attempts to steal work from another thread.

Barnes Hut: The Barnes Hut benchmark [109], ported into UPC from the SPLASH2 benchmark suite, simulates the interaction of a system of bodies (e.g. galaxies or particles)

in three dimensions.

The rest of the benchmarks use bulk memory copies for communication and thus have relatively large message size. Most of them come from the NAS Parallel Benchmarks suite [7], and a brief description is given below.

BT: The NAS BT (Block Tri-diagonal) benchmark simulates a CFD application by solving a system of equations for a three dimensional array of points. In this benchmark, a thread issues a large number of strided remote accesses to its neighbor threads.

CG: The NAS CG (Conjugate Gradient) kernel is a sparse iterative solver in which a sparse matrix-vector multiplication dominates the execution of the benchmark. The benchmark's implementation is described in [14].

FT: The NAS FT kernel solves a partial differential equation on a 3-D mesh using the Fast Fourier Transforms (FFT). The hand-tuned UPC implementation aggressively overlaps communication with computation, by decomposing the FFT computation and communication into smaller pieces instead of performing a global exchange [12]. **FT-pencils** is a variant of the benchmark that further reduces the granularity of the overlap.

IS: The NAS IS (Integer Sort) kernel performs a parallel bucket sort on integer keys. Communication is implemented as a global exchange.

MG: The NAS MG (Multigrid) kernel uses the Multigrid method on hierarchical regular 3-D meshes. At each level of the grid, communication occurs due to the periodic updates of ghost cell regions.

SP: The NAS SP (Scalar Penta-diagonal) benchmark shares a similar structure to the BT benchmark. The benchmark performs a large number of strided remote puts.

Gups-bulk: Gups-bulk is a version of the HPCS RandomAccess benchmark that uses bulk communication [77]. Unlike the fine-grained Gups, in this benchmark a thread "looks ahead" the random address stream and pushes them to the remote threads to perform the updates.

CFD: CFD is a computational fluid dynamics application that solves the time dependent Euler equations for computational fluid flow in a rectangular computational domain, with the high level data structures and algorithms implemented in UPC.

The UPC versions of BT, IS, MG, and SP are derived from the MPI NAS parallel benchmarks, and their implementations are described in [33, 43].

Chapter 4

Single-Node Performance

Most optimizations for PGAS programs attempt to reduce their communication time in order to combat the high network latencies on clusters. An equally important yet sometimes overlooked performance metric for parallel programs is the uniprocessor execution time, as most scalable applications will spend substantially more time doing computation instead of communication. Since UPC is designed as a parallel extension of C, serial performance of UPC programs can be further divided into two components: performance on ordinary C code, and performance on UPC-specific features such as access to shared local data and shared pointer arithmetic. For the former, it is important to ensure that the source-to-source translation strategy adopted by our compiler does not cause a performance slowdown by generating less efficient code. In Section 4.1, we examine the code generation quality of our translator in more detail on a number of architectures. The subsequent sections focus on the serial performance of the unique language features in UPC. In Section 4.2, we describe techniques for lowering the overhead of the UPC shared pointer manipulation functions.

In Section 4.3, we present an optimization framework that could eliminate most of the sequential overhead introduced by the **upc_forall** parallel loop.

4.1 Standard C Code Performance

Since the Berkeley UPC compiler adopts a source-to-source translation strategy, serial performance for UPC programs is largely dependent on the quality of the C compiler used to compile the generated C code. On most supercomputers, the vendor supplied compiler typically delivers much better serial performance than gcc, and our translator is carefully implemented so that the generated code can be compiled by a wide variety of compiler and hardware combinations.

While the quality of the backend C compiler is beyond our control, it is still important to ensure that the source-to-source translation does not cause a performance slowdown due to semantically equivalent but less efficient code being produced. Due to the large number of systems supported, our translator does not apply platform-specific tunings in its code generation. Instead, it aims to keep the translated output as syntactically similar as possible to the original source, to minimize the potential performance perturbation introduced by the translation. For example, control structures such as loops and switch statements are preserved in their original form. Due to optimizations performed by the translator and the lack of a one-to-one mapping between its intermediate representation and the C language, it is generally impossible for the translated output to be syntactically identical to the program source. In these cases, our translator produces high-level C code, in the hope that they will most closely resemble the original code in both appearance and performance. For

example, pointer arithmetic expressions that are represented as byte offsets internally are converted into symbolic field and array accesses whenever possible, so that the generated code is human-readable and likely similar to the original user code. Multi-dimensional array accesses are reconstructed instead of being flattened into one-dimensional array accesses. The translator also reduces the number of unnecessary casts in its output, to avoid confusing the backend C compiler.

Restrict-qualified pointers and pragma directives are the two most common optimization hints for C code, and they are fully supported by the Berkeley UPC compiler. Our translator utilizes information from restrict pointers to make its alias analysis more accurate, and passes the restrict qualifier in the translated code so that the backend compiler could also benefit from them. All Berkeley UPC specific pragmas share the prefix "#pragma bupc", and any pragma directives not matching this form appear unchanged in the same relative location in the generated C code.

We use four of the NAS parallel benchmarks (CG, FT, IS, MG) from Table 3.2 to evaluate the potential overhead of our source-to-source translation. From the parallel UPC code a purely sequential C version is generated by stripping away the UPC features. All references to the *shared* qualifier and the block size are removed, and the THREADS variable is defined to one while the MYTHREAD variable is defined to zero. Synchronization calls such as **upc_barrier** are removed, and memory allocation as well as string copy functions are converted into their C equivalents (e.g., **upc_alloc** becomes **malloc**, and **upc_memget** becomes **memcpy**). Similarly, a **upc_forall** loop is directly translated into a for loop.

The PathScale compiler v2.4 [87] is used for the Opteron cluster, Intel compiler v9.0 [57] for the Itanium cluster, and IBM XL C v7.0 [110] for the POWER5 cluster. The standard

`-O3` flag is used on all three platforms.

For reference, we also include the NPB3.2 Fortran/MPI implementation of the benchmarks, compiled with the Fortran compiler from the above compiler suites using the -O3 flag. Compared to C, Fortran employs much stricter aliasing rules (e.g., arguments to function calls may not be aliased), which makes it more amenable to compiler optimizations. To allow for a fair performance comparison between the two languages, we judiciously apply the restrict qualifiers in both the C and UPC version to assist the compiler's pointer analysis. Since the uniprocessor execution time of the four benchmarks, like many scientific applications, is dominated by a few computation loops, software pipelining [68] is very important to the overall performance. As dependence analysis for C is generally less precise than that for Fortran, we also assist the optimizer by supplying the `ivdep` pragma to loops so that they can be effectively pipelined.

Figure 4.1 presents the results for our experimental configurations. The figure reports normalized performance, by dividing the MFLOP rate of the C/Fortran run over the MFLOP rate of the UPC run on a given platform; a value greater than one thus indicates performance superior to UPC. There are no Fortran results for IS because the benchmark is implemented in C in the standard NAS benchmark release. As the graph shows, the uniprocessor performance of C and UPC is very close, with the C code outperforming the UPC version by approximately 2.5% on average. Since the UPC code contains additional parallel constructs that may incur superfluous overhead in serial execution, this suggests that our compilation framework suffers very little performance loss from the source-to-source translation. The Fortran code outperforms the C and UPC versions in general, owing to the fact that Fortran is designed to be more amenable to high degrees of optimization, espe-

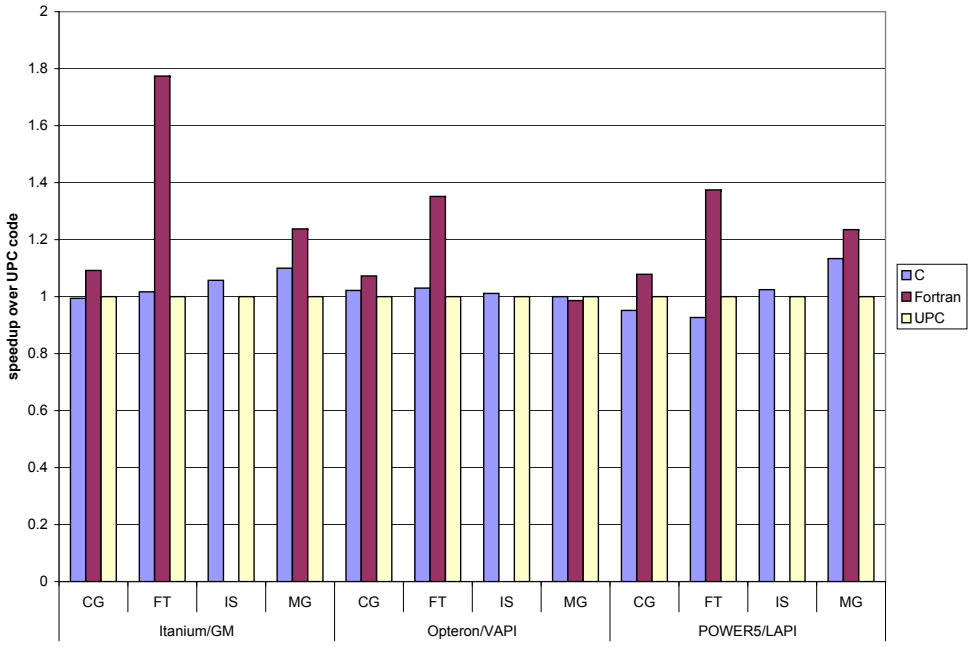


Figure 4.1: Serial performance comparison: UPC v. C v. Fortran. Class A input is used.

cially for scientific code. The largest performance gap occurs on the FT benchmark, where the Fortran version outperforms the UPC code by an average of about 50%. This large gap can be attributed to the different implementation of complex arithmetic. In the Fortran code, complex numbers are implemented using the builtin complex type, with addition and multiplication of two complex numbers directly expressed using the language-supplied operators. In the C version on the other hand, the complex number is implemented as a struct of two doubles, and complex arithmetic operates on the two fields explicitly. Even though the two versions perform semantically equivalent operations on the complex numbers, the

compilers we have tested are much more effective at optimizing the Fortran code.

4.2 Performance of Pointer-to-shared Operations

A pointer-to-shared in UPC needs three logical fields to fully represent the address of a shared object: `thread_id`, `address`, and `phase`. The `thread_id` indicates the thread that the object has affinity to, the `address` field stores the object's *local* address on the thread, while the `phase` field gives the offset of the object within its block. Figure 4.2 demonstrates how the fields in a pointer-to-shared are used to access a shared value.

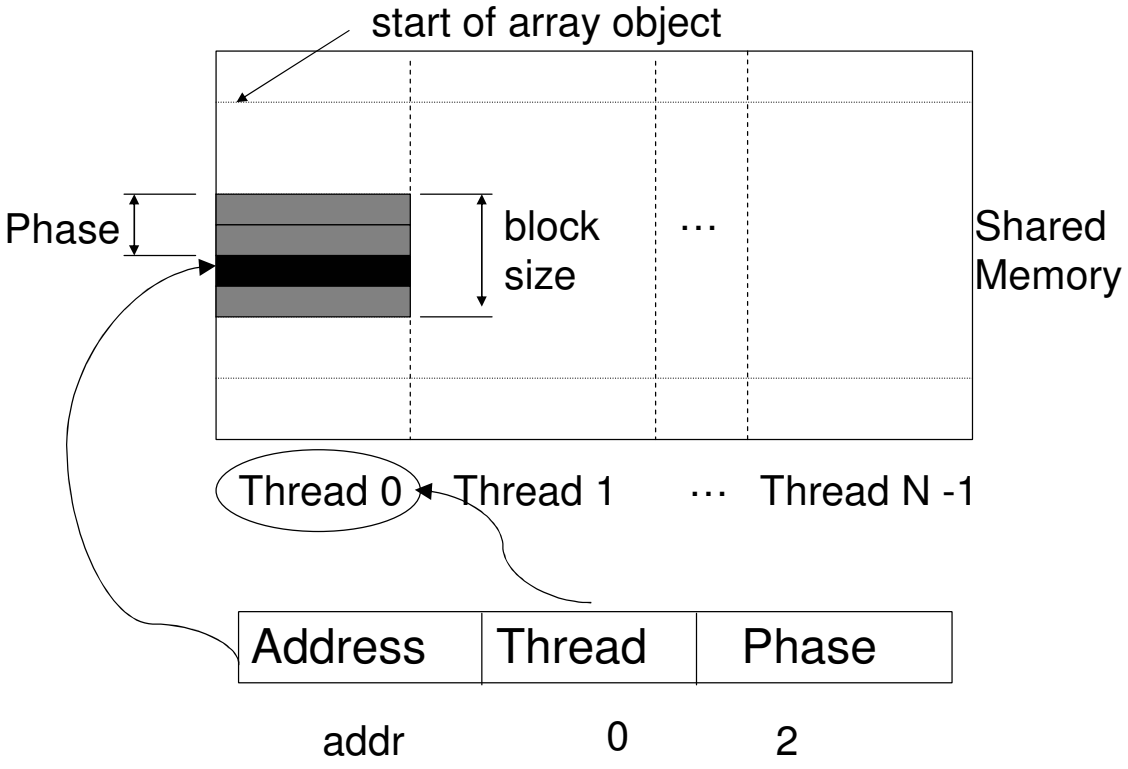


Figure 4.2: UPC pointer-to-shared components.

When dereferencing a pointer-to-shared, the compiler needs to first determine whether the shared object is local or remote. A local shared access is thus generally slower than accessing it through a local pointer, due to the extra branch involved in determining an object's locality. Address arithmetic on pointers-to-shared will also be inevitably slower compared to their counterparts on local pointers, since a pointer-to-shared contains three fields, all of which may be updated during pointer manipulations. Experienced programmers typically avoid this overhead by casting the pointer-to-shared into a local pointer first before accessing it. From a productivity standpoint, however, it is important to keep the overhead of these pointer-to-shared operations low, so that programmers would not be forced to keep two pointers (one local and one global) for the same data. In Chapter 6, we will show that such overhead can be amortized through compiler partial redundancy elimination; here we present runtime techniques that reduce the individual overhead of these operations.

By default the Berkeley UPC compiler uses a packed eight byte integer to represent a pointer-to-shared; this allows pointer-to-shared operations to be more efficiently compiled since the pointer-to-shared format will fit in the registers of most modern high-performance computing platforms. Only when an application needs either a large amount of shared memory ($>$ 4GB per thread) or more than a few thousand threads does a programmer need to switch to a struct-based representation.

Another important runtime optimization for pointer-to-shared is the "phaseless pointer" representation. A UPC pointer-to-shared can be classified into three categories based on its declared block size: *block cyclic* for block size > 1, *cyclic* for block size $= 1$, and *indefinite* for block size $= 0$. For the frequently used cyclic pointers, which have block size one, the

`phase` field can be eliminated since its value is always zero. Similarly, indefinite pointers can omit their phase since all elements reside in a single block. Cyclic and indefinite pointers are therefore named *phaseless*, and our compiler exploits this knowledge to enable more efficient operations for them. For cyclic pointers, shared pointer arithmetic can be implemented with a modulo (for updating the thread id) and a divide (for updating the address) operation, while for indefinite pointers it can be implemented directly as a regular pointer addition, as the thread id never changes.

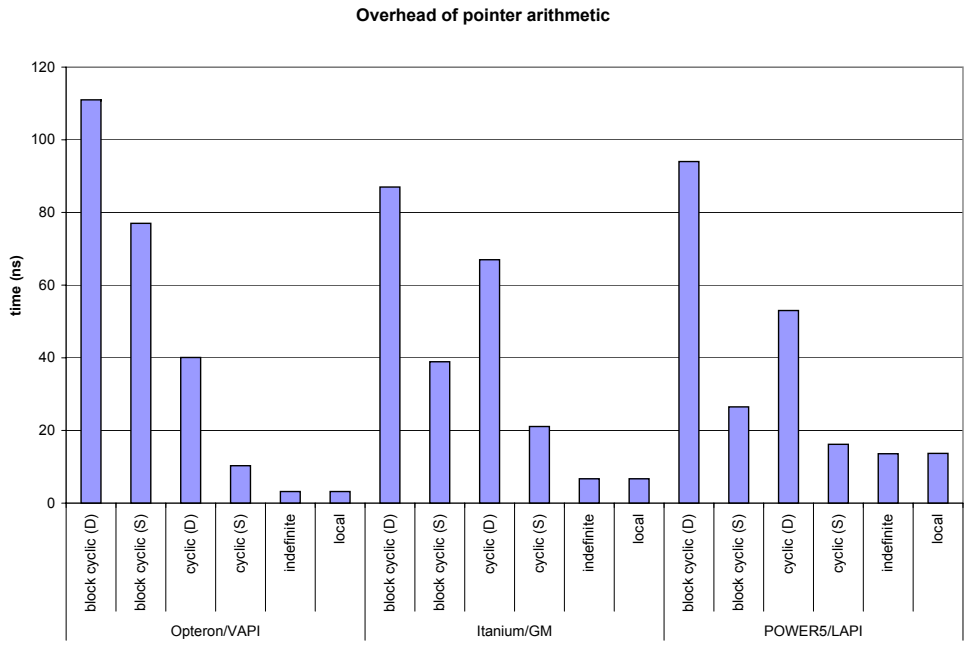


Figure 4.3: Performance of UPC pointer arithmetic. (D) denotes dynamic threads, (S) static threads.

Figure 4.3 presents UPC shared pointer arithmetic performance (ptr + int) on three

platforms from Table 3.1, using the same compilation environment as described earlier in Section 4.1. Two threads were used in the experiments. The variables in the benchmark are declared to be volatile so that the C compiler will not attempt to optimize away the operations. Due to the phaseless pointer representation, pointer arithmetic for cyclic pointers is significantly faster than that for the generic block cyclic pointers on all platforms. Furthermore, pointer addition for indefinite pointers is as fast as regular C pointer arithmetic since they use the same sequence of instructions. For the block cyclic and cyclic pointer we observe a significant performance improvement moving from the *dynamic THREADS* environment to the *static THREADS* environment. The reason is that both types of pointers perform modular arithmetic, which can be more efficiently optimized by the C compiler if the number of threads is a compile time constant (e.g., strength reduced into shift operators when THREADS is a power-of-two). Thus, for programs that perform a non-trivial amount of (block) cyclic pointer-to-shared arithmetic, the number of threads should be specified at compile time. Indefinite pointers, however, are not affected by the dynamic versus static thread setting.

Figure 4.4 presents the cost of a UPC shared local access (*p) to an eight-byte double. It is clear from the figure that accessing local data through a pointer-to-shared incurs substantial overhead due to the extra branch needed to determine the pointer's affinity. The optimization framework described in Chapter 6 can help lower this performance penalty by eliminating some of the redundant locality checks, and an analysis that statically determines which accesses are guaranteed to be local can also be very profitable. In earlier work, we have also compared the performance of shared local accesses between the Berkeley UPC compiler and the HP UPC compiler, and found that the Berkeley compiler is slightly slower due to the source-to-source translation [25].

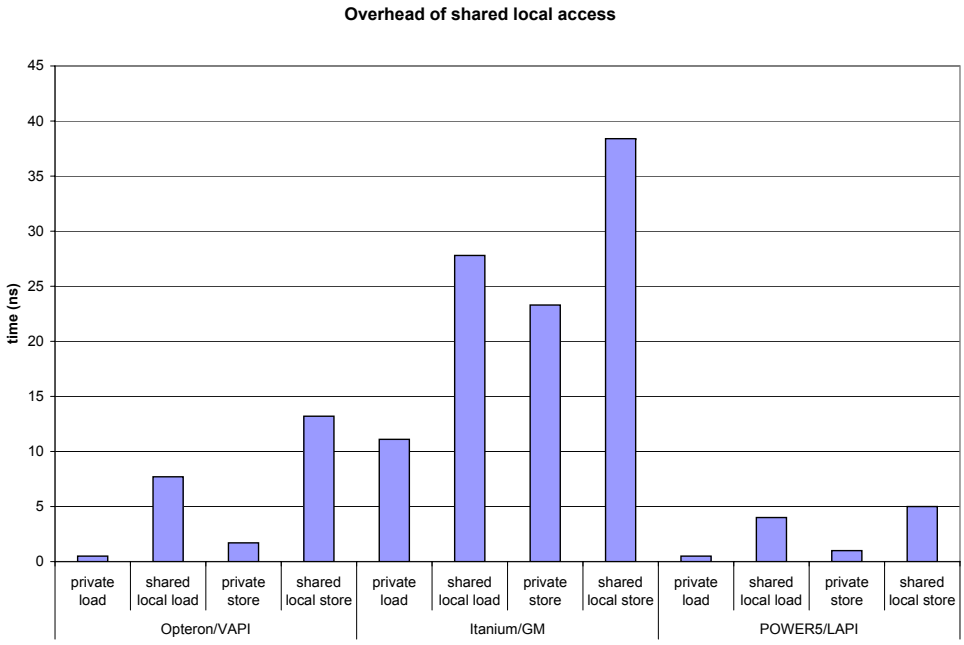


Figure 4.4: Performance for UPC shared local access

4.3 Optimizing UPC Forall 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 of pointer-to-shared type, and the affinity

test checks if the running thread has affinity to this address (i.e., upc_threadof(aff exp) is equal to MYTHREAD.). The affinity expression can also be the `continue` keyword or simply 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. Figure 4.5 b gives the semantic definition of the forall loop in part a, where the loop body is executed only if the affinity test succeeds. The forall loop is a collective operation, which means that all threads will execute the loop. Its controlling and affinity expressions must also be *single valued* [1], meaning that all threads agree on which threads execute each iteration.

UPC forall loops provide a convenient syntactic sugar for thread coordination and the prevention of inadvertent remote accesses, but its primary drawback is the runtime overhead incurred by executing the affinity tests in all loop iterations. In a straightforward translation shown in part b of Figure 4.5, the affinity tests have to be executed on each iteration by all threads, and the presence of the branches in the loop can also inhibit many useful loop optimizations. The mod operation in the figure performs modular arithmetic. Fortunately, while their values naturally change 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.

4.3.1 Affinity Test Removal

The goal of the optimization is to transform the forall loop into an equivalent for loop with the affinity test eliminated. Our optimization operates on forall loops, and by definition their lower bound (L), upper bound (U), and step (S) must all be loop invariant. We first

```
upc_forall (i = L; i < U; i+=S; A*i + B)
  loop body;
```

a) Original upc_forall loop

```
for (i = L; i < U; i+=S)
  if (((A*i + B) mod THREADS) == MYTHREAD) {
    loop body;
  }
```

b) Straightforward translation

```
int P = THREADS / gcd(S*A, THREADS);
int start_th = (A*L + B) mod THREADS;
int my_start = forall_start(start_th, S, A, L);
for (i = my_start; i < U; i += S * P)
  loop body;
```

c) Optimized translation

Figure 4.5: upc_forall loop affinity test removal.

consider the case when the affinity is an integer expression linear to the loop induction variable; in other words, it is of the form $Ai + B$, where i is the induction variable and both A and B are loop invariant constants. For notational convenience, let $K = AL + B$, or the thread that executes the very first iteration of the forall loop, and let $T = THREADS$. The sequence of the threads executing the iterations of the forall loop is then $\{K, K + S * A, K + 2 * S * A, ...\} \pmod T$, since the affinity expression is incremented by $S * A$ in each iteration. Following the rules of modular arithmetic, the sequence will repeat every $P = T/\gcd(T, S * A)$ iterations, where gcd is the greatest common divisor function.

Furthermore, a thread M will be in the sequence if and only if $(M - K) \mid \gcd(T, S * A)$. Figure 4.6 illustrates how the formula can help discover the distribution of the forall loop iterations among the threads. In a), $\gcd(T, S * A)$ is 1, so each thread executes every T iteration of the forall loop. In b), the gcd is no longer 1, and thus only thread 3, 5, and 1 participate in the forall loop, with the sequence repeating every three iterations.

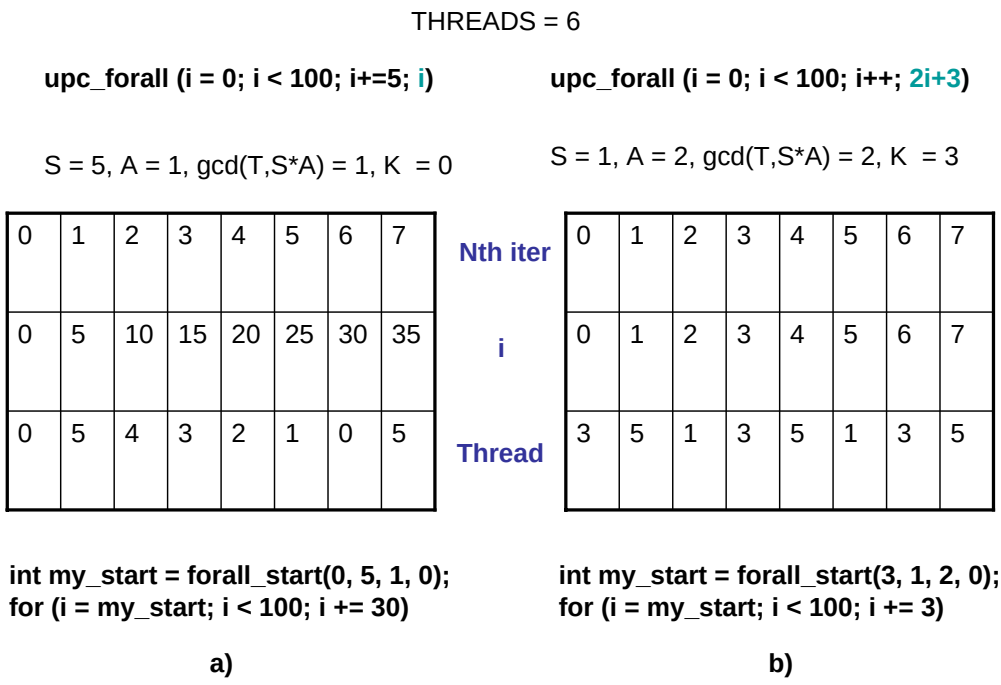


Figure 4.6: Examples of forall loop iteration to thread mapping. The original forall loop appears at the top, while the result C code is at bottom.

Once the mapping of thread id to iteration is statically determined, the affinity test can be eliminated as shown in part c) of Figure 4.5. In the figure, `start_th` stores the id of the thread executing the first iteration, or the K variable in the previous paragraph. The

function `forall_start` computes for a thread the value of i at the first iteration that it will execute. This can be computed by scanning through the sequence $K + S * A$ until we find an element equal to $MYTHREAD$. For threads that do not execute the forall loop at all, the function returns a value larger than U, so that the thread will never enter the transformed loop. The step of the new loop is changed to $S * P$, to reflect the fact a thread is executing the body of the original forall loop every P iterations.

When the affinity expression is a shared address, our optimization first examines the blocking factor of the pointer-to-shared. If the pointer is indefinite, the affinity test branch can be trivially hoisted out of the forall loop, since the affinity expression will always fall on the same thread. If the pointer is cyclic and the index expression is affine (i.e., $ai + k$), the affinity expression is equivalent to the integer expression $ai + (k + upc\_threadof(p))$, where the **upc_threadof** library call returns the thread that has affinity to the shared object pointed to by p. We can therefore apply the same transformation described in the previous paragraph. Affinity expressions involving block cyclic arrays are currently not supported by our optimization due to their rare occurrence, but the framework could be extended to include techniques used by the HPF compilers for block cyclic array distributions [53, 65].

4.3.2 Privatizing Shared Local Accesses in Forall Loops

Another important optimization for **upc_forall** loops is to use the information provided by the affinity expression to privatize the shared local accesses in the loop, by using a local pointer to perform the access. If the affinity expression is a pointer-to-shared ($\&p[exp]$), by definition it must point to a local object inside the loop. Thus, if the expression is dereferenced, the compiler can directly cast it into a private pointer and save a runtime

branch. Furthermore, any shared access $q[exp1]$ inside the forall loop is also local if the compiler can determine at the program point of the access that 1) $exp1 == exp$ and 2) the pointers-to-shared p and q have the same layout, i.e., they have the same block size and **upc_threadof(p) == upc_threadof(q)**). An integer affinity expression e is semantically equivalent to the pointer-to-shared expression $\&a[e]$, where a is a cyclic array, and therefore also satisfies the two properties.

Based on these observations, we develop an algorithm that checks every shared access in a forall loop to see if it satisfies the two properties. The algorithm currently operates on cyclic pointer-to-shared expressions, though it could be extended to support block cyclic pointers. Since the affinity expression cannot be modified inside the loop body, verifying condition 1 is a simple case of pattern matching for structural equivalence; if the two expressions contain the same set of terms and operators[1], they must produce the same value since none of the variables are modified in the loop body. Global value numbering can also be applied to determine if two expressions have the same value.

Condition 2 is trivially true if p and q are arrays, since they both always point to thread zero. If q is a pointer, however, verifying condition 2 becomes more complicated, since it could point to any thread prior to entering the loop. If the compiler could determine that q is never modified inside the loop, it could dynamically check that p and q point to the same thread before entering the loop, and choose the privatized/non-privatized version of the loop based on the branch. This transformation has the undesirable effects of increasing code size, however, and is therefore not enabled by default. An interesting future work would be to apply static analysis like the one in [113] to track the thread that a pointer-to-shared points to at each program point.

[1]The optimizer has simplified expressions into canonical forms at this point.

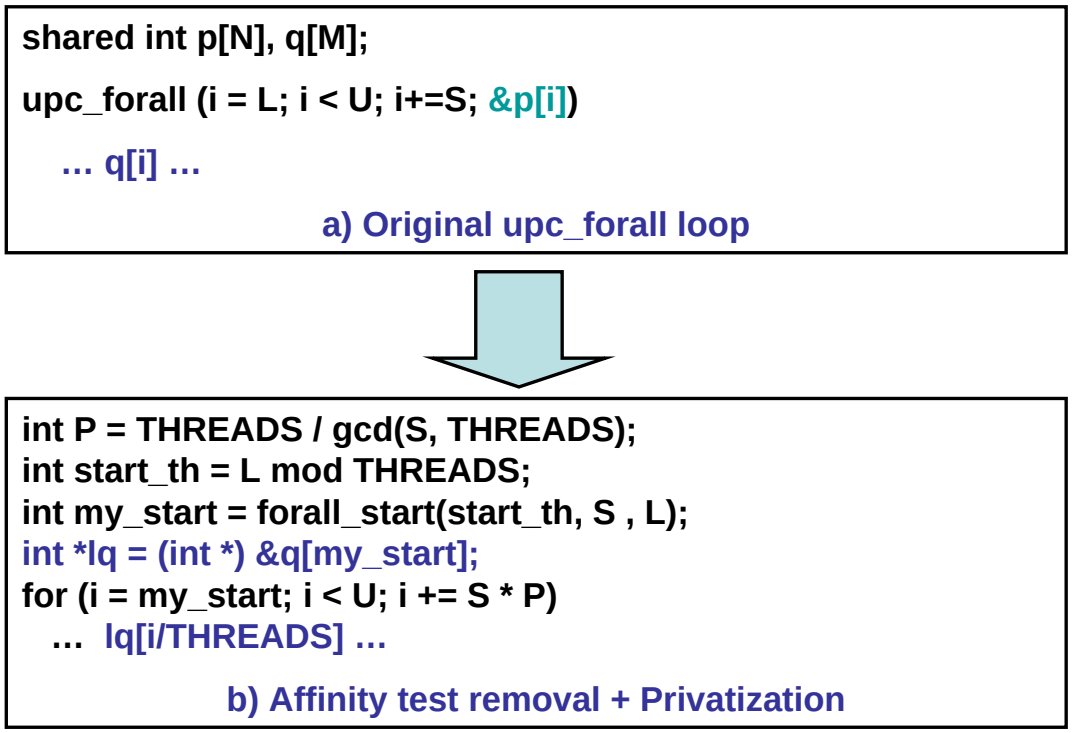


Figure 4.7: Privatization of shared local accesses in forall loops.

Once the optimizer determines the two conditions are satisfied, the pointer-to-shared expression is replaced with an equivalent private pointer as shown in part b) of Figure 4.7. The index expression of the new private pointer also needs to be updated, since the iteration space is now a thread's private portion of the cyclic array.

4.3.3 Experimental Results

A version of the HPC stream triad benchmark [77] is used to evaluate the effectiveness of our forall optimizations. The benchmark scales and adds two vectors, and stores the results into a separate vector ($a[i] = b[i] + \alpha * c[i]$). The implementation declares three cyclic arrays

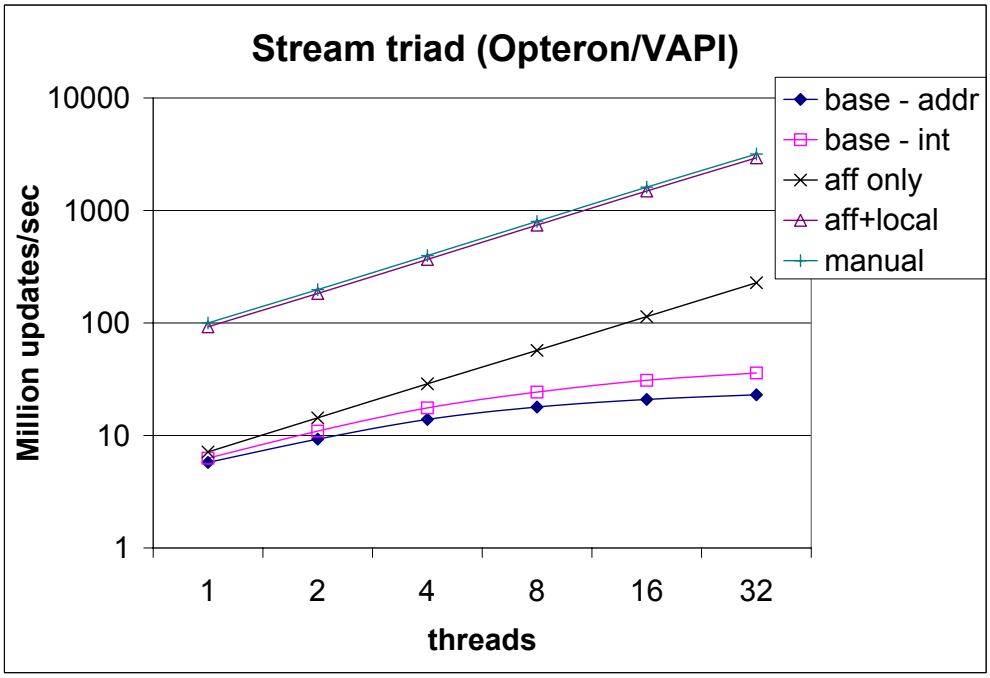


Figure 4.8: Stream triad benchmark on the Opteron/VAPI system.

to represent the vectors, and uses a **upc_forall** loop to ensure that each thread only performs computation on its local elements. Each cyclic array contains one million * THREADS elements. We experiment with both integer and shared address affinity expressions for the loop.

Figure 4.8 presents the results on the Opteron/VAPI cluster in Table 3.1. The results from the other two platforms behave similarly and are not shown. Pathscale C compiler version 2.4 [87] is used as the backend compiler, with the `-O3` option. Five configurations are examined: two *base* versions compiled without optimization, with one using integer affinity (i) and the other a pointer-to-shared ($\&a[i]$); an *aff-only* version that performs only affinity test removal, and an *aff+local* version that also performs privatization; and finally a *manual* version that uses a for loop with private pointers. For the optimized versions we

do not distinguish between integer and pointer-to-shared affinity, since the same code is generated. Comparing the two base versions, the integer affinity outperforms the pointer-to-shared affinity even though they produce the same iteration assignments. The reason is that an integer affinity is converted into a C % operation, while a pointer-to-shared affinity turns into a runtime function call that checks the pointer's thread id. Inlining the function call would help performance in the case when the optimization fails to eliminate the affinity test (e.g., when the affinity is a non-linear expression).

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 base versions scale poorly even with a small number of threads, the optimized output achieves linear speedup. Privatizing the shared local accesses brings an order of magnitude performance improvement, since the loop contains minimal amount of computation and thus spends most of its time on the overhead of shared pointer arithmetic, which is eliminated as part of the privatization. All three vectors in the loop are privatized by our optimization, thus making it as efficient as manual tuning.

Chapter 5

Optimizing Fine-grained Array Accesses

Efficient communication code generation for loop nests is critical to the performance of most array-based parallel programs. This is especially important for a PGAS language like UPC, which supports flexible fine-grained remote accesses for communication. The most well-known optimization, *message vectorization*, hoists individual remote array accesses out of a loop nest by fetching the data using bulk transfers. This transformation, which can be done either manually or automatically by the compiler, speeds up communication by amortizing startup overhead and taking advantage of the higher bandwidths realized by large messages on clusters. Recognized as an extremely useful optimization for message-passing programs, vectorization has been implemented in optimizing compilers for parallel programming languages such as HPF and Fortran D [18, 48, 52].

While message vectorization is an effective optimization, it does not exploit communication overlap. For PGAS programs on modern networks, better performance can be achieved with a finer-grained message decomposition and aggressive pipelining, using a

technique called *message strip-mining* [54, 105]. While a vectorized loop waits for the remote memory access to complete before it proceeds with local computation, message strip-mining divides communication and computation into phases and pipelines their executions by skewing the loop. This leads to an increase in the number of messages and thus message startup costs, but has the potential to reduce communication overhead through the overlapping of nonblocking send and receive operations with independent computation.

Applying message strip-mining carelessly may result in performance degradation due to both increased message startup costs and network contention. Furthermore, performance is directly influenced by application characteristics; the data transfer size and the ratio between communication and computation affect the amount of available overlap. In this chapter, we present a systematic approach for evaluating if a vectorizable loop benefits from message strip-mining based on network parameters and application characteristics. The model has been incorporated into our compiler's loop optimization framework for fine-grained regular accesses. Starting from loop nests with affine shared array accesses, the optimizer first performs message vectorization, then chooses a good decomposition for strip-mining based on the performance model.

5.1 Optimizing Regular Communication in Loops

Consider the loop nest in Figure 5.1, where the arrays `a` and `b` are local and `r` is remote. Figure 5.2 displays the results of applying message vectorization to the loop, which can be performed either manually by the programmer or automatically by the compiler. As described in Chapter 2, the get call performs a nonblocking shared memory read, and a sync

call completes the outstanding communication operation. Performance is significantly improved by copying all remote values in one bulk transfer instead of performing a read in every iteration. One disadvantage with such a transformation, however, is that the processor must wait for the completion of the remote transfer before proceeding with the local computation. Figure 5.3 demonstrates the process of message strip-mining, whose goal is to hide this communication latency. The single bulk transfer from Figure 5.2 is divided into several blocks based on the strip size S, and the loop is then skewed so that the communication and computation code can be performed in a pipelined manner.

The computation is not the only source of overlap; as Figure 5.4 shows, loop unrolling can be additionally applied to allow the communication operations for different strips to be issued at the same time and increase the amount of overlap (both computation and communication) available in the unrolled loop body. In the ideal scenario (ignoring the overheads of initiating and completing remote operations), the communication time of each strip is completely overlapped with independent computation or communication from previous iterations, leaving only the overhead of transferring the very first strip. Because S is typically much smaller than the total message size N, the maximum performance gain can be significant.

5.2 Practical Considerations for Message Strip-Mining

Message strip-mining decomposes the transfer of data into a series of smaller transfers overlapped with local computation. It therefore could cause performance degradation by increasing the startup cost of communication. To understand the impact of combining the

```
for(i=0; i<N; i++)
  a[i] = b[i]+r[i];
```

Figure 5.1: Unoptimized loop, where r is remote.

```
h = get(lr, r, N);
sync(h);
for(i=0; i<N; i++)
   a[i] = b[i]+lr[i];
```

Figure 5.2: Vectorized loop.

```
h0 = get(lr, r, S);
for(i=0; i<N; i+=S) {
  h1 = get(&lr[i+S], &r[i+S], S);
  sync(h0);
  for(j=i; j<i+S; j++)
    a[j] = b[j]+lr[j];
  h0=h1;
}
sync(h0);
for (j=N-S; j<N; j++)
  a[j] = b[j]+lr[j];
```

Figure 5.3: Message strip-mining.

```
h[0] = get(lr, r, S);
for(i=0; i<N; i+=S*U) {
  h[1] = get(&lr[i+S], &r[i+S], S);
  ...
  h[U] = get(&lr[i+(U-1)*S],
       &r[i(U-1)*S], S);
  sync(h[0]);
  for(j=i; j<i+S; j++)
    a[j] = b[j]+lr[j];
  ...
  sync(h[U-1]);
  for(j=i+S*(U-1); j<i+S*U; j++)
    a[j] = b[j]+lr[j];
  h[0] = h[U];
 }
```

Figure 5.4: Unrolling a strip-mined loop.

transformation, one has to take into account both machine and application characteristics.

Machine Characteristics:

Message transmission time on a cluster can be divided into two components: a per-message cost affected by the latency of the network, and a per-byte cost affected by the communication bandwidth of the network. The LogGP [2, 38] model can be used to quantify the costs of both components. As illustrated in Figure 5.5, the end-to-end latency (EEL) of a message transmission can be approximated by o, the CPU send and receive overhead

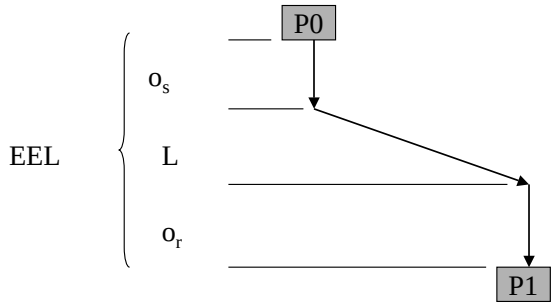


Figure 5.5: Traditional LogP model for sending a point-to-point message.

of a message, and L, the transport latency of the network hardware. Other parameters of relevance are G, the inverse network bandwidth, and g, the minimal gap required between the transmission of two consecutive messages. The values of these parameters directly influence the choice of strip size. A higher software overhead decreases the effectiveness of message strip-mining, since breaking a large message into smaller transfers becomes more costly. A large g makes unrolling less effective by increasing the latencies of pipelined accesses. Similarly, a higher peak bandwidth means a larger strip size likely will achieve better performance.

Application Characteristics: The data transfer size is perhaps the most important factor in determining the effectiveness of message strip-mining. Since each message incurs a fixed issuing cost, there exists a minimum size where the message becomes latency bound and does not benefit from further decomposition. An intuitive rule of thumb is that each strip transfer should be a bandwidth-bound message, so that the increase in message startup costs can be compensated by the performance gain from hiding the message transfer time.

Even with a large transfer size, message strip-mining is not useful unless we can discover enough computation to overlap. In other words, the computation cost of a strip should

not be significantly smaller than its communication cost. Loops with heavy computation overhead should benefit more from message strip-mining, since it allows more communication latencies to be hidden. Furthermore, a large computation overhead also implies that a smaller strip size should be used. Since network performance is generally orders of magnitude worse than CPU performance, it may appear that sufficient overlap could not be attained without a large amount of computation. This assumption, however, neglects the cost of memory access. Because remote data is typically transferred by RDMA directly to main memory (and not the cache), they must then be brought into the cache for further processing. Cache miss penalties, which are also incurred in the vectorized loop, represent additional computation overhead that can be overlapped with the nonblocking strip transfers.

5.3 An Empirical Study for Strip-mining

In this section, we construct a performance model for message strip-mining based on empirical data collected on the target clusters. Ideally, we would like to set up a system of equations based on the LogGP model to represent the communication costs of a given loop, then apply the overhead and bandwidth values of the network to solve for the optimal strip size. Finding the optimal message decomposition is unfortunately NP-complete, however, as it can be reduced to the classic integer programming problem [35]. Thus, we need to develop message decomposition heuristics that are easy to implement yet can achieve significant performance gains over a vectorized loop. A simple decomposition is to make each block equal-sized, so that the total communication cost for a given loop only depends on one variable. Even with this simple scheme, solving the above equations to find the

optimal strip size is still difficult due to the presence of recurrences. To make things worse, the LogGP model assumes an "ideal" network with no resource constraints and considers the parameters to have constant value. In practice, o, g, and G can all vary based on the message size [56], making the problem even less tractable. Thus, in constructing our performance model, we directly search for the best fixed message decomposition using synthetic benchmarks on the target platforms. We cut down the search space by considering only N and S that are powers-of-two.

We implement a micro-benchmark to study the potential for message strip-mining as well as develop heuristics for guiding the decomposition. Thread zero in the benchmark copies remote data and performs some computation on each element. The total transfer size ranges from 1KB to 1MB, while the strip size varies from 512 bytes to half the total transfer size (i.e., two strips). To study the effects of local computation overhead on strip-mining, we implement both a "light" and a "heavy" computation version. The light computation version performs a reduction on the transfered data; this in a sense represents the minimal amount of computation that can be performed. The heavy computation version calls the `sqrt()` function on each array element.

5.3.1 Overall Benefit of Strip-mining

Figure 5.6 shows the maximum speedup achieved by strip-mining for the transfer sizes (x-axis) examined. The value is calculated by dividing the time of the vectorized loop over the best execution time achieved for the given transfer size, with or without strip-mining. A value of 1 thus indicates strip-mining actually hurts performance. The heavy computation version is used. As the figure suggests, the effectiveness of strip-mining tends to increase

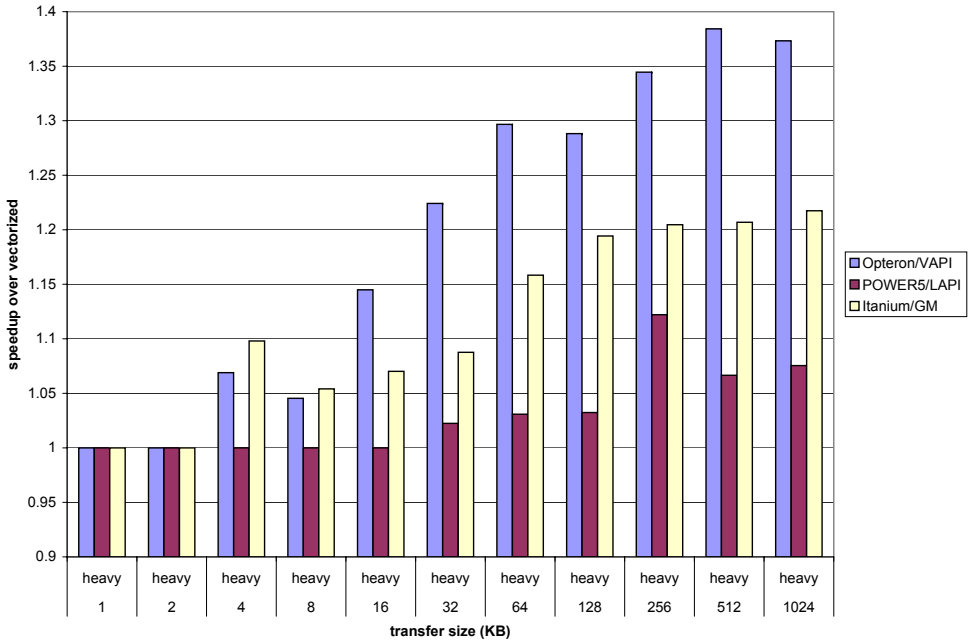


Figure 5.6: Maximum speedup achieved by message strip-mining, for transfer size from 1KB to 1MB.

with the transfer size, with small message sizes not benefiting from the optimization at all. For the large message sizes (524KB and 1MB), strip-mining could deliver a nearly 23% speedup, thus making it a potentially very worthwhile optimization. The POWER5/LAPI cluster benefits the least from strip-mining, due in part to its high software overhead (nearly 4us) compared to the other networks (about 1us)[1]. The best performance is achieved on the Opteron/VAPI system, with a close to 40% speedup for large message sizes. The minimal transfer size required for strip-mining to become effective is 4KB for the Opteron/VAPI

[1] The current GASNet LAPI implementation does not use RDMA, but support for it will be available in the future.

system, 8KB for the Itanium/GM system, and 32KB for the POWER5/LAPI system; the values reflect how efficient each system is at exploiting communication and computation overlap.

5.3.2 Effects of Loop Computation Overhead

The effectiveness of message strip-mining depends heavily on whether the vectorized loop contains a sufficient amount of computation that can be used to hide network latencies. The cost of bringing the transfered data into memory could be a major component in the loop's overall computation overhead. On commodity clusters, the network is not tightly integrated with the memory hierarchy, and instead uses RDMA operations that bypass the processor's cache. Accordingly, the computation cost for the transferred data is composed of two parts: 1) the cache miss penalties incurred by accessing the transfered data, and 2) the execution time required by the computation itself. While the second component obviously varies from application to application, the cache miss penalty is an inherent part of the computation overhead and does not depend on the type of computation performed.

The light computation benchmark thus serves as a good case study as to whether the local memory overhead provides sufficient overlap for strip-mining. Figure 5.7 presents the same speedup information as Figure 5.6 on the light version. The results suggest that there is essentially no limit on the minimum amount of computation a loop must contain before it can gain from strip-mining; as long as the loop accesses all of the transfered elements, the cache miss penalties will provide enough overlap. At 1MB, the local execution overhead accounts for an average of 21% of the benchmark time on the three systems, and strip-mining is able to achieve a 16% speedup.

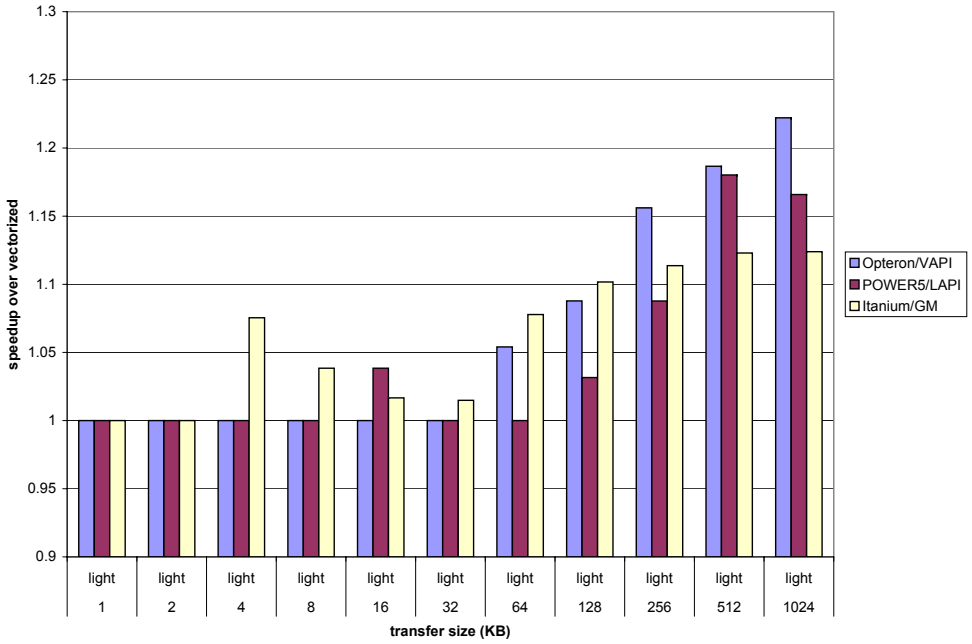


Figure 5.7: Maximum speedup achieved by message strip-mining, with light computation.

Comparing Figure 5.6 and Figure 5.7, we observe that the strip-mining as expected delivers a higher speedup for the heavy computation version, due to the higher amount of overlap available. More interestingly, the minimum transfer size at which strip-mining becomes effective is also affected. While for heavy computation loops strip-mining becomes effective as early as 4KB, the light version fails to benefit strip-mining until the transfer size reaches 64KB on two of the networks. This is because in the light computation version the memory access time is the only source of overlap, and at a small transfer size it is negligible compared to the communication latency. Thus, when deciding whether to strip-mine a loop, the optimizer must also take into account its computation overhead. Since an

accurate estimate of a loop's computation costs is difficult to obtain, especially at compile time, our optimizer uses a simple model by dividing the loops into computation bound and communication bound ones. The former uses the model predicted by the light computation benchmark, while the latter uses the heavy computation version. The model determines the threshold for strip-mining based on message size.

5.3.3 Selecting the Strip Size

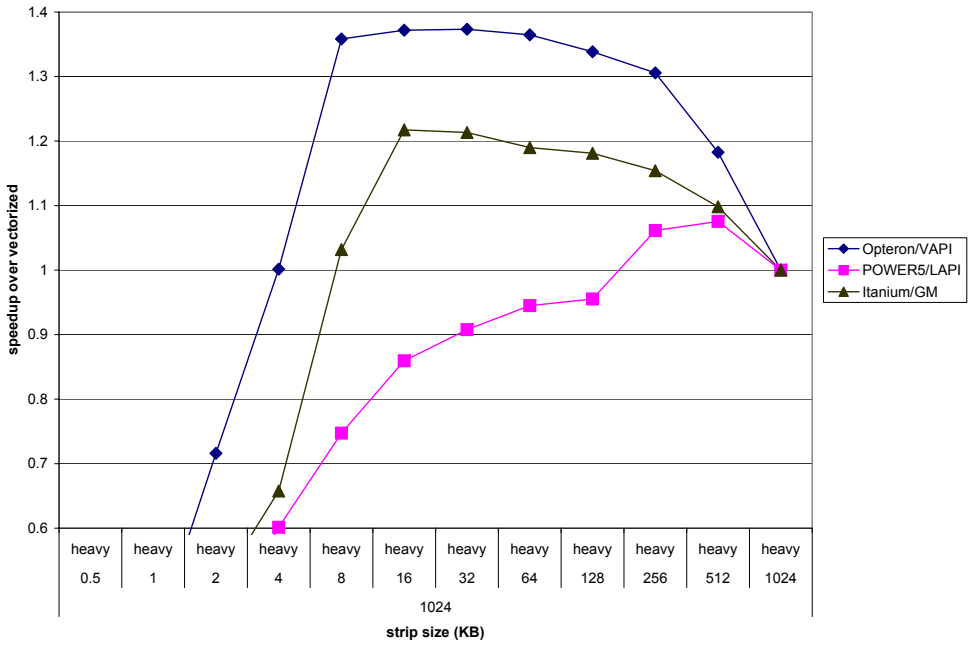


Figure 5.8: Speedup achieved by various strip sizes for a 1MB transfer, heavy computation.

Figure 5.8 and Figure 5.9 plot the speedup achieved by varying the strip size for a 1MB

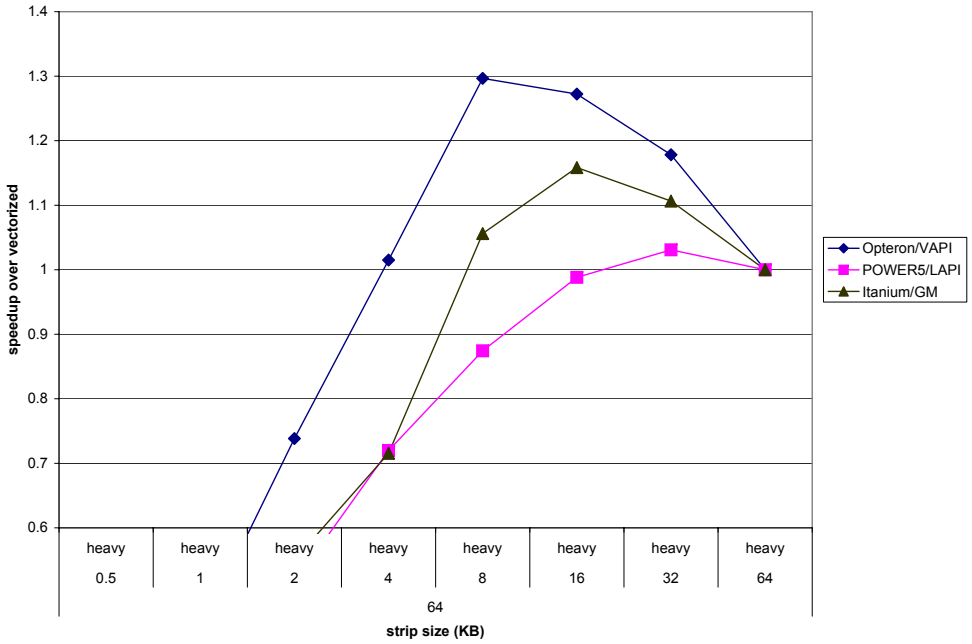


Figure 5.9: Speedup achieved by various strip sizes for a 64KB transfers, heavy computation.

and a 64KB transfer, respectively. The rightmost value represents the vectorized loop (strip size == transfer size), and is again used as the baseline for calculating the speedup. We make the following observations about strip size selection.

For the two networks (Itanium/GM and Opteron/VAPI) that exhibit more potential for overlap, there exists a wide range of "good" strip sizes. More formally, a good strip size is defined as one whose performance is closer to that of the best performing strip size than the vectorized loop. For example, for the 1MB transfer on the Opteron/VAPI system, any strip size between 8KB and 256KB will give performance that is within 5% of the best,

while offering a 30% speedup over the vectorized loop. Similarly, on the Itanium/GM system at least any strip size between 16KB and 256KB will offer good performance. This observation also holds for the light computation version, although the range of good strip size is narrower since the minimum message size needed to benefit from strip-mining becomes higher. This suggests that our performance model does not need to single out the best strip size, but simply has to find the typically large range of good strip sizes. The POWER5/LAPI system does not follow this rule, however, as the best decomposition strategy there is to have a small number of strips (2 or 4), even for transfers as large as 1MB.

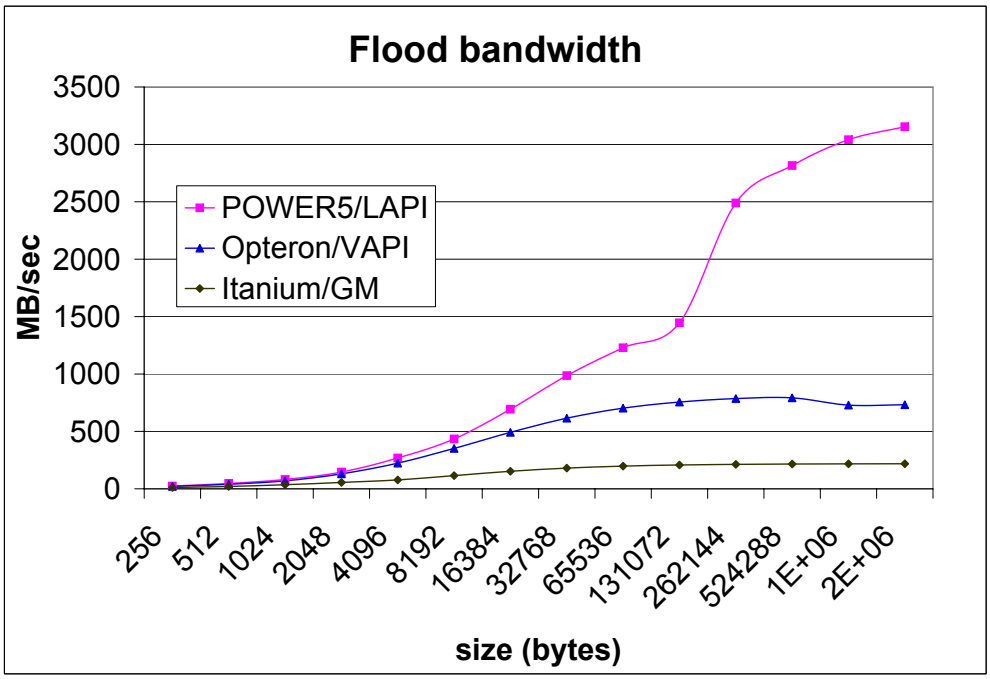


Figure 5.10: Flood bandwidth for blocking gets.

This apparent contradiction can be explained by examining the bandwidth of the three

networks in Figure 5.10. The POWER5/LAPI cluster has a significantly higher peak bandwidth compared to the other two clusters; this also means, however, the two networks will saturate to the peak bandwidth at much lower message sizes. Peak bandwidth on the POWER5/LAPI system, on the other hand, is not approached until 256KB, and the bandwidth at 256KB is in fact 80% higher compared to the bandwidth at 128KB. For this cluster, messages smaller than this threshold are unlikely to benefit significantly from strip-mining, since the finer-grained decomposition does not enjoy the additional bandwidth provided for large messages. If the transfer size is increased to 8MB on the POWER5/LAPI cluster, however, we again observe a wide range of good strip sizes.

We now present a heuristic for picking a good strip size. The smallest message size S_{peak} at which the communication layer saturates to the peak bandwidth serves as a good strip size for our transformation. Aggregation beyond this point no longer improves bandwidth, so for large messages it is typically profitable to divide them into S_{peak} chunks and overlap them with computation. The increased software overhead is insignificant, since each strip is still transfered at near peak bandwidth. Thus, for messages greater than S_{peak} our model selects it as the strip size. Messages below this threshold may still benefit from further decomposition, if there is sufficient amount of overlap to offset the bandwidth loss. For these small to medium size messages we set the number of strips to be two in an effort to simplify the model while still achieving overlap. Finally, we do not perform strip-mining for any transfer size that does not benefit from it in our experiments.

This performance model can be constructed on a platform with only three parameters S_{peak}, the peak bandwidth, M_{heavy}, the strip-mining threshold for heavy computation, and M_{light}, the strip-mining threshold for light computation. The latter two values are the

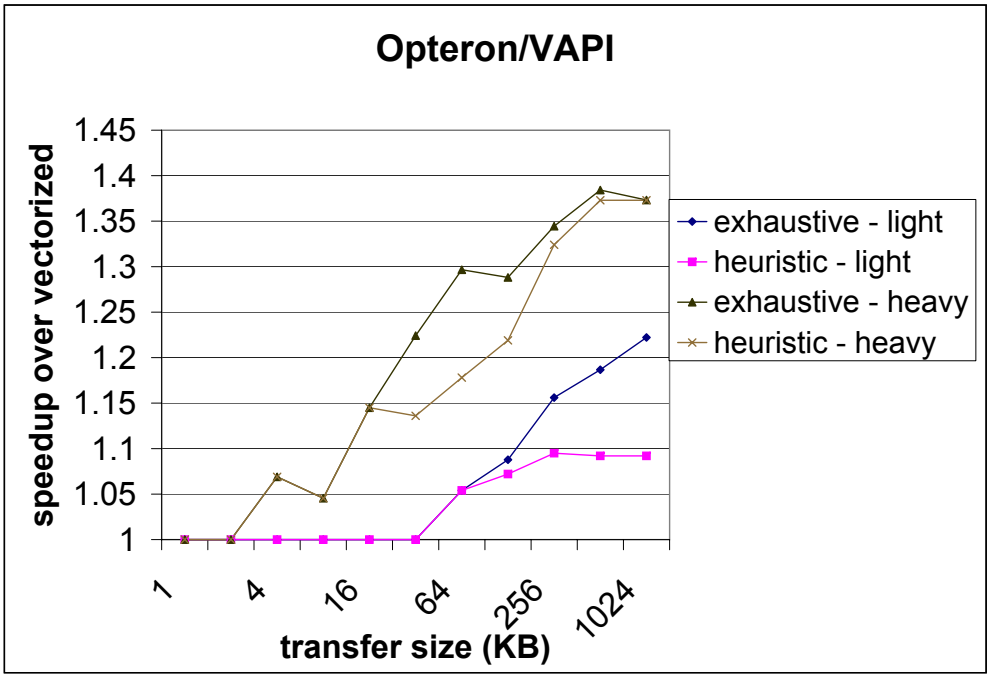


Figure 5.11: Accuracy of strip-mining performance model for the Opteron/VAPI cluster.

smallest transfer sizes at which strip-mining first becomes effective. We measure S_{peak} to be 256KB for the POWER5/LAPI cluster, 32KB for the Itanium/GM cluster, and 32KB for the Opteron/VAPI cluster based on experimental results. Figure 5.11, 5.12, and 5.13 compare the speedup achieved by our heuristics versus that obtained via exhaustive search. Despite our model's simplicity, it is able to approach the maximum speedup on all three systems, and in nearly all of the cases the model's performance improvement is at least 50% to that of exhaustive search. The largest error margin tends to occur at the medium size transfers, due to the higher degrees of performance variation among the different strip sizes. The model is highly accurate for small transfers, since they either do not benefit from

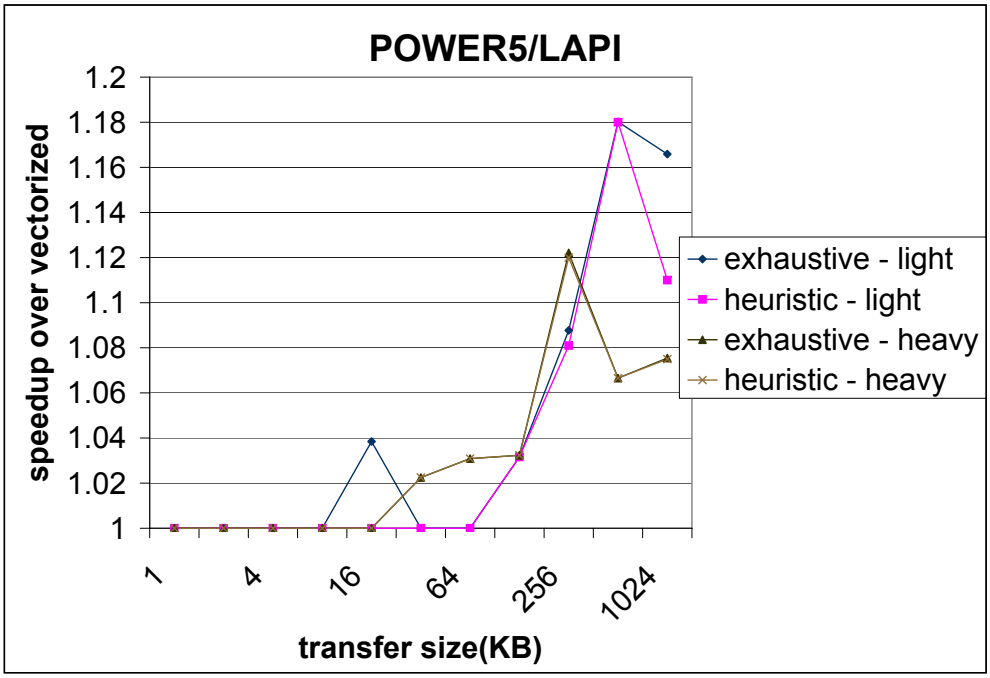


Figure 5.12: Accuracy of strip-mining performance model for the POWER5/LAPI cluster.

strip-mining at all or need only a small number of strips. The model is also effective for large transfers, for which there is a long range of good strip sizes to pick from.

5.3.4 Effects of Unrolling

As mentioned previously, loop unrolling could expose more potentials for overlap by increasing the number of messages that are simultaneously issued. Large unroll depths, however, may not be desirable in practice due to network limitations. In other words, we have to exploit the tradeoff between the two communication schedules shown in Figure 5.14. To

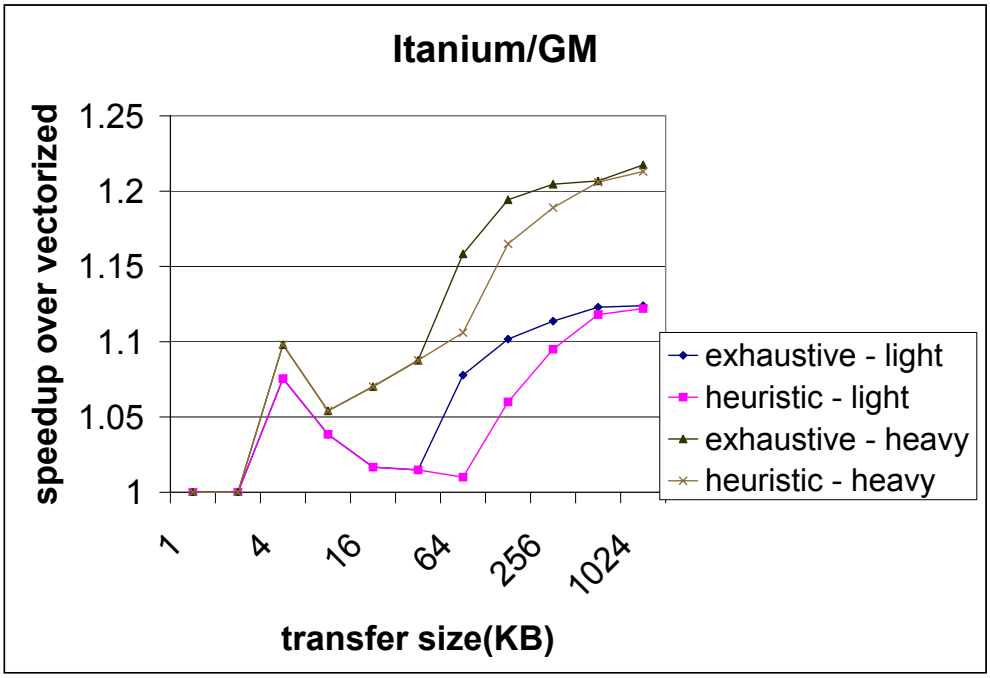


Figure 5.13: Accuracy of strip-mining performance model for the Itanium/GM cluster.

study the effects of unrolling, we modify the micro-benchmark to also perform unrolling, by dividing the loop into U equal-sized chunks and pipelining their execution. The value of U is varied from 2 to 128. Experimental results suggest that additional unrolling further improves performance over strip-mining by 1-2% , and the best unroll depth is typically 8 to 16. Since the performance benefit accrued from unrolling is relatively small, we do not consider unrolling in the performance model.

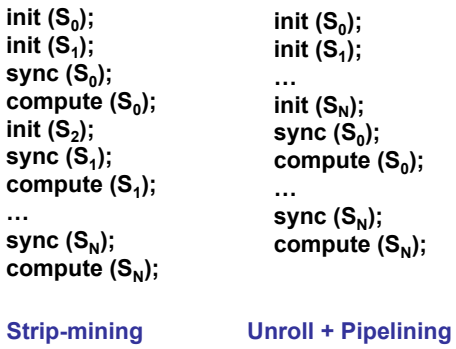


Figure 5.14: Communication schedule for strip-mining and unrolling.

5.4 Implementation

In this section, we summarize the structure of our loop optimization framework. We analyze and transform programs written in a shared memory style, with fine-grained remote array accesses. Message vectorization is first applied to singly nested loops, and the vectorization candidates are indefinitely blocked UPC arrays (i.e., all elements reside on the same thread). After the loop has been normalized, the analysis walks through the loop expressions and builds for each distinct array symbol a (lo, up) bounding box for its index values. 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. Vectorization inhibiting loop-carried array dependences are detected by intersecting the def and use sets of the loop.

Once a loop is vectorized, the analysis next determines whether it could further benefit from strip-mining. Since strip-mining does not change the iteration order of the loop,

any vectorizable loop can be strip-mined as long as the true dependence between the bulk transfer and the local computation code is preserved. In practice, this means if $ar[i + k]$ appears in the vectorized loop, any strip transfer would need to fetch k additional elements. Since important parameters such as transfer size and loop computation overhead may not be known at compile time, our framework uses a combination of compile time analysis and runtime support to dynamically perform strip-mining. The compiler generates template code that resembles the code in Figure 5.3, except that the strip size is computed by invoking a `get_strips` function before entering the loop. The function takes in as arguments the loop's transfer size as well as a flag indicating whether the loop is communication or computation bound. With these two parameters, the runtime analysis can calculate the strip size based on our performance model. The computation overhead is currently estimated by counting the number of array references and reduction operations in the loop. If this count is above a user tunable threshold, the loop is considered to be computation bound, and vice versa. All benchmarks we have experimented with so far have only communication-bound loops, and this rough estimate of computation overhead satisfies the optimization's needs. If more accuracy is desired, we can peel off a few iterations of the loop and measure their execution overhead directly. The generated loop itself is guarded by a conditional so that strip-mining can be disabled when it is not profitable.

Non-unit stride loops may also benefit from strip-mining, although the use of VIS aggregation calls described in Section 2.2 could result in better performance since they avoid copying extraneous data. Thus, we currently do not enable strip-mining for non-unit stride array accesses. Deeper loop nests with multi-dimensional arrays are in a sense already strip-mined, as there is natural overlap between the iterations of the outer loops. Long innermost loops, however, may still benefit from further blocking. Accordingly, we also

vectorize and strip-mine innermost loops with unit stride.

5.5 Experimental Results

We validate the strip-mining transformation on two of the NAS parallel benchmarks, FT and IS. The original UPC implementations use bulk remote transfers, which we manually convert into shared memory style code so that they could be analyzed and optimized by the loop framework. Specifically, the FT benchmark performs an all-to-all exchange followed by a local transpose, and the modified code merges the two steps into a single loop with shared array accesses. For the IS benchmark, the all-to-all communication is combined with the local computation code that determines key population. The other NAS benchmarks in Table 3.2 perform strided accesses to multi-dimensional arrays, and the individual chunks are too small to benefit from strip-mining. Similarly, the fine-grained benchmarks do not contain vectorizable loops and are also not used in the experiments.

Figure 5.15 presents the performance results[2]. Two version of the benchmarks are compared: the baseline that uses manually vectorized blocking communication, and the fine-grained version that has been vectorized and strip-mined based on our performance model. Speedup is calculated by dividing the MFLOPS rate of the strip-mined version over that of the baseline code. Class A input is used for FT and class B is used for IS. Our strip-mining transformation is effective on the FT benchmark, achieving close to 20% speedup on all three clusters. Since the transfer size is quite large for this benchmark[3], strip-mining is able to overlap the all-to-all communication latencies with local computation, while still

[2] 32 thread results on the Itanium/GM system are not shown, as both benchmarks fail to scale with or without our optimizations.

[3] $128MB/(THREADS * THREADS)$ for our input configuration.

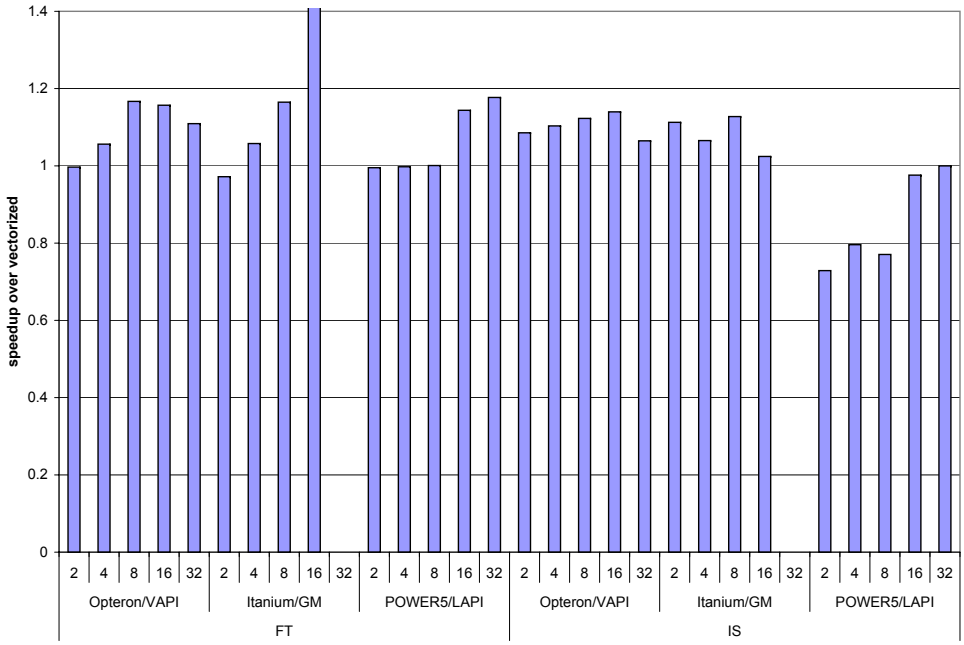


Figure 5.15: Speedup for message strip-mining on two NAS benchmarks.

amortizing the costs of increased message rate. Furthermore, strip-mining becomes more effective as the number of node involved in the global exchange increases, since it avoids the bandwidth bottleneck associated with the blocking all-to-all exchange.

For the IS benchmark strip-mining is also effective at hiding communication latency on the Opteron/VAPI and the Myrinet/GM cluster, achieving a more than 10% speedup in many cases. The transformation, however, suffers a substantial performance slowdown on POWER5/LAPI, due to an increase in the program's barrier synchronization time. Since each node on that cluster is an eight-way SMP, all threads will run on the same node for executions with less than or equal to eight threads. The increase in the number of messages caused by strip-mining leads to contention on the network interface, and the resulting load imbalance in turn aggravates the synchronization time. The FT benchmark is less susceptible to this issue, as there exists more computation available to be overlapped.

Chapter 6

Optimizing Fine-grained Irregular Accesses

Fine-grained accesses are inherently expensive operations on clusters with high network latency, and scalable PGAS applications typically use bulk memory copies rather than individual reads and writes to the shared space. Optimizations such as message vectorization in the previous chapter can also automatically aggregate regular array accesses in loops. Fine-grained sharing, however, is still useful for scenarios such as dynamic load balancing, event signaling, and distributed hashtables. These common irregular communication patterns are usually not amenable to our loop-based optimizations since they either use pointer dereferences or have dynamic access patterns (e.g., hash lookup). Manual coarsening of the fine-grained accesses into bulk communication is possible, but often requires non-trivial changes to the algorithm and data distribution. Thus, compiler algorithms that decrease the number, reduce the volume, and hide the latencies of the message traffic for

irregular applications can be very beneficial.

In this chapter, we describe an optimization framework for fine-grained irregular UPC applications. Using the SSA representation from the Open64 compiler, 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. 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.

6.1 Algorithm Overview

The candidates for our optimizations are shared pointer arithmetic and accesses in UPC. Pointer-to-shared variables in UPC are almost as expressive as normal C pointers, and can generally appear anywhere in the code where it is legal for a C pointer to appear. Within the Open64 framework, the Berkeley UPC translator extends the type system and uses the same internal program representation for shared expressions, which are distinguished based only 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 chapter are performed on the existing Hashed SSA (HSSA) program representation in Open64 [30]. 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[i]`, `**p`, `p->x`. HSSA uses a global value numbering approach to build a sparse program representation that captures the aliasing information for scalar and indirect memory reference (both pointer and array) expressions.

We present the following analyses: 1) partial redundancy elimination (PRE) for pointer arithmetic on shared types 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 [29] (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 accesses and coalescing. In addition, 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 a 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.

6.2 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 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. 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

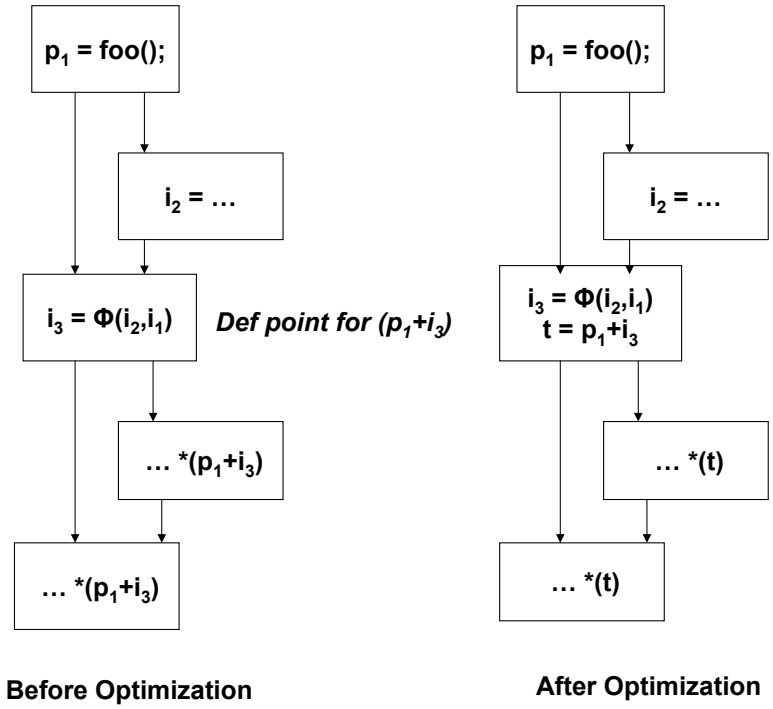


Figure 6.1: Redundancy elimination for shared pointer arithmetic.

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 6.1 illustrates the transformations of our algorithm. In the example, the shared pointer arithmetic expression $p + i$ can be computed immediately after the ϕ-assignment to i. We place the original expression there and assign its value to a newly created temp variable. All occurrences of the expression are then replaced with the temporary. While this 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`.

6.3 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. Communication operations on invalid addresses will generate runtime exceptions, and speculative communication movement is thus unsafe. Furthermore, **get** in our communication system uses RDMA to copy remote data directly into a stack-allocated temporary. All outstanding nonblocking reads must 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* [80]. An expression is *anticipatable* 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 anticipatable 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. Correctness is guaranteed if a **get** is placed at

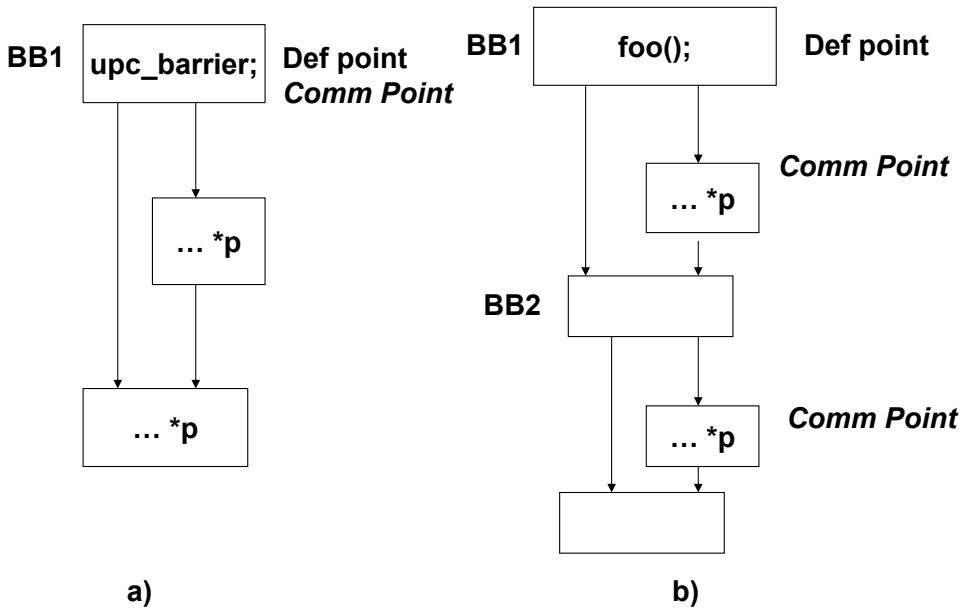


Figure 6.2: Split-phase analysis for reads. Communication points correspond to gets, actual use syncs.

each cp, as every cp has the property that it is dominated by the definition point and that e is anticipatable at cp. Such code generation, however, does not maximize the amount of overlap; for example, in part a of Figure 6.2 it is safe to place the communication before the branch, as $*p$ is anticipatable 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 6.2 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 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.

6.4 Split-phase Communication for Writes

```
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

A different algorithm is needed for remote writes, primarily because the HSSA repre-

sentation 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 6.3, 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 6.3(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 preclude 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.

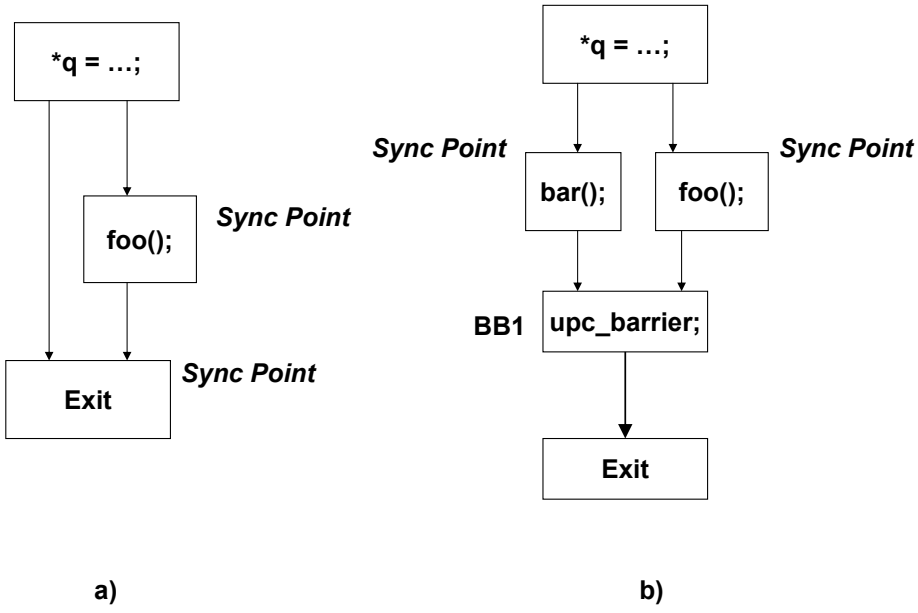


Figure 6.3: Split-phase analysis for writes.

6.5 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. Figure 6.4 illustrates the code transformation for coalescing.

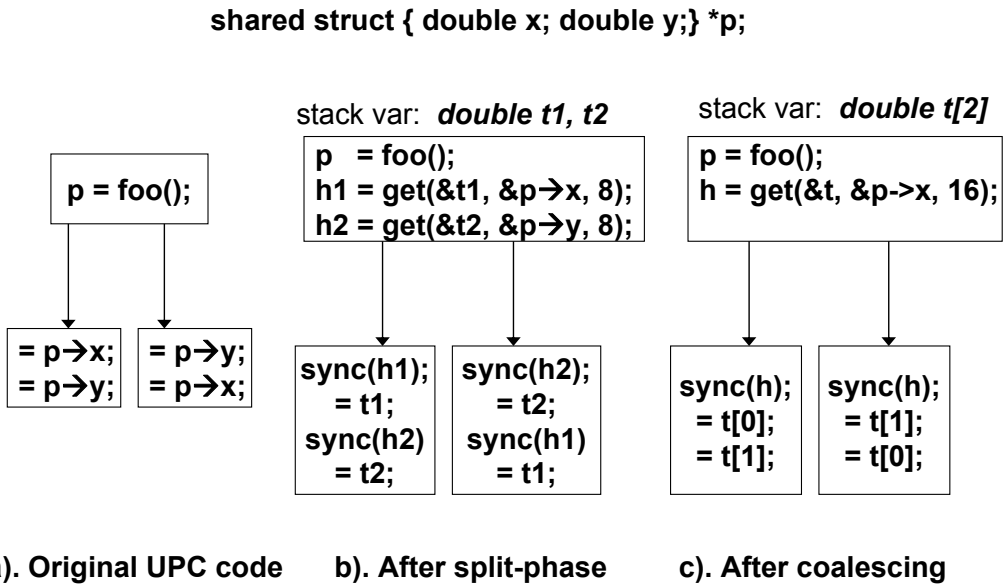


Figure 6.4: Compiler directed coalescing.

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. If it can be statically determined that $addr1$ and $addr2$ belong to the same processor, profitability for coalescing depends on the distance between the two addresses. Specifically, there are three methods for fetching the remote data: pipelining the two gets, static coalescing

using a single transfer to copy the bounding box, and dynamic coalescing using GASNet VIS calls [17], which performs runtime aggregation using active messages. We use micro-benchmarks to determine the tradeoffs between coalescing and pipelining on a network.

Figure 6.5 shows the performance difference between pipelining and coalescing for the Opteron/VAPI and the Itanium/GM cluster. The micro-benchmark compares the different methods for fetching two remote doubles with varying stride between them. We choose eight byte as the transfer size since it is the most common access granularity for irregular UPC applications. For reference we also include the performance for issuing two blocking gets. As the figure shows, static coalescing (*bounding*) performs best when the distance is small, though its performance diminishes once the stride grows to beyond 1KB, due to the extraneous data being copied. Dynamic coalescing, on the other hand, performs slightly worse than pipelining due to the small message count. Based on the results, we apply static coalescing for any two gets that belong to the same processor and have a less than 1KB distance between them. Pipelining is used if the locations of the addresses can not be resolved statically; while dynamic coalescing may become faster with a large message count (as we will see in Chapter 7), in our experience it is rare to find more than a handful of gets at a communication point.

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 [114], Zhu and Hendren present an algorithm capable of coalescing non-contiguous write operations to struct fields, by first fetching the "filler" fields in be-

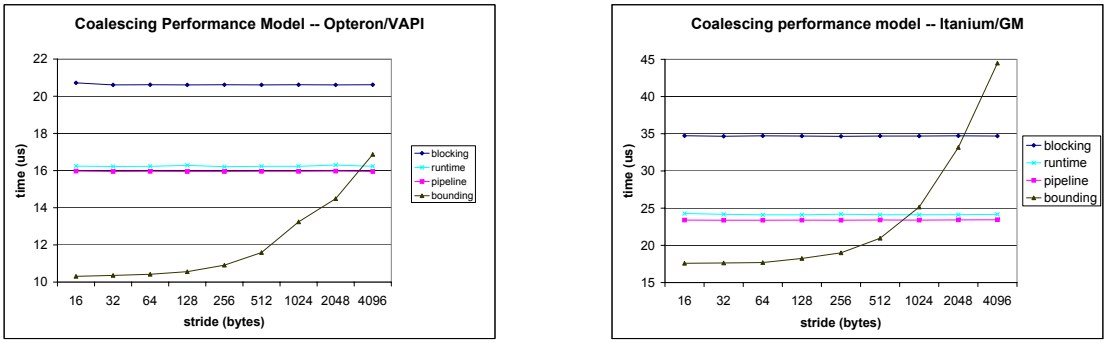


Figure 6.5: Performance model for fine-grained coalescing.

tween 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.

6.6 Example

Figure 6.6 provides a concrete example of how our compiler automatically performs the communication optimizations. The code is extracted from the psearch benchmark in Table 3.2, 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 gets before the lock call.

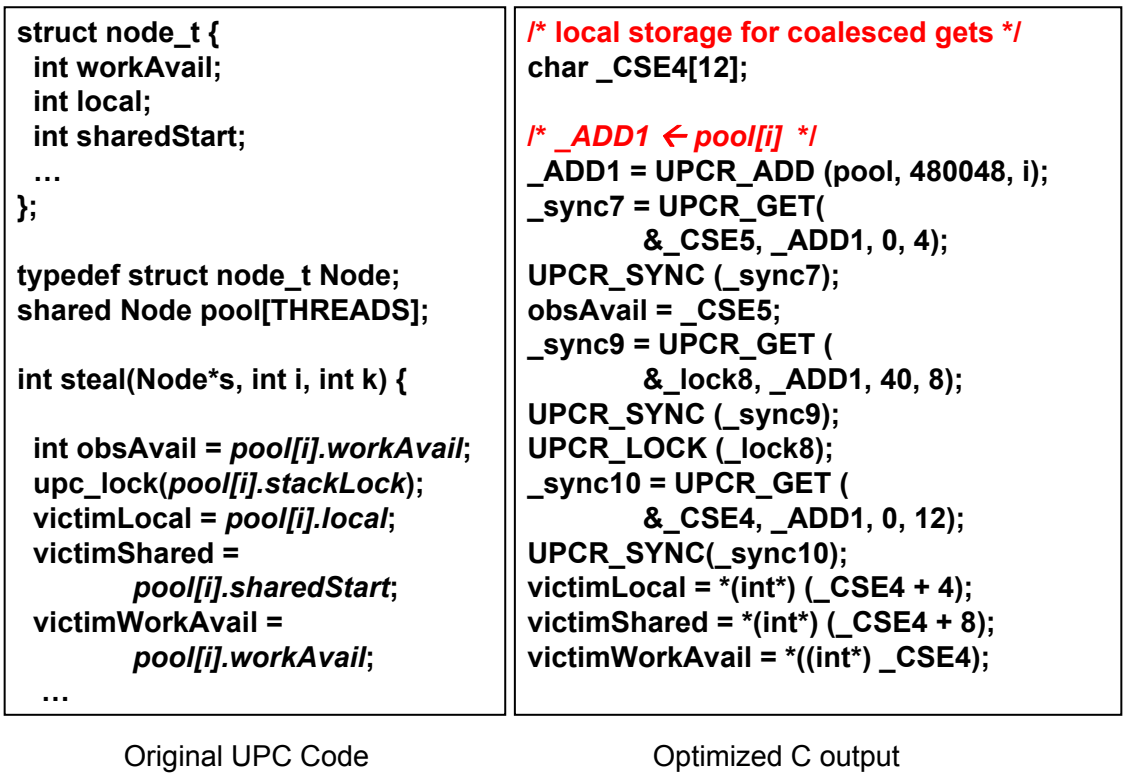


Figure 6.6: Sample code from optimized programs.

6.7 Experimental Results

We evaluate the effectiveness of the optimizations on the five communication-intensive UPC benchmarks with fine-grained accesses from Table 3.2. 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. Figure 6.7 presents the speedups achieved by our optimizations over the unoptimized version of the benchmarks. The speedup is measured by $(T_{base} - T_{opt})/T_{base}$, so that a value greater than zero indicates that our optimizations

improve performance. Each benchmark is discussed in more details below.

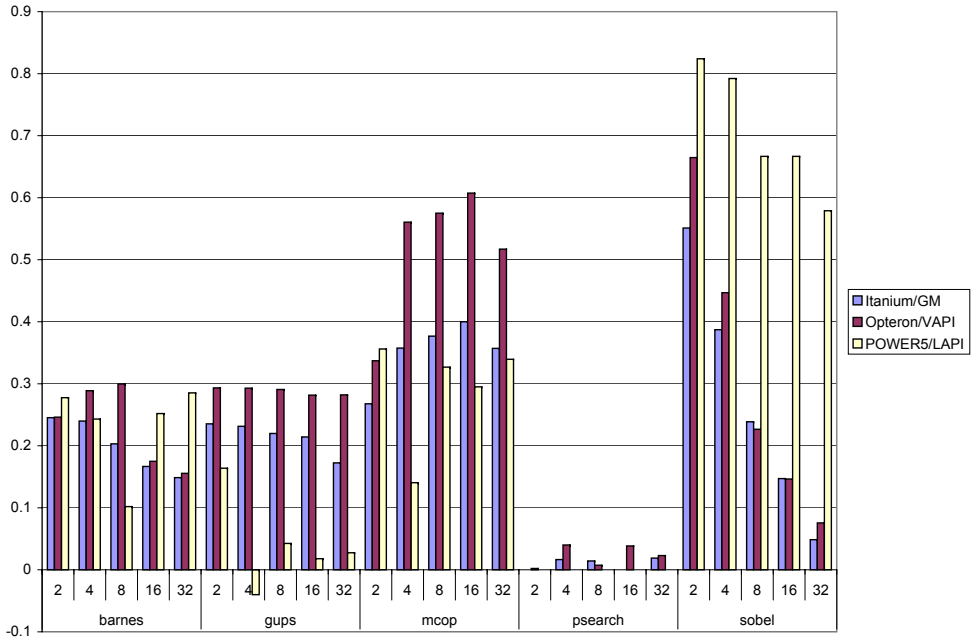


Figure 6.7: Optimization speedup, measured as fraction over unoptimized version.

Barnes: 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 22% speedup on average.

Gups: The read/modify/write loops in the benchmark are manually unrolled to allow for communication overlap. Due to the presence of indirect memory accesses, neither

redundancy elimination nor message coalescing could be applied, and the benchmark only benefits from split-phase accesses. Our optimizations improve the benchmark's running time by an average of 19% on the three systems. The least improvement is observed on the POWER5/LAPI cluster, as the lack of RDMA support on the network makes split-phase accesses less effective. Furthermore, since each node on that system is an eight-way SMP, on higher processor count we observe the effects of contention on the node's network interface. The best improvement is observed on the Opteron/VAPI cluster, with an average improvement of 27%.

Mcop: In this benchmark, the matrix data is distributed along columns, and communication occurs in the form of accesses to elements on the same row. Most of the benefits come from both redundancy elimination and nonblocking communication, as the inner loop of the benchmark contains several shared scalar reads that can be pipelined. Our optimizations improve the execution time by an average of 38%.

Psearch: This benchmark benefits the least from optimizations, since communication occurs only during 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. The small performance improvement comes primarily from the elimination of redundant shared pointer arithmetic. Our optimizations improve the execution time by an average of 1.5% across all platforms and processor configurations. No speedup is observed on the POWER5/LAPI system, as the large discrepancies between the computation and communication speed makes work stealing not worthwhile.

Sobel: 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 48%, with a maximum of over 80% on the POWER5/LAPI system. The speedup decreases at higher thread count due to increased network contention, since the communication volume for each thread remains unchanged due to the data layout.

6.8 Application Study

Experimental results from the previous section show that our framework can deliver significant speedup on fine-grained application kernels that are typically a few hundred lines in length. Furthermore, all three optimizations contribute to the reduction of the program's execution time. Here we present a major case study of one complete scientific application written in UPC, a CFD code developed at the Army High Performance Computing Research Center. The application combines automatic mesh generation with a fluid flow solver, and most of the communication takes place during mesh refinement as updates to dynamic shared data structures. This code provides an ideal optimization target for our framework, because it was originally developed for the Cray X1E [36], a vector supercomputer with hardware support for fine-grained remote accesses. As such, it contains a large number of fine-grained irregular accesses, in loops with dynamic trip counts that are difficult to vectorize. Achieving good performance on clusters for this program can thus be challenging.

The experiments were performed on three different clusters at the center. Two different setting were compared, one with and the other without our optimizations. Sixteen threads

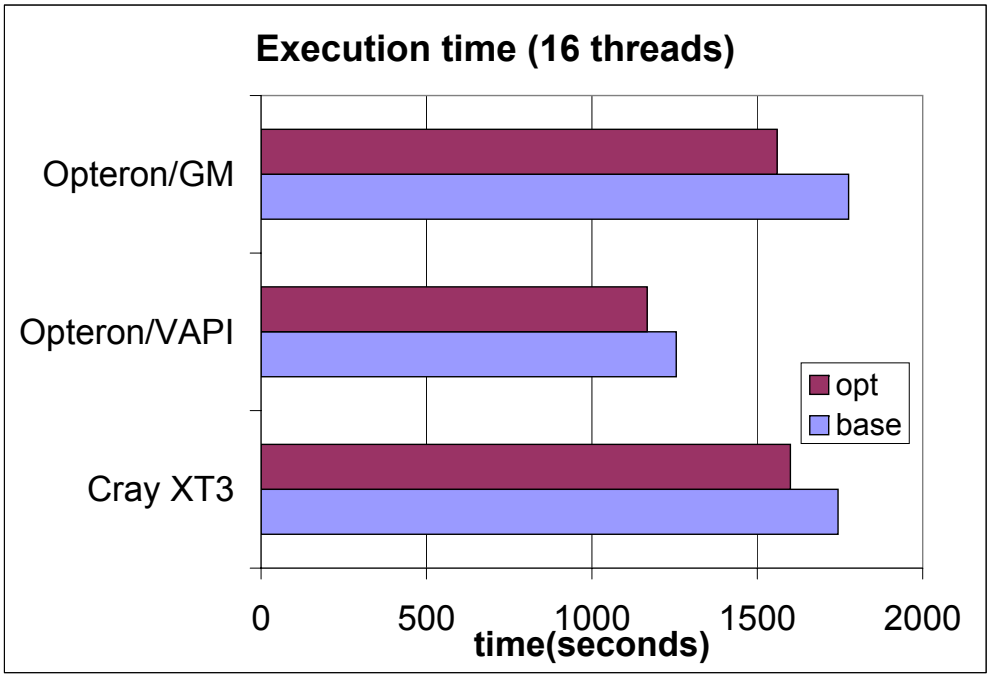


Figure 6.8: Performance on a CFD application.

were used in the experiments, and the benchmark executes ninety time steps. Figure 6.8 presents the results. Our optimizations are able to reduce on average about 10% of the total execution time. If we focus on the communication phase of the application that performs mesh refinement and coarsening, the reduction in running time is improved to 30%. We are not able to produce a detailed performance analysis due to lack of source code access, but the results suggest that our framework can be effective even for a large application written in shared memory style.

Chapter 7

Optimizing Bulk Communication

In the previous chapter, we presented a SSA-based optimization framework that automatically performs PRE and generates split-phase communication for individual gets and puts. The optimizations are very effective for fine-grained accesses, but a successful optimization framework also needs to support the bulk communication routines that most well-tuned PGAS programs use to amortize the high remote access latency on clusters. Extending our static analysis to support bulk memory transfer proves to be more difficult, since a bulk access could potentially touch arbitrary memory locations (the size of the transfer may not be known statically, and the starting pointer address could alias with other variables) and may not be safely reordered with other statements. Runtime support is thus often required in order to overcome the limitation of static analysis for scheduling bulk communication.

In this chapter, we present a runtime framework for automatic generation of nonblocking communication in UPC programs. Our system intercepts blocking bulk communication calls and aggressively reschedules and transforms these operations. Remote puts are syn-

chronized on-demand when a synchronization operation or other conflicting accesses are encountered. Remote gets are automatically prefetched by the runtime based on past access history. To further improve performance, the runtime automatically performs aggregation and selects the most efficient communication primitives available for special patterns such as strided accesses. We also present a number of techniques that help reduce the amount of runtime overhead associated with conflict checking as well as message initiation and synchronization.

7.1 Design and Implementation

The basic principle of communication overlapping is to issue the initiation of a communication operation as early as possible in the program schedule and the completion of the operation as late as possible. The constraints on the placement and scheduling of these operations are determined by application data dependencies. The candidates of our optimizations are the **upc_memget** and **upc_memput** communication calls in Figure 7.1. Both calls are part of the standard UPC library and perform semantically blocking memory-to-memory operations with relaxed memory consistency; the former copies (remote) shared data into the thread's private address space, while the latter updates (remote) shared data with contents from its private buffer. The **upc_memcpy** call is also handled in the special case where the source or destination is local, by rewriting it to the appropriate **upc_memget** or **upc_memput** call at runtime.

At runtime, any program path between two synchronization events becomes an optimization region. Most UPC programs use barriers to perform synchronization, but any

```
void upc_memget (void * dst, shared void * src, size_t nbytes);

void upc_memput(shared void * dst, void * src, size_t nbytes);

void upc_memcpy(shared void * dst, shared void * src,
                size_t nbytes);
```

a) Bulk communication calls in UPC

```
for( j=0; j<JMAX; j++ )
  {
    upc_memget(&backsub_info_priv_d(c,j,0,0),
               &rhs_th_d(c,1,j+1,1,0),
               (sizeof(double)) * BLOCK_SIZE * KMAX );
  }
```

b) Example of an aggregation region

Figure 7.1: Candidates for our nonblocking optimization.

statements that imply a strict memory access (e.g.,**upc_lock** library call) are also considered synchronization events. We monitor the sequence of communication operations (source/destination/size) within one optimization region. For any blocking communication call within a region, we decouple the initiation of the operation from its completion. Initiations of put operations are executed in the same program order, while their completions are dynamically delayed in the execution path until a synchronization event or conflicting operation is encountered. Initiation of get operations is speculatively executed at the synchronization point directly preceding the operation in the execution path, while their completion is executed in program order. For the case of gets the operation of our runtime

is equivalent to speculative prefetching.

Data consistency and dependence is preserved by a combination of dynamic conflict checks and double buffering. Communication aggregation is performed for special communication patterns such as strided accesses, and the system is carefully tuned to reduce the overhead for programs that do not benefit from overlap. This is achieved by a combination of compiler support and runtime profitability analysis. Similarly, our system reduces the overhead of prefetch initiation and metadata maintenance by using flow control heuristics and by overlapping them with communication.

7.1.1 Optimizing Puts

The upward mobility of the initiation of **upc_memput** operations is limited by control dependencies, so we focus on delaying its completion. The completion of a put operation can be postponed until either a synchronization event or a memory access that conflicts with the put is encountered. Ideally such code-motion inhibiting statements should be identified at compile time so that a **sync** call of the put could be placed right before them. However, in practice static analysis alone results in overly conservative synchronization placements, due to the imprecision of the alias analysis, especially for C-based languages.

Thus, automatic nonblocking transformation of **upc_memput** is best done dynamically in a demand-driven style by the runtime system, when the exact dependence information becomes available. When **upc_memput** is called, it is converted to a nonblocking call and added to a runtime-managed list of outstanding puts (a tuple of $\langle$ *handle, remote_addr, src_addr, nbytes*$\rangle$). Completion of an outstanding put operation is necessary at the following program points:

1. Synchronization events: all outstanding put operations are completed.

2. Statements that read or write the remote destination: any outstanding put operation whose target overlaps the remote memory region involved in the operation has to be completed, in order to maintain local dependencies. At these entry points, our runtime checks for conflicts and retires the operation involved, if any.

3. Statements that modify the local source: any outstanding put whose source is modified has to be completed. Since we do not rely on static program analysis, any local write could potentially fall into this category, and checking for local dependence would pose scalability problems. To address this, we eliminate local conflicts by using a nonblocking put call in our communication layer that allows the local source memory to be modified once the initiation call returns. Depending on the transfer size, this may result in the source memory being copied into a temporary buffer prior to sending.

Figure 7.2 depicts the runtime data structure for nonblocking put management. The outstanding puts are organized into an array of lists indexed by destination node, so that conflict checks will only be performed against puts with the same destination. The lists store and capture the FIFO order of the outstanding operations. To facilitate scalable completion at synchronization points of all outstanding operations, a separate array of pointers to the lists containing active operations is maintained. Source buffering is performed inside the communication layer and thus does not require special support.

Our approach requires minimal compiler support, as the compiler no longer needs to guarantee code motion safety but only needs to estimate the profitability of the nonblocking transformation. At each **upc_memput** site, the translator performs a forward traversal; if

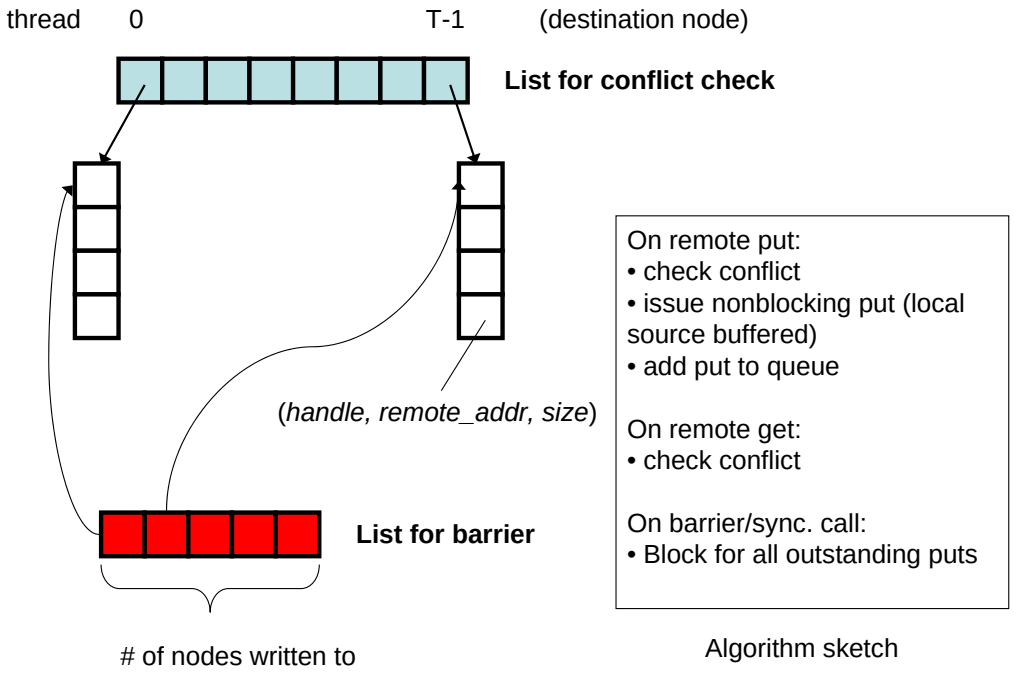


Figure 7.2: Runtime structure for nonblocking puts.

it encounters a communication call, a loop, or a function call before reaching a barrier or other synchronization events, it assumes that sufficient overlap exists and marks the **upc_memput** as a candidate for nonblocking optimization.

The dynamic conflict checking required by our approach constitutes a source of runtime overhead that we try to minimize. Since operations are maintained in per-target lists, the overhead of conflict checks grows linearly with the number of outstanding requests to a specific node. However, our application experience indicates that very often programs issue communication requests to memory regions whose address varies monotonically (e.g.,

traversing through a remote array in a loop). This observation is also validated on a number of numerical applications in [86]. Therefore, with each list of outstanding accesses we dynamically maintain a bounding box of the remote memory region involved, by keeping track of the minimum and maximum value in the list. If the target address falls outside the bounding box, clearly no conflict exists; otherwise, the runtime reverts to the slower but precise list scanning.

In programs that issue memory accesses monotonically, this technique reduces the conflict checking overhead to O(1) complexity. For programs that do not exhibit this property, a more sophisticated data structure such as balanced search trees could be used to reduce the asymptotic complexity of conflict checking[1]. In order to further reduce list traversal costs, during a conflict check the runtime queries the network layer to test if the earliest nonblocking put is complete, and removes it from the list accordingly. The runtime also imposes a tunable limit on the maximal put list length; when the maximum is reached, the runtime synchronizes on all the puts in the list before proceeding. Besides metadata maintenance, the biggest source of runtime overhead is the buffering required inside the networking system. Section 7.2.1 presents some performance results.

7.1.2 Optimizing Gets

The downward mobility of the completion for a **upc_memget** operation is limited by any use of the local destination buffer. Therefore we would instead like to move the message initiation up, effectively prefetching the remote data. The challenge here is that the runtime needs to have knowledge about future accesses; the translator could help determine the

[1]The stored intervals are guaranteed to be non-overlapping by virtue of the conflict-checking invariants, allowing them to be unambiguously sorted based on start address.

earliest safe place to initiate a nonblocking get, but its effectiveness is severely restricted by the inherent imprecision of static analysis.

Our solution is to exploit the structured nature of most parallel code. A large class of SPMD programs exhibit spatial and temporal locality in their communication pattern. While the data values being communicated are usually updated by local computation from one program phase to the next, the communication structure (size, source, and destination address of the **upc_memget** operations) often remains unchanged. For example, in a stencil algorithm like the Multigrid method, the communication ghost regions are allocated once and reused in subsequent phases. Similarly, studies on the MPI NAS benchmarks suggest a significant portion of the communication calls are *dynamically analyzable*, with constant parameters at runtime [44]. Thus, we contend that for an important class of applications, one can use past history to predict future access patterns by analyzing the communication when a phase is first executed, and automatically prefetching the gets for the phase in its subsequent executions.

Figure 7.3 describes the design for our nonblocking get optimization. We consider a program phase to be the set of statements executed by a thread after any synchronization event and before reaching the next synchronization event. Each static source level instance of a synchronization event is assigned a unique identifier by the compiler. When a given thread reaches a synchronization statement for the first time, we create a data structure to store the prefetch candidates for the subsequent phase. Subsequent gets are recorded and associated with the event. This operation is overlapped with the execution of the original get and does not add any visible runtime overhead.

On the first execution of a phase no speculative actions are performed. For subsequent

executions of a phase, prefetch calls are issued by each thread immediately upon entrance (right after a barrier), to maximize the amount of available overlap. To avoid local dependency violations or spurious updates to user data on a mispredict, the prefetched data are stored into runtime-allocated temporary buffers. When a **upc_memget** call is encountered, if it matches one of the prefetches, the runtime synchronizes the outstanding transfer and copies the desired data from the prefetch buffer into its destination. Otherwise, the runtime adds this previously unseen get to the prefetch list and issues the get as usual.

At the end of a phase, the runtime synchronizes and deletes any unused prefetches, to avoid performing useless communication in the future[2]. Any put operation within a phase requires conflict checks against the outstanding gets, to ensure that the application does not read stale values that violate data dependencies. No conflict checking is required when issuing the prefetches.

The runtime structures for nonblocking gets consist of two hash tables. One maps the current phase to a list of nodes that the thread has been prefetching from, while the other maps $\langle phase\_id, node\_id \rangle$ to a list of prefetches. Similar to the case for puts, the compiler algorithm complexity is greatly reduced, since prefetching temporary buffers and runtime conflict checking eliminate the need for static dependence analysis. The get optimization notably assumes that remote addresses appearing in a **upc_memget** operation remain valid for the lifetime of the program, such that subsequent speculative get operations can be safely issued without danger of causing a runtime fault (for example, if the relevant remote object has been freed). In Berkeley UPC this is already guaranteed because the shared memory allocator never unmaps memory pages.

[2] Mispredicted gets must be retired before their target buffers can be safely reused for subsequent prefetches, to prevent a race between two outstanding gets to the same buffer.

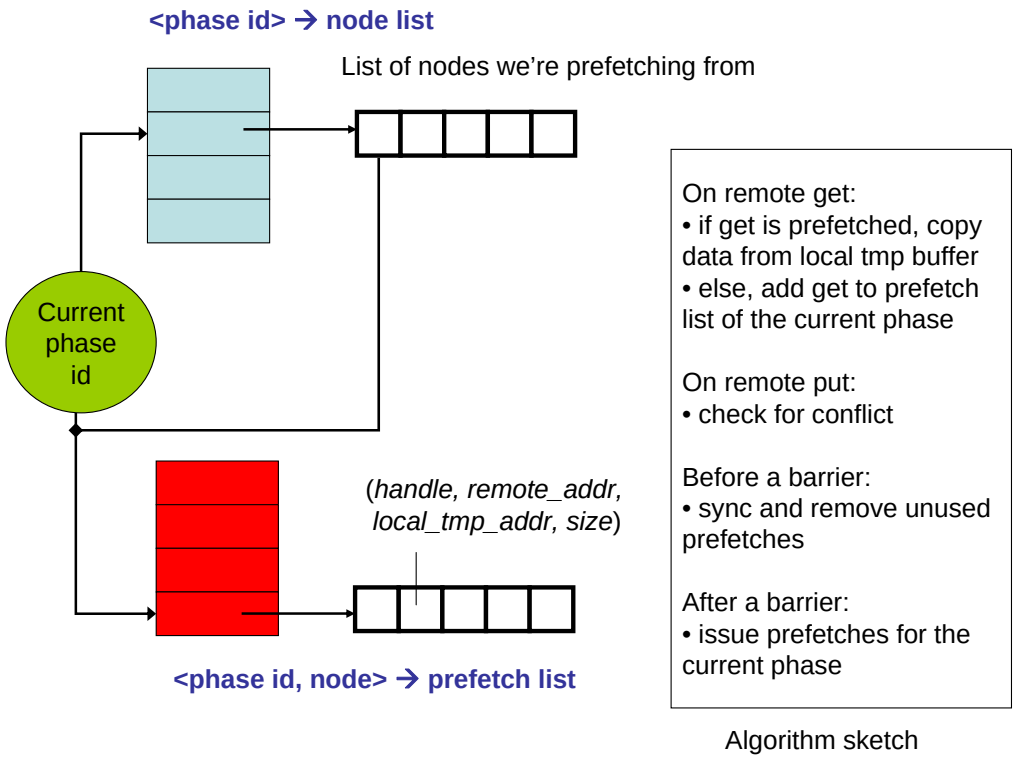


Figure 7.3: Runtime structure for get prefetching.

To avoid prefetching a get whose copy overhead outweighs its available overlap, each get's profitability is estimated before its addition to the prefetch list. Profitability is determined dynamically by comparing the amount of overlap (time of **upc_memget** - time of phase start) to the memory copy overhead. The estimation overhead can be completely overlapped by the cost of the remote get.

7.1.3 Automatic Communication Aggregation

Thus far in our framework, the gets and puts are issued individually to hide their communication latencies through overlapping. Many SPMD programs, however, have an alternating-phase structure with remote accesses grouped into a phase separate from local computation. The accesses in a communication phase are generally non-contiguous - however, combining puts and gets between the same pair of nodes into larger transfers is often profitable, as it amortizes the high per-message overheads of cluster network hardware over larger data payloads, thereby achieving a higher effective transfer bandwidth. Manual packing and unpacking of the non-contiguous accesses can be tedious and error-prone, however, and departs from the one-sided communication model since they require the cooperation of the remote thread. An optimization that automatically detects and aggregates the communication bursts would therefore be very useful.

We augment the framework to perform communication aggregation by targeting the VIS functions described in Section 2.2. When the translator detects at compile time a potential burst of gets or puts (e.g., a **upc_memget** inside a loop in Figure 7.1, taken from the NAS BT benchmark), it inserts special **begin_aggregate** and **end_aggregate** runtime calls to mark the *aggregation region*. When the runtime encounters a remote access inside the region, instead of issuing the access immediately (for puts) or adding it to the prefetch list (for gets), the runtime stores it into an aggregation queue, which is again organized based on the remote node. Upon exiting the aggregation region, the runtime issues the communication using a single VIS call per remote node, letting the network decide the best way to combine and schedule the accesses. Special patterns such as strided accesses and accesses with identical size are recognized and supported using the more efficient VIS calls

with reduced metadata overhead. Conflict checks proceed as usual inside an aggregation region, and a conflict terminates the aggregation region prematurely by switching to the default behavior described earlier.

In the current design, no buffering is done for the accesses in an aggregation region, and the translator must guarantee that local memory used by the gets and puts is not modified by other code in the aggregation region. As most communication phases (the candidates for our aggregation regions) are short and contain no computation code, so far our translator has been successful in proving that the **upc_memget** and **upc_memput** operations inside aggregation regions can safely be reordered without violating data dependences. We are planning to add a runtime check function that the translator can issue when it cannot verify if a local access may be in conflict with the **upc_memget** or **upc_memput** calls.

7.2 Performance Analysis

The last section presented the basic design for our automatic nonblocking communication framework. In this section, we provide a performance analysis for our framework. Our optimizations are primarily designed for clusters where remote communication latency is high relative to the processor speed. Since the system's overhead largely depends on the network and architecture parameters, we choose the Opteron/VAPI cluster in Table 3.1 for an in-depth investigation. Our findings, however, are applicable to any cluster systems where communication overlap is available (i.e., message overhead is smaller than the network latency).

7.2.1 Buffering Overhead

The put initiation cost depends on the message startup overhead of the network, as well as the overhead imposed by the communication layer's buffering of the local source. Figure 7.4 compares the cost of issuing a nonblocking put without and with source buffering; the semantic difference between the two is that the former poses the additional requirement that the local source memory cannot be safely modified until the nonblocking put completes. The latency of a blocking put is also included for comparison. For very small puts (up to 72 bytes on this GASNet network), the transfer is automatically performed using PIO and thus incurs no buffering overhead. Larger puts incur a cost for copying the source to a bounce-buffer, and beyond about 1KB this memory copy begins to affect the nonblocking put initiation time, and grows roughly linearly with the transfer size. Even with buffering, however, the cost of issuing a nonblocking put is still significantly less than the latency of a blocking put. Since the buffering is done at the discretion of the communication layer, a nonblocking buffered put should never perform worse than a blocking put.

Two main sources of overhead exist in the case of get operations. If the get has been prefetched, there may be a cost for the synchronization of the matching prefetch, which should never be higher than that of the original blocking get. Additionally, since the prefetch stores the remote data into a temporary buffer, the runtime needs to pay a copy overhead. Figure 7.5 measures the copy overhead by comparing the execution time of a local memcpy to that of a blocking get. While the copy overhead is negligible for small gets, it could equal about 30% of the communication latency for large transfers; this motivates our profitability analysis described in Section 7.1.2.

An additional potential performance penalty for buffering is that it may increase mem-

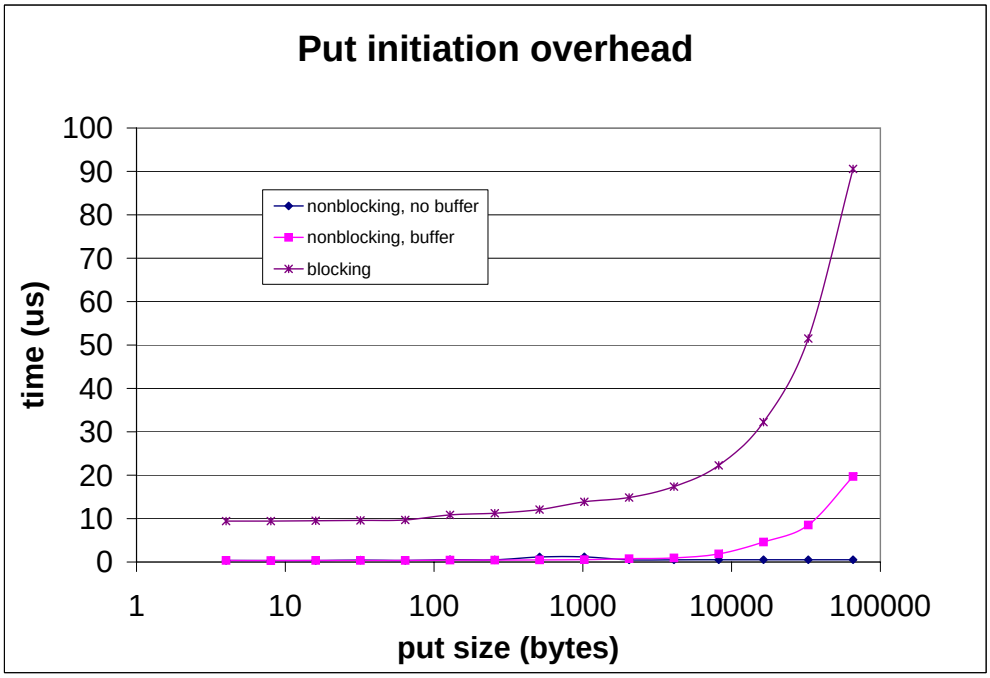


Figure 7.4: Put initiation overhead.

ory pressure by evicting live data from the caches. We have not observed this effect in our experiments, however. Without our optimization, the data fetched by a get sit in memory after an RDMA, and the application would need to pull it into cache on demand on first use. Since it is likely that the application will immediately use the data after a get, our copy operation has the effect of streaming the data from the prefetch buffer into cache. Finally, the penalties imposed by get prefetch "misses" in an RDMA system are almost entirely constrained to NIC resources and memory bus bandwidth. Mistakenly prefetched data should never occupy space in cache.

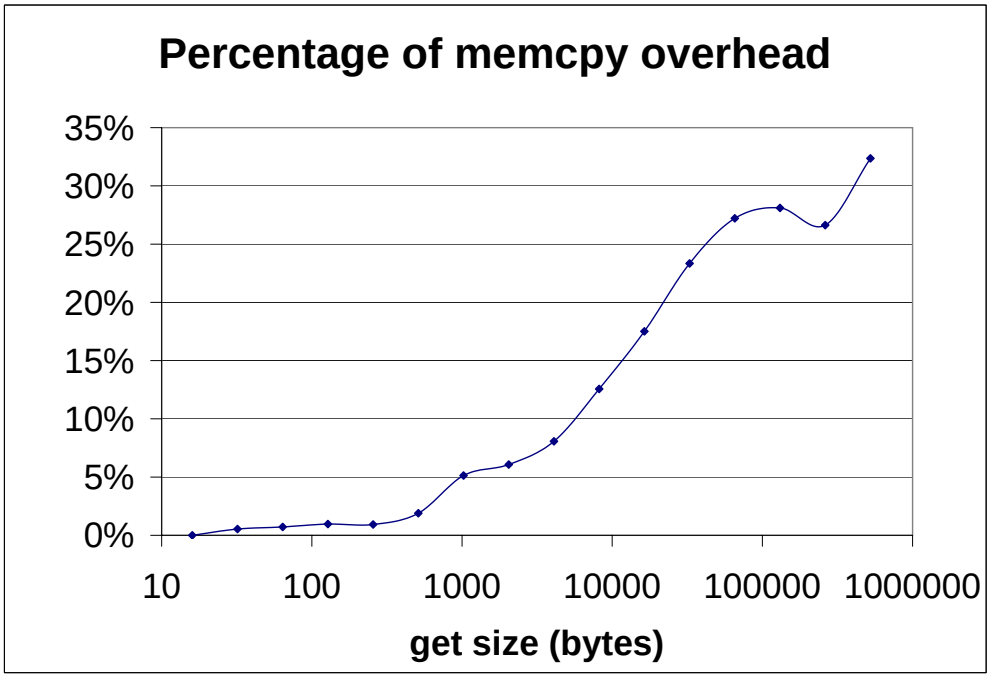


Figure 7.5: Memory copy overhead for prefetched gets, measured as T_{memcpy}/T_{memget}.

7.2.2 Communication-Related Overhead of Speculative Prefetch

The costs of prefetch synchronization before a barrier is affected by the number of mis-predicted prefetches in the phase, as the useful ones would have been synchronized earlier when their matching **upc_memget** call was encountered. To improve prefetching accuracy, the runtime removes from the list prefetches that are never used in the phase. Furthermore, the prefetch clearing time can be overlapped with the barrier latency by using *split-phase barriers*. In a split-phase barrier, a thread first notifies other threads that it has reached the barrier, then waits until all threads have executed the notification call. By inserting prefetch

synchronization code between the notify and the wait call, we can effectively overlap its overhead with the barrier latency. Micro-benchmark results on our target platform suggest that even in the absence of load imbalance, a barrier still takes $55us$ for 16 nodes and up to $87us$ for 64 nodes, well above the round trip latency for small- and medium-sized gets. We therefore expect the overhead of list clearing to be completely hidden by barrier latency for most applications[3].

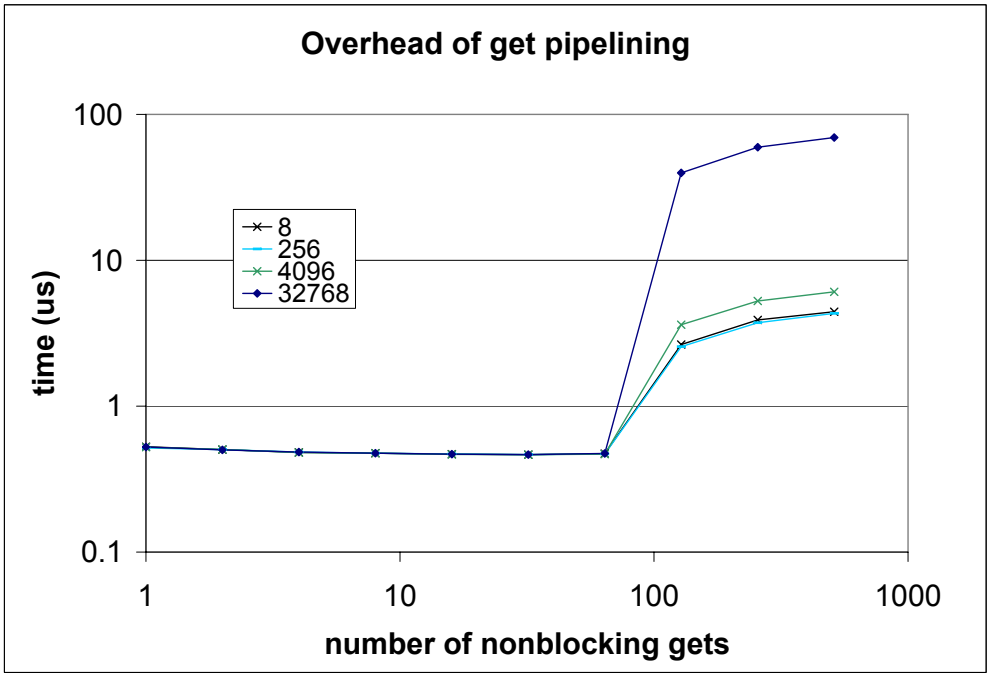


Figure 7.6: Prefetch initiation overhead, for each individual get.

The prefetch initiation overhead is primarily determined by the network's ability to

[3]This optimization does not apply to nonblocking puts, because UPC's memory model requires a thread to globally complete its put operations before issuing the notify operation. The reordering is legal on the prefetches, however, because the fetched data is never used.

issue consecutive nonblocking gets, specifically the message gap parameter in the LogP model [11]. Since the initiation occurs after a barrier and before user code executes, it falls in the critical path and we cannot easily hide this synchronous overhead. Figure 7.6 measures the initiation overhead of a sequence of nonblocking gets on our target platform. Four different message sizes are measured: 8 bytes, 256 bytes, 4KB, and 32KB. While the per-get overhead of initiating 64 nonblocking gets is the same as that of one get, at 128 gets performance begins to suffer dramatically for all message sizes, due to the (tunable) network queue depth being exceeded. Once the network queue depth is exceeded, subsequent initiation operations stall until the head of the queue is retired, and the stall time is primarily determined by the message size and the network bandwidth. Hence once you exceed the injection queue length, the injection time grows roughly linearly with the payload size.

We implement a simple flow-control mechanism to prevent a flood of messages during prefetch initiation. Instead of issuing all prefetches immediately after the barrier, the runtime issues them in 64-element chunks. When a **upc_memget** is encountered, our system checks (without blocking) if the previously issued prefetches to the same destination node have finished; their completion indicates that the network is likely not busy, and the system issues the next chunk of prefetches. Since the array of prefetches for each node is kept in FIFO order, our system will also first prefetch for **upc_memget** operations that occur earlier in program execution. This technique thus would work well for iterative code with locality in communication schedule, and these applications are exactly the kind we are targeting for the get prefetching optimizations.

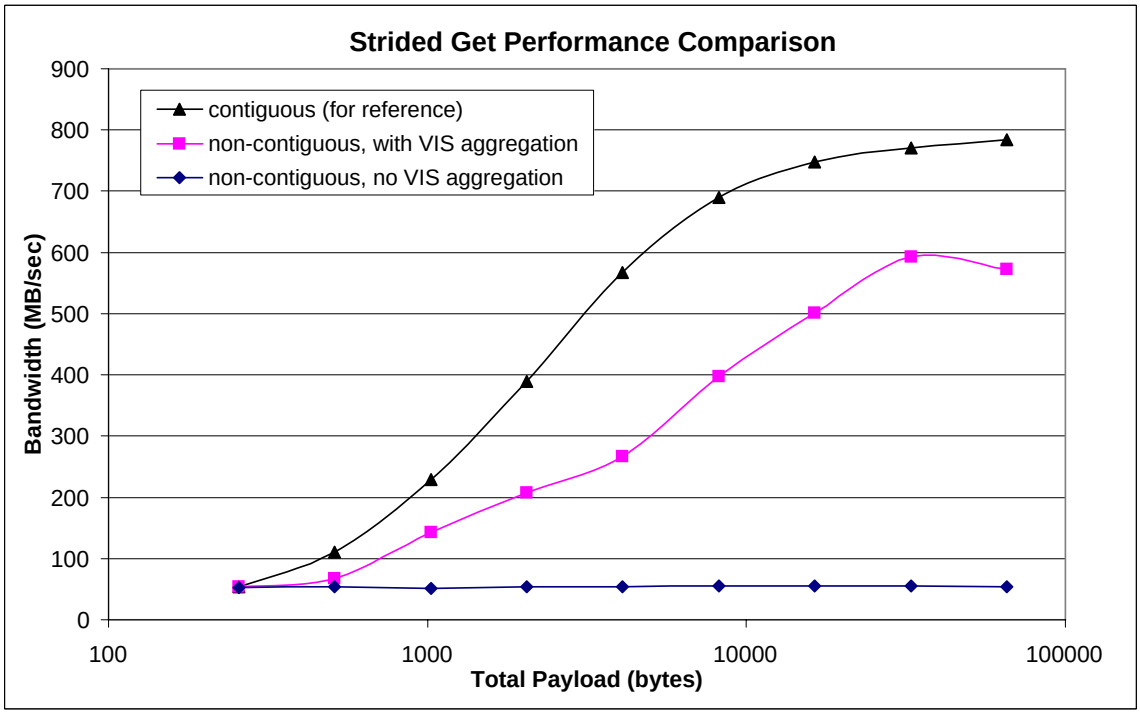


Figure 7.7: Strided get performance micro-benchmark.

7.2.3 Effectiveness of Communication Aggregation

Figure 7.7 compares the effective bandwidth of performing a logically non-contiguous get operation using a single aggregated VIS call versus a flood of individual (non-aggregated) get operations that achieve the same data movement. The micro-benchmark specifically measures the bandwidth for a 1-D strided get operation with 256-byte elements and a stride of 850 bytes at both the source and destination, while varying the number of elements to vary the total transfer size – this setup was chosen because it closely matches the most common access pattern for the BT benchmark. The figure demonstrates that performing the non-contiguous get using message aggregation provides a huge bandwidth advantage over the non-aggregated approach; the latter achieves consistently poor bandwidth due to

the high message count and relatively heavy per-message overheads. The non-aggregation curve is flat because in the experiment all NIC-level messages are 256 bytes independent of the total payload for the higher-level strided operation (shown on the x-axis). Thus, we observe a constant bandwidth equal to the flood bandwidth for 256-byte messages.

For comparison purposes, the figure also includes the raw bandwidth for fully contiguous transfers of the given total payload size (where no aggregation is necessary), to represent the theoretical maximal bandwidth for a get of the given payload size. The non-contiguous, aggregated transfer pays CPU and memory system overheads for gathering and scattering the payload from the non-contiguous source and destination locations to contiguous transfer buffers at the network layer, which explains the degradation relative to the raw contiguous transfer performance.

7.3 Experimental Results

We use the nine bulk communication benchmarks in Table 3.2 in our evaluation. Figure 7.8 presents the optimization speedup of our optimization framework. Three configurations are compared: a *blocking* version that uses fully blocking communication, a *manual* version where the communication calls are manually converted to nonblocking in a way that maximizes overlap, and finally an *auto* version with our optimizations. Due to the size of the cfd application, we have not manually converted the communication calls to be nonblocking; this underscores the importance of optimizations that could automatically generate nonblocking communication. Class B input size is used for the NAS benchmarks, while the gups-bulk benchmark executes eight million updates.

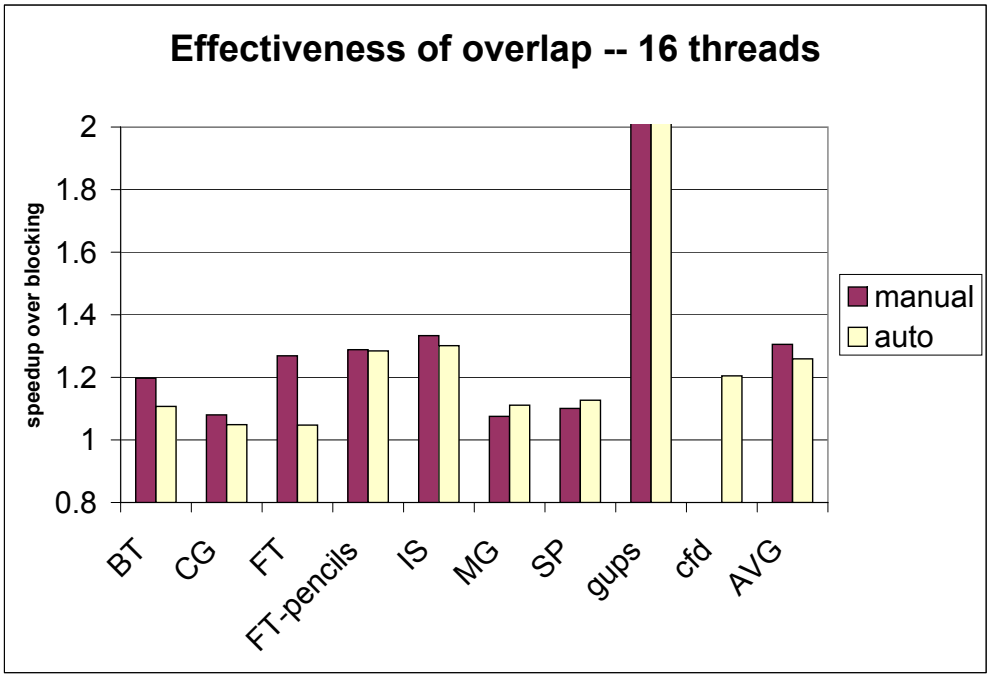


Figure 7.8: Optimization speedup for 16 threads.

Sixteen threads are used in the experiments, with one thread per node. The blocking version is used as the baseline for calculating the performance speedup. As expected, non-blocking communication is an effective method for hiding communication latencies, with the manual version speeding up the benchmarks by 30% on average; for some benchmarks (e.g., FT and SP) the communication latency is almost entirely eliminated. Our automatic optimization framework is somewhat less effective and achieves a 26% speedup, and is faster than the blocking version for all benchmarks.

The largest performance discrepancy between the manual and the auto version occurs

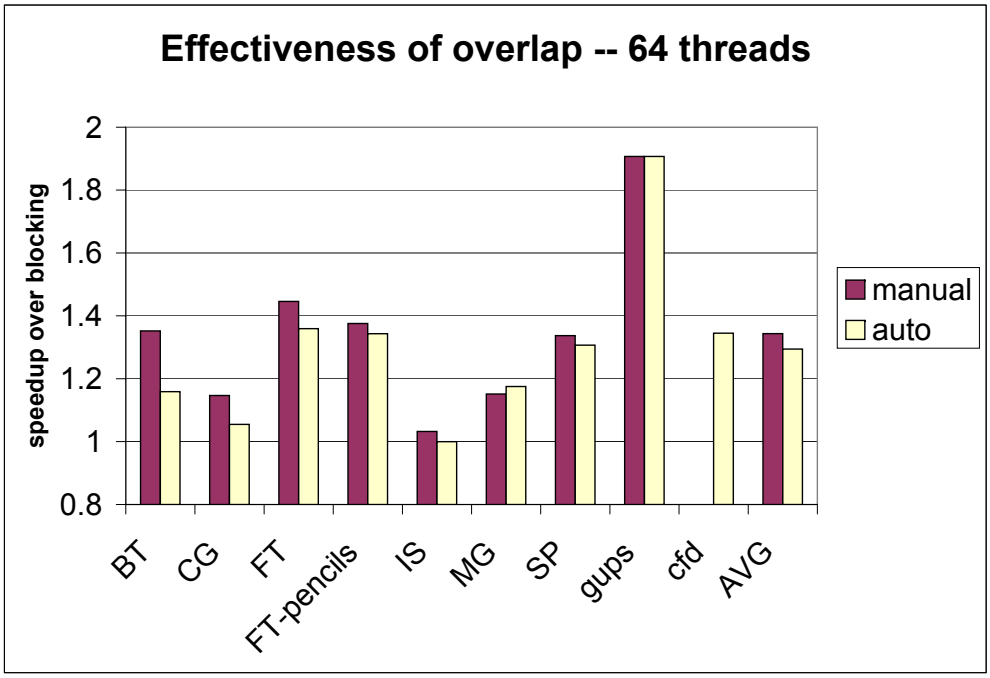


Figure 7.9: Optimization speedup for 64 threads.

on the FT benchmark, whose large transfer size induces a high local buffering overhead that negates most of the advantages from overlapping. Disabling network buffering for puts (this happens to be correct for this benchmark) brings the auto version's performance to within 3% of that of manual blocking, but has no effects on the other benchmarks. Thus, while our framework is effective for small to medium transfer sizes, a zero-copy consistency checking algorithm could further benefit applications with large-sized puts.

To test the scalability of our framework, we run the benchmarks with the same input size using 64 threads in Figure 7.9. Manual tuning improves performance by 34%, while

our system achieves a 29% speedup overall. Nonblocking communication is effective on all benchmarks with the exception of IS, which fails to benefit from either automatic or manual optimizations. The communication part of the IS benchmark is implemented as a collective all-to-all exchange, with the total number of messages in the program increasing quadratically as more threads are added. In the absence of independent computation to be overlapped, the resulting network contention reduces the effectiveness of pipelined communication. Compared to 16-processor runs, our automatic optimization becomes much more effective on the FT benchmark due to a reduction in message size; because the input size is fixed, quadrupling the thread count decreases transfer size by the same ratio, and therefore results in a much smaller buffering overhead.

7.4 Breakdown of Benchmark Performance

	blocking time	issue time	check time	sync time
BT	16.7	5.07	0	0.02
FT	182	151	1.42	0
FT-pencils	11.4	1.67	0.56	0
MG	34.5	2.88	0.07	0.36
SP	11	3.94	1.98	0.21
Gups-bulk	16.9	0.78	0	0.51

Table 7.1: Breakdown of nonblocking put time (16 threads). All values are in microseconds.

To further understand the performance characteristics of our system, we add timers to measure the time spent in the individual runtime functions. Table 7.1 presents a breakdown of the average execution time for each individual bulk puts in our framework. Sixteen threads were executed, and the reported data are from thread zero. Aggregation is disabled

in the experiments so that we can consider the execution time of each put individually. The blocking time column refers to the average cost of a blocking put in the base version. Each nonblocking put's execution time, in turn, can be divided into three main components: the time to issue the nonblocking put, the time to perform the conflict checks, and finally the time spent on waiting for the put's completion. Since buffering for puts is performed inside the network layer, the buffering overhead is included as part of issue time; this explains the high issue time of the FT benchmark. For several of the benchmarks the conflict check time is negligible, suggesting that our bounding box approach is effective at reducing conflict check overhead. Finally, the execution time for the blocking puts is significantly higher than that of the nonblocking puts for most benchmarks, indicating that our framework successfully hides most of the communication latencies.

	blocking time	issue time	check time	sync time	copy time
BT	15.2	1.3	0	5.7	0.2
CG	120.7	2.4	0.03	62	26.7
IS	1562	45.2	0	622	148

Table 7.2: Breakdown of nonblocking get time (16 threads). All values are in microseconds.

Table 7.2 provides a similar breakdown for the get prefetches. The blocking time column again refers to the average cost of a blocking get in the base version. The execution time for each prefetch can be broken down into four components: the prefetch issue time, the conflict check time, the time to synchronize the prefetch when we have a hit, and finally the time spent on copying the data from the prefetch buffer to user memory. We do not report the time to synchronize and remove useless prefetches, since it should be completely overlapped by the barrier overhead in most cases. Compared to puts, the gets are less likely to be completely overlapped, though some of the communication latencies can still be hid-

den through pipelining. Specifically, the high synchronization and copy time for IS can be attributed to both its large message size as well as its all-to-all communication pattern. Since the gets in our benchmarks are mostly confined in short pure communication phases, aggregating them is likely to be more effective than pipelining. Finally, conflict check time as expected is low on these get-based benchmarks, due to the scarcity of puts.

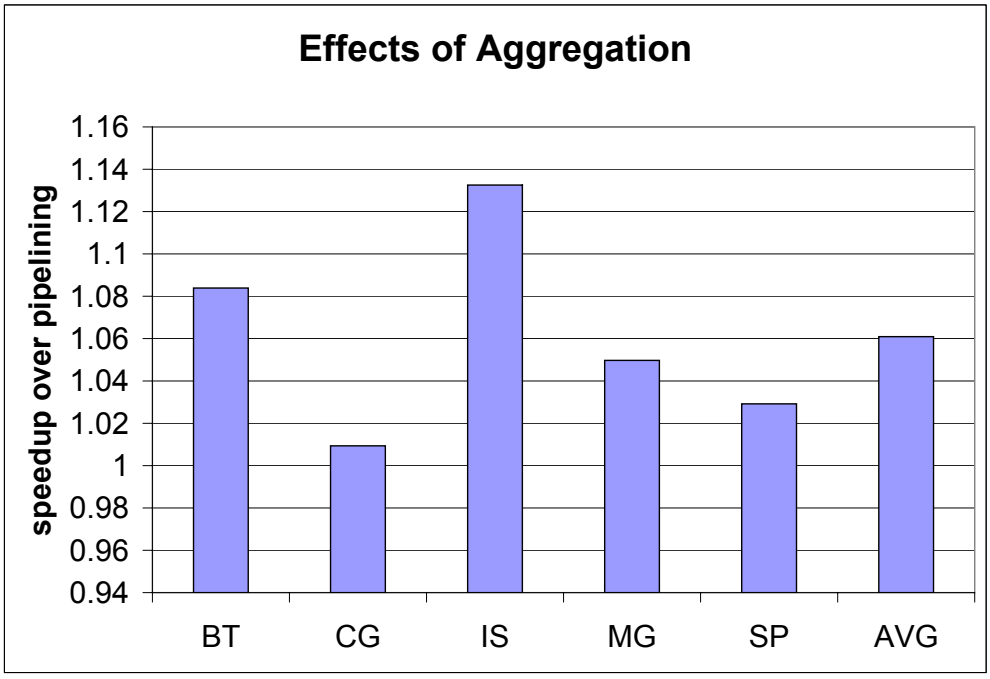


Figure 7.10: Performance comparison with and without aggregation.

Figure 7.10 presents the performance difference in our framework with and without automatic aggregation. Our compiler identifies aggregation regions in five of the benchmarks listed in the graph, and nearly all of the bulk communication calls in these benchmarks were converted into runtime VIS calls. The aggregation regions in the benchmarks con-

sist of strided accesses to multi-dimensional arrays on a remote node, with the exception of IS, whose aggregation region is a loop that performs all-to-all exchange. Ironically, IS benefits the most from the use of VIS calls, even though the communication calls in the region all have different destinations. This is because the compiler guarantees the absence of local conflicts inside the aggregation region, thereby eliminating the need for runtime buffering; as we have observed, this copy time can be expensive due to the large message size. BT also benefits substantially from aggregation because of its large number of strided small gets. The two put-based benchmarks (MG and SP) experience a somewhat smaller speedup, though the aggregation versions slightly outperform the manual pipelining ones. CG benefits the least from aggregation, as the size of each individual strided get is large enough that aggregation does not provide a significant bandwidth advantage over pipelining.

Chapter 8

Related Work

8.1 Parallel Programming Models

As the amount of prior work on parallel programming models is too extensive to be completely covered here, we will concentrate on other Partitioned Global Address Space languages as well as the current language efforts on the High Productivity Computing System (HPCS) program.

Titanium: Titanium [51] is an explicitly parallel dialect of Java designed for high-performance computing developed at UC Berkeley. Like UPC, it is a PGAS language that follows the SPMD execution model. To assist in the creation of large distributed data structures, Titanium provides a powerful multi-dimensional array abstraction that greatly simplifies the construction of grid-based scientific computations. Communication is generally expressed as copy operations over multi-dimensional arrays. To support a global address space memory model, Titanium augments Java's type system with type qualifiers

to express the locality and sharing properties of distributed data structure [71, 72]. Other notable Titanium features include an unordered loop construct for iterating through multi-dimensional arrays, user-defined immutable classes that facilitate pass-by-value semantics, and C++ like operator overloading support. A recent study suggests that Titanium implementations for three of the NAS parallel benchmarks can match the performance of the standard MPI/Fortran implementations, while requiring substantially fewer lines of code [40].

Compared to UPC, expressions and data structures in Titanium carry more high-level semantic information, which may allow for more aggressive compiler analysis and optimizations. On the other hand, more burden is placed on the compiler since more levels of abstraction exist between the language and the target code. Given the similarities between the compilation framework of the Berkeley UPC and the Titanium project (both perform source-to-source translation with C and GASNet as the target), a lot of the optimizations described in this dissertation could be applied to Titanium as well. For example, the optimization framework described in Chapter 7 could be integrated into the Titanium runtime to make the array copy operations automatically nonblocking. The fine-grained scalar accesses in Titanium could also benefit from our optimization framework in Chapter 6.

Co-Array Fortran: Co-Array Fortran (CAF) is an SPMD parallel extension of the Fortran 90 language [83]. A commercial compiler is available on the Cray X1, and a portable open-source implementation has also been released from the Rice University [31]. The Rice-CAF compiler achieves portability by using a strategy resembling that of the Berkeley UPC compiler: CAF code is translated into Fortran 90 with calls to either GASNet or ARMCI [82], another one-sided communication library. Several studies [32, 33] have

compared CAF and Fortran/MPI versions of the NAS parallel benchmarks, concluding that the CAF compiler can deliver roughly equivalent performance to that of MPI.

As part of these performance studies several communication optimizations were manually applied to CAF programs, chief among them *synchronization strength reduction* [102], which replaces collective barriers with uni-directional point-to-point synchronization primitives to increase the amount of concurrency. They also proposed adding mechanisms for programmers to declare nonblocking communication regions and to manually pack strided accesses. Our optimization framework in Chapter 7 performs similar transformations automatically, and thus has the advantage of increasing programmer productivity.

HPCS Languages: As part of DARPA's program on creating the next generation's High Productivity Computer System, three experimental languages are currently actively under development: the Chapel language from Cray [19], the Fortress language from Sun [3], and the X10 language from IBM [22]. Specifications for these languages have been published, but implementations for distributed memory architectures are not yet available as of this writing. The Chapel language employs a multithreaded parallel programming model that includes support for task, data, and nested parallelism. The language supports locality optimization on data and computation through data distribution abstractions called *locales* and data-driven placement of subcomputation. The Fortress language adopts a shared global address space as it data model, and also relies on multithreading as its control model. Parallelism is to be automatically inferred by the compiler, and synchronizations for shared variable accesses are planned to be satisfied by a transaction memory system. The X10 is a parallel distributed object-oriented language, and is also a member of the PGAS family. Targeted for clusters of multi-core SMP chips with non-uniform memory hierar-

chies, X10 allows users to explicitly specify an object's locality in terms of *places*. Task parallelism can be achieved with special language constructs such as *ateach* and *foreach*, while data parallelism is supported through global arrays and data structures.

The HPCS languages are similar to the PGAS languages in that they all provide the shared address space abstraction and permit programmer to have explicit control of data locality. They are different from SPMD languages like UPC, however, in that their execution models support dynamic parallelism. Unlike SPMD programming, where a fixed number of thread is created at startup with all of them executing the same code, the HPCS languages abstract away processor information from the program, and parallelism is introduced by the user through special language constructs. For example, Chapel supports forall loops and cobegin statements to create parallelism among loop iterations and statements, and X10 introduces the notion of *asynchronous activities* to help the user create thread-based parallelism. The HPCS languages thus have a productivity advantages over PGAS languages in that they provide more support for common parallel programming styles such as data and task level parallelism. On the other hand, more compilation and optimization efforts are required for these languages to achieve good performance. Optimizations for one-sided communication, however, are beneficial to any languages with a shared address space, and lessons learned from this dissertation may well be applicable to these new languages.

8.2 Optimizations for Parallel Programs

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 develop compiler algorithms to reduce communication overhead [5, 6, 21, 39, 62, 63, 114]. Traditionally the optimization problems have been studied along with communication code generation in parallelizing and data-parallel compilers. For example, the Stanford SUIF compiler automatically parallelizes and optimizes sequential programs on shared memory multiprocessors [5, 6, 49, 50, 73]. The Polaris system [9, 15] and the PARADIGM compiler [8, 47, 97] are other prominent compiler projects that automatically parallelize Fortran 77 programs. Similarly, a number of optimizations including communication vectorization and pipelining have been implemented for High Performance Fortran [4, 48, 99].

These parallelizing compilers share some common characteristics as the framework described in this dissertation. For cluster environments, both need to reduce the message count and volume through redundancy elimination and communication aggregation, and both hide network latencies through communication and computation overlap. Some of the optimization techniques for data parallel programs are also applicable to PGAS languages, but the fundamental differences between the programming models distinguish our work from previous efforts. The parallelizing compilers accept programs with serial semantics and thus have more freedom in program transformation, but have an added burden of detecting parallelism and managing data decomposition. Due to the complexity involved, today automatic parallelization is generally considered infeasible for large applications. Data-parallel languages avoid some difficulties of automatic parallelization with user directives, but sophisticated analyses are still required to map the fine-grained parallelism to the coarser-grained architecture. In contrast, PGAS languages are explicitly parallel and give programmer control of data layout, so that users can implement high-performance parallel programs without sophisticated compiler support. The role of compiler optimiza-

tions in PGAS languages, then, is to improve productivity by allowing programmers to write simple code with fewer manual optimizations. The PGAS model simplifies the analysis problem, but also presents new challenges since the optimization framework needs to approach the performance of well-tuned user code.

Another difference is that while the earlier parallel compilers tend to focus on array optimizations, our optimization framework supports both fine-grained read and writes as well as remote memory copies. Previous efforts typically focus on minimizing the number and volume of communication; the one-side communication model used by PGAS languages separates data transfer from its synchronization, and this exposes more opportunities for overlap. Thus, our framework successfully combines both communication aggregation and communication overlap to achieve better performance. Finally, by seamlessly integrating runtime support, we are able to bypass the limitations of static analysis and also obtain more accurate network performance models.

MPI is currently the de facto standard for cluster programming, and there has been a large amount of work on tuning the collective implementations [45, 46, 95, 103]. While these library-based optimizations can produce efficient collective subroutines, they analyze only one communication call at a time and are therefore unable to overlap communication with independent computation. Yuan et al. [63] proposed a combined compiler and library approach to optimize MPI programs, but their techniques are again limited to individual MPI collective routines.

8.2.1 Optimizing for Fine-grained Regular Accesses

A significant amount of work has been done on communication optimizations for array-based regular accesses, most of them in the context of data-parallel compilers. Initial work focuses on array optimizations for individual loops, while later efforts apply global program analysis across loop nests. Chakrabarti et al. [21] have implemented a global communication scheduling algorithm for High Performance Fortran that handles remote accesses in an interdependent manner. They have also explored using late placement to expose more opportunities for combining messages. Kandemir et al. [61] use a combination of dataflow analysis and linear algebra framework to perform optimizations such as message vectorization and message coalescing. More recently, an algorithm for coalescing communication in regular data-parallel applications has been presented in [24].

Wakatani and Wolfe [105, 106] introduce message strip-mining and analyze its impact for array redistribution in HPF and a code that implements a simple inspector-executor. The idea of decomposing message traffic also appears in [90, 91]. In order to alleviate switch contention, Prylli et al. implement transparent message decomposition inside the BIP networking layer. They model the steps involved in message transmission and derive formulas for predicting the optimal decomposition, and overall message passing performance.

Liu and Abdelrahman [74] propose a compiler transformation that peels off iterations accessing remote data in a parallel loop and schedules them at the end. This transformation is effective for data parallel programs but less so for PGAS languages, since in general every iteration may perform communication. Danalis et al. [39] describe a program transformation that overlaps MPI collective routines with computation loops, by restructuring the computation code into blocks and interleaving them with asynchronous communica-

tion. The transformation is similar to message strip-mining, but programmers are left with the burden of picking the tile size and number of transfers.

8.2.2 Optimizing for Fine-grained Irregular Accesses

In the context of communication optimizations that overlap communication and computation, perhaps the prior research that is most closely related to our techniques in Chapter 6 is Hendren and Zhu's work on parallel C programs [114]. 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. They have also developed a static locality analysis based on type inference algorithms for fast points-to analysis [113].

The optimizations presented in this dissertation operate under UPC's relaxed memory model, so that only local data dependency needs to be preserved when reordering shared accesses. More sophisticated parallel analyses become necessary, however, if a strict memory model such as sequential consistency is used. Krishnamurthy and Yelick [67] present compiler analysis and optimizations for explicitly parallel Split-C [37] programs with a global address space. Most of their work focuses on improving the accuracy and efficiency of the cycle detection [94] algorithm for SPMD programs, which enforces sequential consistency under reordering transformation. 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.

More recently, Kamil et al. [59, 60] developed compiler analysis techniques to reduce the number of memory fences required for enforcing sequential consistency, and used the

analysis to automatically convert blocking array copies in Titanium into nonblocking operations. Their optimization significantly improves the performance of two matrix vector multiply benchmarks. Su and Yelick [98] have developed an array prefetching algorithm for irregular array accesses in Titanium using inspector-executor techniques. Their optimization supports loops with indirect accesses ($A[B[i]]$) and uses a performance model to pick the most efficient data access method.

Lee et al. [69, 70] 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 nonblocking communication optimizations are not relevant.

Barton et al. [10] describe the design and implementation of a UPC compiler for the BlueGene/L supercomputer, and report good performance scalability up to hundreds of thousands of processors. For the Gups benchmark, their compiler applies a remote update optimization, where the compiler recognizes a read-modify-write operation on some shared address and automatically transforms it into an active message based update on the remote processor. They also describe an affinity removal optimization for upc_forall loops, though their technique is limited to the case where the affinity expression is the loop induction

variable.

8.2.3 Optimizing Bulk Communication

Iancu et al. [55] utilized virtual memory support to overcome the limitation of compiler analysis for bulk memory transfers. When a nonblocking communication is issued, the local memory pages involved in the communication are marked as having no access through the `mprotect` system call, so that synchronization will happen on-demand when the computation statement access those pages and receive a memory protection error. The scheme maximizes the amount of overlap, but is not portable; the POSIX standard [88] requires that the protected memory be obtained from the `mmap` call, but the local memory involved in nonblocking communication generally either lives on stack or comes from malloc, which may not use mmap to obtain new memory.

Intelligent runtime systems have received renewed interest in recent years and show very promising potential in terms of performance, scalability and programmer productivity. The approach most closely related to ours is the Charm++ [23] runtime. Charm++ provides for latency hiding through an abstract execution model based on processor virtualization and message-driven execution. Charm++ decomposes the computation and achieves overlap by rescheduling threads that block for communication. In a sense, our approaches are converses – Charm++ schedules computations while our optimizations aggregate and schedule communication. Sorensen and Baden [96] present a data-driven programming model and run time library that manages communication pipelining and scheduling through task graph, actor-like execution.

Software caching of remote memory has been studied extensively in the context of dis-

tributed shared memory (DSM) systems [20, 58, 64, 93], and is available in a number of UPC compilers [34, 92]. While remote reference caching can be very effective for shared memory style code which is oblivious to data locality [112], it may not help programs for which the user has manually replaced fine-grained accesses with bulk communication. Most well-tuned PGAS applications use bulk communication to amortize the high remote access latencies on clusters, and our optimizations are specifically designed to further improve the performance for these kind of programs.

Numerous hardware and software prefetching schemes have been proposed to hide the main-memory access latencies [28, 76, 78, 104]. The primary difference between uniprocessor prefetching versus prefetching in a distributed environment is that the latter lacks a hardware cache that can store prefetched data, and thus has to pay an extra copy overhead to deliver the data to the client. Prefetching accuracy is also more important, since a mispredicted prefetch wastes not only bandwidth but also message startup and cleanup overhead.

Chapter 9

Conclusions

Effective use of communication networks is critical to the scalability of parallel applications. Partitioned Global Address Space languages have proven effective at utilizing modern networks because their one-sided communication is a good match to underlying network hardware. In this dissertation, we have described an optimization framework that improves the performance of UPC programs on cluster architectures. The communication optimizations serve dual purposes: aggregating individual reads and writes to reduce message count, and making blocking remote accesses split-phase to achieve communication and computation overlap. The framework covers three major access patterns in PGAS languages: fine-grained array accesses in loop nests, fine-grained irregular pointer accesses, and bulk memory transfers. A combination of compiler and runtime support is used to increase the optimizations' effectiveness. The framework is transparent to the programmers and thus frees them from the details of communication management.

To summarize the major results of this dissertation:

- We have demonstrated that the source-to-source translation adopted by our compilation framework incurs only a small overhead, and introduced several techniques to lower the cost of shared pointer arithmetic and shared local accesses in UPC. We have also shown a strategy for compiling UPC's parallel loop constructs.

- We have implemented an optimization framework that effectively reduces the overhead of fine-grained array accesses by hoisting them out of loop nests. Our framework extends the traditional communication vectorization with strip mining, so that the transformed loop could also benefit from communication and computation overlap. We have also developed a simple heuristic that can help the optimizer or the programmer select a good strip size for message decomposition.

- We have developed a static optimization framework for fine-grained irregular accesses in UPC programs. The framework consists of three optimizations: a PRE optimization that eliminates redundant shared pointer arithmetic and shared memory accesses, a split-phase optimization that exploits communication and computation overlap, and a coalescing optimization that reduces the message startup overhead.

- We have presented a runtime framework for optimizing bulk synchronous programs by transparently converting the blocking remote bulk transfers into nonblocking ones in a way that maximizes the overlap of communication with computation, while still maintaining the memory consistency guarantees of the language. The system also recognizes special access patterns and automatically aggregates communication to further improve performance.

Although sophisticated compilation technology is not a prerequisite for the success of PGAS languages, it is still important to have communication optimizations that can achieve

better performance out of programs that are written without significant hand-optimizations. To exploit communication and computation overlap, today parallel programmers typically have to manually apply nonblocking communication primitives provided by the language, and explicitly insert synchronization calls for the nonblocking accesses. To reduce message startup overhead, programmers similarly have to manually pack and unpack the remote transfers. This manual scheduling and aggregation of remote accesses creates additional obstacles on the already difficult task of parallel programming. Manual optimizations are cumbersome to code, and programmers may have difficulty applying more sophisticated optimization techniques. Thus, our framework can significantly improve programmer productivity by freeing them from the details of communication management, while still offering performance comparable to that of manually optimized code.

We have implemented the system in the Berkeley UPC compiler and evaluated it on over a dozen benchmarks exhibiting different communication styles. Performance portability is a major goal for our compiler project, and we have tested the framework on three supercomputer clusters with different processor and network interconnect. Experimental results reveal that our framework offers comparable performance to aggressive manual optimization, and can achieve significant speedup compared to the fine-grained and blocking communication code that programmers find much easier to implement. Furthermore, the compiler has been used to develop a large UPC application dominated by fine-grained shared accesses, and our optimization framework is able to deliver a significant performance speedup without any programmer assistance.

Bibliography

[1] A. Aiken and D. Gay. Barrier inference. In *the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 342–354, 1998.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. *The Fortress Language Specification, version 1.0 beta*, 2006. http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

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

[5] S. Amarasinghe and M. 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.

[6] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993*

conference on Programming language design and implementation, pages 112–125, New York, NY, USA, 1993. ACM Press.

[7] 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.

[8] P. Banerjee, J. A. Chandy, M. Gupta, E. W. H. IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[9] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[10] C. Barton, C. Cascaval, G. Almasi, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral. Shared memory programming for large scale machines. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 108–117, New York, NY, USA, 2006. ACM Press.

[11] 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.

[12] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

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

[14] 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.

[15] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel Distrib. Technol.*, 2(3):37–47, 1994.

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

[17] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet. Technical Report LBNL-56495, Lawrence Berkeley National Lab, October 2004.

[18] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, 1993.

[19] D. Callahan, B. Chamberlain, and H. Zima. The Cascade high-productivity language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60, April 2004.

[20] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[21] 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.

[22] P. Charles, C. Donawa, K. Ebcioglu, et al. X10: An object-oriented approach to non-unifrom cluster computing. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, October 2005.

[23] CHARM++ project web page. Available at http://charm.cs.uiuc.edu.

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

[25] 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.

[26] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *16th International Workshop on Language and Compilers for Parallel Processing*, 2003.

[27] W.-Y. Chen. Building a source-to-source UPC-to-C translator. Master's thesis, University of California at Berkeley, 2004.

[28] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[29] F. C. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–286, 1997.

[30] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Computational Complexity*, pages 253–267, 1996.

[31] 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.

[32] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. A multi-platform Co-Array Fortran compiler. In *the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004)*, 2004.

[33] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, et al. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 36–47, 2005.

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

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

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

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

[38] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[39] A. Danalis, K. Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *Supercomputing 2005*, Nov 2005.

[40] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *18th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, 2005.

[41] J. Duell. Allocating, initializing, and referring to static user data in the Berkeley UPC compiler, 2002.

[42] J. Duell. Pthreads or processes: Which is better for implementing global address space languages? Master's thesis, University of California at Berkeley, 2007.

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

[44] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, November 2002.

[45] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *The 19th ACM International Conference on Supercomputing(ICS)*, June 2005.

[46] A. Faraj, X. Yuan, and D. K. Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *The 20th ACM International Conference on Supercomputing (ICS)*, June 2006.

[47] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *International Conference on Supercomputing*, pages 87–96, 1993.

[48] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 71. ACM Press, 1995.

[49] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, New York, NY, USA, 1995. ACM Press.

[50] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

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

[52] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[53] C.-H. Hsu, Y.-C. Chung, D.-L. Yang, and C.-R. Dow. A generalized processor mapping technique for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):743–757, 2001.

[54] C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks. In *VECPAR 2004*, June 2004.

[55] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the overlap. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[56] C. Iancu and E. Strohmaier. Optimizing communication overlap for high-speed networks. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2007.

[57] Intel compilers. http://www.intel.com/cd/software/products/asmona/eng/compilers/284132.htm.

[58] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[59] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005 (SC'05)*, November 2005.

[60] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, November 2005.

[61] 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.

[62] M. T. Kandemir, A. N. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing data and synchronization costs in one-way communication. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1232–1251, 2000.

[63] A. Karwande, X. Yuan, and D. Lowenthal. CCMPI: A compiled communication capable MPI prototype for ethernet switched clusters. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, June 2003.

[64] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, 1994.

[65] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *International Conference on Supercomputing*, pages 180–184, 1995.

[66] R. Kennedy, F. C. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 144–158, London, UK, 1998. Springer-Verlag.

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

[68] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.

[69] 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.

[70] 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.

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

[72] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *10th Annual International Static Analysis Symposium (SAS)*, June 2003.

[73] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *International Conference on Supercomputing*, pages 228–237, 1999.

[74] G. Liu and T. S. Abdelrahman. Computation-communication overlap on network-

of-workstation multiprocessors. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1998.

[75] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 1998. ACM Press.

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

[77] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite, 2005. http://www.hpcchallenge.org/pubs/index.html.

[78] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, 1991.

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

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

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

[82] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Workshop on Communication Architecture for Clusters (CAC02) of IPDPS'02*, 2002.

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

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

[85] OpenMP application program interface, 2005. http://www.openmp.org/drupal/mp-documents/spec25.pdf.

[86] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[87] PathScale compiler suite. http://pathscale.com/ekopath.html.

[88] POSIX standard. http://www.opengroup.org/onlinepubs/009695399/.

[89] 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.

[90] L. Prylli, B. Tourancheau, and R. Westrelin. Modeling of a high speed network to maximize throughput performance: the experience of BIP over Myrinet. In *Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.

[91] L. Prylli, B. Tourancheau, and R. Westrelin. The design for a high-performance MPI implementation on the Myrinet network. In *PVM/MPI*, pages 223–230, 1999.

[92] 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.

[93] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, 1996.

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

[95] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI collectives on clusters of large scale SMPs. In *Supercomputing 1999*, Nov 1999.

[96] J. Sorensen and S. Baden. A data driven model for tolerating communication delays. In *Proceedings of the 12th SIAM Conference on Parallel Processing for Scientific Computing*, 2006.

[97] 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.

[98] 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.

[99] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 18–25, New York, NY, USA, 1996. ACM Press.

[100] The UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.

[101] Top 500 Supercomputer Sites. Top 500 list for June 2007. http://www.top500.org.

[102] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *5th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 144–155, 1995.

[103] S. Vadhiyar, G. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Supercomputing 2000*, Nov 2000.

[104] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[105] A. Wakatani and M. Wolfe. Effectiveness of message strip-mining for regular and irregular communication. In *PDCS (Las Vegas)*, Oct 1994.

[106] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proceedings of PARLE'94 (Athen, Greece)*, Jul 1994.

[107] WHIRL intermediate language specification. http://open64.sourceforge.net.

[108] 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.

[109] 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.

[110] XL C Enterprise Edition for AIX. http://www-306.ibm.com/software/awdtools/caix/.

[111] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10:825–836, 1998.

[112] Z. Zhang and S. Seidel. Benchmark measurements of current UPC platforms. In *4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, April 2005.

[113] Y. Zhu and L. Hendren. Locality analysis for parallel C programs. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1997.

[114] 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.