University of Alberta

Library Release Form

Name of Author: Christopher Mark Barton

Title of Thesis: Improving Access to Shared Data in a Partitioned Global Address Space Programming Model

Degree: Doctor of Philosophy

Year this Degree Granted: 2009

_____

Christopher Mark Barton

Date: _____

I think there is a world market for maybe five computers.

– Thomas J. Watson, chairman of IBM, 1943.

University of Alberta

IMPROVING ACCESS TO SHARED DATA IN A PARTITIONED GLOBAL ADDRESS SPACE PROGRAMMING MODEL

by

Christopher Mark Barton

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**

in

Department of Computing Science

Edmonton, Alberta
Spring 2009

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Improving Access to Shared Data in a Partitioned Global Address Space Programming Model** submitted by Christopher Mark Barton in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

_____

José Nelson Amaral

_____

Vivek Sarkar

_____

Richard Sydora

_____

Jonathan Schaeffer

_____

Paul Lu

Date: _____

To my beautiful wife Jennifer,
whose continuing support
has made this work possible.

Abstract

Partitioned Global Address Space (PGAS) programming languages offer an attractive, high-productivity programming model for programming large-scale parallel machines. PGAS languages, such as Unified Parallel C (UPC), combine the simplicity of shared-memory programming with the efficiency of the message-passing paradigm. PGAS languages partition the application's address space into private, shared-local, and shared-remote memory. The latency of shared-remote accesses is typically much larger than that of local, private accesses, especially when the underlying hardware is a distributed-memory machine and remote accesses imply communication over a network.

To achieve good performance, an optimizing compiler must be able to handle two features commonly found in PGAS languages: shared data distribution and a parallel loop construct. When developing a parallel application, the programmer identifies data that is shared among threads and specifies how the shared data is distributed among the threads. This thesis introduces new static analyses that allow the compiler to distinguish between local shared data and remote shared data. The compiler then uses this information to reduce the time required to access shared data using three techniques. (i) When the compiler can prove that a shared data item is local to the accessing thread, accesses to the shared data are transformed into traditional memory accesses; (ii) When several remote shared-data accesses are performed and all remote shared-data is owned by the same thread, a single coalesced shared-data access can replace several individual shared-data accesses; (iii) When shared-data accesses require explicit communication to move shared data, the compiler can overlap the communication with other computation to hide the communication latency.

This thesis introduces a new locality analysis, describes the implementation of four locality-aware optimizations in a UPC production compiler, and presents a performance evaluation of these techniques. The results of this empirical evaluation indicate that the analysis and code transformations implemented in the compiler are crucial to obtain good performance and for scalability. In some cases the optimized benchmarks run as much as 650 times faster than the unoptimized versions. In addition, the performance of many of the

transformed UPC benchmarks is comparable with the performance of OpenMP and MPI implementations of the same benchmarks.

Acknowledgements

I would first like to thank my supervisor, José Nelson Amaral. He has offered me guidance, support and most of all patience. I am truly grateful for the knowledge he has shared with me and I look forward to working together in the future.

I would also like to thank Calin Cascaval, whom I have had the pleasure of working closely with for the last several years. His experiences and insights played a key role in the development of this work.

IBM's Center for Advanced Studies has provided an opportunity for me to work with an excellent group of software developers. I have taken away more from this experience then they could ever imagine. The TPO development team at the IBM Toronto Software Laboratory has provided an excellent environment for me to work in and explore my ideas. I stumbled many times along the way, and I must thank the entire team, specifically Raul Silvera and Roch Archembault for their patience and guidance as I found my way. In addition, I would like to thank Ettore Tiotto and George Almási for the extensive and fruitful discussions that contributed to this work.

Finally, my family. I can honestly say that this would not have been possible if I had not received your support throughout my quest to further my education. Thank you.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. This work was funded by the IBM Center for Advanced Studies.

Table of Contents

List of Tables

List of Figures

List of Symbols

$A_{i,j}$	A shared reference representing the shared-array access $A[i][j]$, 23
$\mathcal{AP}$	An address partition. Given a shared-array A, an address partition will contain $\mathcal{B}_A \times \mathcal{T}_{AP}$ consecutive elements of A, 21
$\mathcal{B}$	The blocking factor of a shared object, 19
$Block_T$	The block of shared-array elements within a block group that is owned by the executing thread T, 20
$BlockEnd_T$	The last element in the block group owned by the thread T, 20
$BlockStart_T$	The first element in the block group owned by the thread T, 20
$\mathcal{G}$	A group of blocks of shared-array elements such that each thread owns one block in the group, 19
$\|\mathcal{G}\|$	The size of the block group $\mathcal{G}$, 19
L_{nest}	A loop nest, 23
$\mathcal{T}$	The total number of threads, 19
$\mathcal{T}_{AP}$	Thread group size (*i.e.* the number of threads that map to an address partition), 20

List of Acronyms

Chapter 1

Introduction

With the advent of petascale computing, programming for large-scale machines is becoming evermore challenging. Traditional languages designed for uni-processors, such as C or Fortran[1], only allow the simplest kernels to scale to millions of threads of computation. When building solutions for real-life applications, understanding the problem and designing an algorithm that scales to a large number of processors is a challenge in itself. Thus, adequate programming tools are essential to increase programming productivity for scientific applications.

A long-standing issue in High Performance Computing (HPC) is the productivity of software development for high-end parallel machines. Partitioned Global Address Space (PGAS) languages are increasingly seen as a convenient way to enhance programmer productivity for HPC applications on large-scale machines. A programming language that is designed under a PGAS programming model facilitates the encoding of data partitioning information in the program. Closing the gap between the programming and the machine models should increase software productivity and result in the generation of more efficient code.

Unified Parallel C (UPC) is a PGAS extension of the C programming language that provides a few simple primitives to allow for parallelism. The programming model is Single Program Multiple Data (SPMD). The memory model allows for PGAS programming with each thread having access to a private section, a shared-local section and a shared-global section of memory. The programmer specifies the data that is shared among threads using the **shared** keyword. All data not explicitly marked as shared is considered private to each thread. Threads have exclusive, low latency, access to the private section of memory. Typ-

[1]The original version of FORTRAN, as well as FORTRAN77, was conventionally spelled in all-caps. This convention was changed with Fortran 90 and subsequent versions to only capitalize the first letter. The official language standard now refers to it as Fortran [1].

ically the latency to access the shared-local section is lower than the latency to access the shared-global section.

UPC provides programmers with two distinct advantages over alternative parallel programming languages. First, the programmer must specify what data is to be shared among threads and can describe an *affinity* between shared data and threads. Every shared data item has an affinity with one, and only one, thread. This affinity can be seen as a logical ownership of the data. The second advantage of UPC is that it will run on both shared-memory and distributed-memory architectures. These two advantages allow programmers to rapidly prototype a data-parallel program and determine its performance characteristics on a variety of hardware platforms.

The UPC specification defines an *object* as a "*region of data storage in the execution environment that can represent values*" and a *shared object* as "*an object allocated using a shared-qualified declarator or allocated by a library function defined to create shared objects*" [56]. The same definitions will be used in this document. A shared object can also be used to describe a single element of a shared array. For example, the element of a shared array A_s at position i, $A_s[i]$, can also be represented as a shared object O_s. Thus, $A_s[i]$ and O_s can be used interchangeably.

The IBM UPC Runtime System (RTS) provides a platform-independent interface that allows compiler optimizations to be applied independent of the machine-code generation. The RTS provides functions to get and set the value of each shared variable in a UPC program. The RTS maintains a Shared Variable Directory (SVD) internally that stores information about where to find specific shared objects. The SVD function is similar to the function of a cache directory. The SVD is a partitioned data structure that is designed to scale to a large number of threads while allowing for the efficient manipulation of shared data.

Every access (use or definition) of a shared variable is translated into a function call to get or set the value of the shared variable. These are *shared object accesses*. These function calls use the SVD to locate the underlying memory containing the shared variable. The location resolution consists of a series of pointer dereferences to determine the thread that owns the shared object, followed by an address translation to determine the location of the shared-object in the owning thread's address space. These pointer dereferences and the subsequent address translation are very costly in terms of performance.

1.1 Objectives

The primary goal of this research is to implement a compiler framework — analysis and code transformation — that improves the performance of accessing shared data in UPC programs. This goal will be achieved through optimizing local shared-object accesses, coalescing multiple shared-object accesses into a single access, and scheduling remote shared-object accesses to overlap communication and computation. The second goal is to reduce the overhead incurred during the execution of parallel loops.

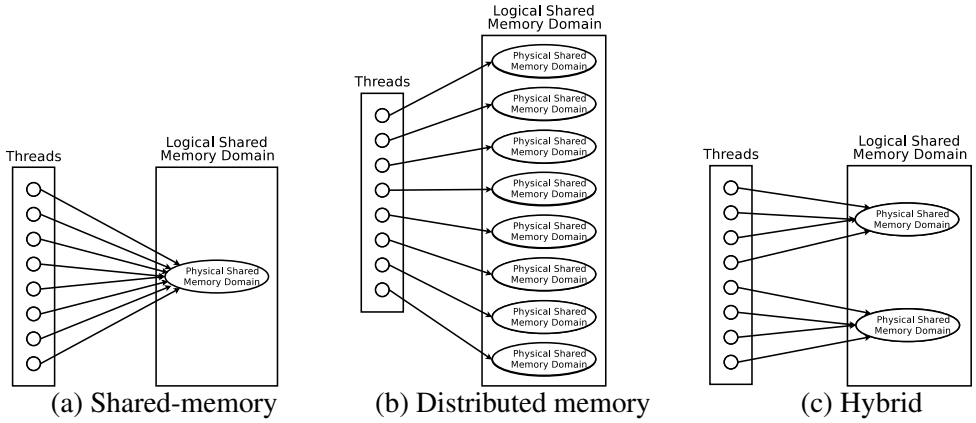


Figure 1.1: Target environment configurations

Figure 1.1 shows the three types of environments that this work will consider. Unless stated otherwise, a *thread* will refer to a UPC thread that is started by the UPC RTS prior to the execution of main and that is stopped prior to the UPC program exiting. In Figure 1.1(a), N threads map to one physical shared-memory domain, resulting in a single global address space (all-to-one mapping). This will be referred to as a shared-memory or Symmetric Multiprocessing (SMP) environment. In Figure 1.1(b), N threads map to N physical shared-memory domains (one-to-one mapping) resulting in each thread having a unique address space. This will be referred to as a distributed-memory environment. In Figure 1.1(c), N threads map to M physical shared-memory domains, $1 < M < N$ (many-to-one mapping), resulting in more than one thread sharing the same address space, but not all threads sharing the same address space. This is equivalent to a cluster of shared-memory parallel machines and will be referred to as a hybrid environment. The configuration in Figure 1.1(c) includes multi-processor nodes as well as multi-threaded single-processor nodes (*e.g.,* nodes with Simultaneous Multi-Threading (SMT) capabilities). For simplicity, we

assume that in Figure 1.1(c) each physical shared-memory domain is shared by the same number of threads. We will refer to the physical shared-memory domain for a thread as an *Address Partition*. Compiler options are used to specify the number of threads and the number of address partitions at compile time.

The thread that owns a shared object O_s will be represented as $Owner(O_s)$, where the subscript s indicates that the object is shared. Similarly, $Owner(A_s[i])$ represents the thread that owns the element i of the shared array A_s. A *local* shared access occurs when a thread T accesses a shared object O_s that is located in the address partition of T. A *remote* shared access occurs when O_s is not located in the same address partition as T.

In the IBM UPC compiler, every UPC shared variable has a unique *handle* that is used by the RTS to locate the object using the SVD. This handle is conceptually a *fat pointer* that contains several fields. A *shared-object access* is performed using functions defined in the RTS. Each access (i) determines the location of the shared object in memory using the SVD and (ii) sets or gets the value of the shared object, depending on the access type. The SVD determines the memory location of a shared-object in two distinct steps. First, the owning thread is determined by querying several data structures containing information about the layout of the shared object. Although these queries translate into pointer dereferences and thus are expensive, they allow the SVD to scale to a large number of threads. Once the owning thread has been determined, the SVD performs an address translation to determine the exact memory location in the owner's address space. This address translation is required by the UPC language specification, which states how shared-objects are laid out in memory.

A *local shared-object access* occurs when a thread T accesses a shared object O_s and both T and the owner of O_s map to the same address partition. A *remote shared-object access* describes a shared object O_s that is being accessed by a thread T that maps to a different address partition than the owner of O_s. In an all-to-one mapping, all shared-object accesses are local, while in a one-to-one mapping only shared-object accesses performed by the object's owner are local — all other accesses are remote. In a many-to-one mapping, some shared-object accesses are local while others are remote, based on the layout of the shared objects.

A *direct-pointer access* refers to directly accessing the shared object using its address. Direct-pointer accesses can only be performed by threads that map to the address partition containing the shared object. In other words, it is not possible to perform a *remote direct-pointer access*. A local shared-object access uses the SVD to determine the owner of the shared object and to locate the underlying memory location. A remote shared-object access

also uses the SVD to determine the owner of the shared object but in addition it requires a message to be sent to the owner to either request or update the value of the shared object. A direct-pointer access is a direct access to the underlying memory containing the shared object (a pointer dereference in C). Thus, direct-pointer accesses are much more efficient than shared-object accesses, but are only applicable in the address partition containing the shared object. Performing a shared-object access, while much slower than performing a direct-pointer access, is always legal.

Unless stated otherwise, all of the analysis and optimizations presented in this document focus on statically allocated shared arrays. The analysis and optimizations presented here can also be applied to dynamically allocated shared data (*e.g.,* pointers to shared data) however additional compile-time and runtime checks are required. This extension is discussed in more detail throughout the document. In addition, the current implementation requires the machine configuration (number of threads and address partitions) to be specified at compile time. Chapter 11 discusses how the implementation can be extended to perform the required analysis and optimizations when the machine configuration is not known until runtime.

1.1.1 Optimizing Shared-Object Accesses

When a thread reads a shared object O_s, a function call to the RTS is inserted to retrieve the value of O_s from memory. Similarly, when a thread writes a shared object, a function call to the RTS is inserted to assign a value to O_s. These function calls use the handle associated with O_s to locate the object in memory using the SVD. Using the SVD to locate a shared object in memory introduces a large overhead and significantly impacts the performance of UPC programs. When the underlying memory of the shared object is located in a thread T's address partition, T can access O_s directly. Thus, the fat pointer used to identify O_s in the SVD can be converted into a *thin pointer*, representing the memory location of the shared object. If the underlying memory location is not in T's address partition, function calls to the RTS must be used to access O_s. That is, conversion from fat to thin pointers is only valid when the accessing thread maps to the address partition containing the shared object.

The RTS allocates the underlying data for the shared object based on the shared object's affinity. If a thread T_1 owns a shared object O_s, the underlying storage for O_s is located in T_1's address partition. Thus, all shared-object accesses of O_s by T_1 can use thin pointers. A second thread, T_2, that maps to the same address partition can also use thin pointers to access O_s. Similarly, T_1 will be able to convert fat pointers to thin pointers for any shared

objects owned by T_2.

The conversion from fat pointer to thin pointer is facilitated by functions defined in the RTS. These functions return the base address of the shared object in local memory. This conversion requires several address translations by the SVD and thus has a cost similar to a shared-object access. Thus, converting fat pointers to thin pointers at runtime for every shared-object access will result in little performance improvement. However, if a shared-object is referenced many times, the base address is only computed once and all subsequent shared object accesses will use the thin pointer which will result in a performance improvement. In the case of shared arrays, the base address of the array is obtained from the RTS functions. Local elements of the shared array can then be accessed directly using the base address and an offset computed based on the index for each array element.

The analysis will focus on shared-object accesses located in **upc_forall** loops. Data-parallel programs tend to spend most of their computation time in loops. Therefore, optimizing loops should have the greatest impact on performance. The affinity between a shared object and a thread is determined at compile time using the affinity test specified in the **upc_forall** loop combined with the blocking factor of the shared object and the number of threads.

1.1.2 Shared-Object Access Coalescing

The coalescing of shared-object accesses allows several such accesses to be executed by a single call to the RTS. Assume N shared references, $r_{s_1}, r_{s_2}, \ldots r_{s_N}$ each accessing a different shared object $O_{s_1}, O_{s_2}, \ldots O_{s_N}$. Each shared-object access is translated into a function call to the RTS, resulting in N function calls. When all shared objects belong to the same shared array and have affinity with the same thread T, the N shared-object accesses can be coalesced into a single function call to the RTS. This Shared-Object Access Coalescing (SOAC) eliminates the overhead of $N-1$ function calls. When the shared-object accesses are remote reads, the coalesced RTS function will only send a single message to T requesting all N shared objects. Thus, N remote shared-object accesses that originally resulted in N individual messages now result in a single coalesced message to obtain N shared objects. When the shared-object accesses are remote writes, a single coalesced message is used to specify the N remote shared objects to be written.

To perform SOAC, the compiler must allocate and manage a temporary buffer to contain the coalesced shared-objects. Shared-objects that are read are placed into the temporary buffer by the owning thread. Subsequent uses of the shared-objects must be modified by the

compiler to refer to the temporary buffer. Similarly, write instructions to coalesced shared-objects must be modified by the compiler to write to a temporary buffer. The content of this buffer is sent to the owning thread to be updated once all coalesced writes are performed in the buffer.

1.1.3 Shared-Object Access Scheduling

When a thread accesses a remote shared-object, the access requires communication from the accessing thread to the owner of the shared-object. The primary goal of Shared-Object Access Scheduling (SOAS) is to overlap the latency of the required communication with other computation. Let O_s be a remote shared object. The retrieval of O_s must be completed before O_s is used. To avoid stalled CPU cycles due to long communication latencies, the function call to retrieve O_s should be scheduled such that O_s is available before the instructions that use O_s begin execution. While O_s is being retrieved, other computations can be performed. Thus, SOAS only applies to remote shared-object accesses. Performing this split-phase communication requires: data locality analysis by the compiler to distinguish between local and remote shared objects, a def-use analysis by the compiler to determine the placement of the RTS calls, and a mechanism in the RTS to (i) initiate the retrieval/update of a shared object, (ii) proceed while the retrieval/update is taking place, and (iii) verify that the retrieval/update has finished before proceeding past a certain point. This is similar to software prefetching, a well-known compiler optimization that attempts to reduce the number of data cache misses thereby improving performance [4, 6, 58].

Two cases must be treated separately: an access that *must* be remote and an access that *may* be remote. In the case of an access that must be remote, a call to the RTS initiates the data transfer and a subsequent call to the RTS blocks until the transfer is complete (possibly using the return value from the original call for identification). These calls should be inserted by the compiler. In the case of an access that may be remote, a runtime check is performed in the RTS to determine the location of the shared object. If the object is local, it can be returned immediately and the subsequent blocking call is not necessary. If the object is remote, a later blocking call is required. Analysis and placement of the initiation and blocking calls will be performed by the compiler.

1.1.4 Optimizing Parallel Loops in UPC

A **upc_forall** loop is a special version of a **for** loop that is used to distribute iterations of the loop among all threads. A fourth parameter is added to the loop that specifies an *affinity*

test. For a given thread T and a specific iteration of the loop L_i, if the affinity test is true then T executes L_i. The UPC front-end transforms a **upc_forall** loop into a traditional **for** loop that contains information about the affinity test. In many cases, the loop can be strip mined to remove the necessity of a branch inside the loop. The new loop nest created by strip mining will have two loops: the outer loop iterates over blocks of the shared variable owned by each thread while the inner loop iterates through each block. When the compiler cannot determine how to strip-mine the **upc_forall** loop, it must insert a branch statement into the loop body based on the affinity test. Every thread executes the branch statement on every iteration of the loop to determine if the body of the loop should be executed.

shared [B] **int** A[N];

```
upc_forall(i=0; i<N; i++; i) {
  A[i] = 0;
}
```

(a) **upc_forall** loop with integer affinity

```
for (i=MYTHREAD; i<N; i+= THREADS) {
  A[i] = 0;
}
```

(b) **for** loop after strip mining

```
upc_forall(i=0; i<N; i++; &A[i]) {
  A[i] = 0;
}
```

(c) **upc_forall** loop with pointer-to-shared affinity

```
for (i=MYTHREAD*B; i<N; i+=THREADS*B) {
  for (j=i; j<i+B; j++) {
    A[j]=0;
  }
}
```

(d) **for** loop after strip mining

Figure 1.2: **upc_forall** loops and the resulting strip-mined loop

Figure 1.2 provides two examples of common **upc_forall** loops and the resulting loops after strip-mining has been applied. Figure 1.2(a) shows a **upc_forall** loop with an integer affinity test. Note that the resulting strip-mined loop is a single nest because the inner loop would be a single iteration (iterate from 0 to 1) and thus can be simplified to the loop shown in Figure 1.2(b). Figure 1.2(c) shows a **upc_forall** loop with a pointer-to-shared affinity test based on a one-dimensional shared array A. Each thread is allocated B contiguous elements of A, where B is the blocking factor specified at compile time. The corresponding strip-mined for loop is shown in Figure 1.2(d).

1.2 Contributions

The thesis presented here is that the overhead of accessing shared data in a PGAS programming model can be significantly reduced using the optimizations described in this

dissertation. To support this thesis this dissertation makes the following contributions:

- Presents a new locality analysis that allows a compiler to identify the relative thread that owns shared references in a **upc_forall** loop.
- Describes four locality-aware optimizations that use the results from the locality analysis: (i) Shared-Object Access Privatization converts accesses to shared data that is local to the accessing thread into direct memory accesses, circumventing the overhead of the UPC runtime; (ii) Shared-Object Access Coalescing combines multiple shared-object accesses to the same remote thread into a single remote access, reducing the number of messages required to access remote data; (iii) Shared-Object Access Scheduling hides the cost of accessing remote shared data by overlapping the remote accesses with other computation; and (iv) Updating remote shared-object accesses reduces the number of accesses to remote data by specifying the operation to be performed by the owner of the remote data.
- Describes strip-mining optimizations that a compiler can use to reduce the overhead of executing parallel loop nests.
- Presents performance results that demonstrate significant benefits from performing locality-aware optimizations and parallel loop overhead reduction. These results include up to 650 times increase in the sustained memory bandwidth for the STREAM benchmark; a 60% reduction in the number of remote accesses for the Sobel benchmark; and a 2-time increase in the Millions of Floating Point Operations Per Second for the Random Access benchmark. The results also show that the performance of the optimized UPC benchmarks are comparable with corresponding OpenMP and MPI implementations.

1.3 Organization of this Document

The remainder of this document is organized as follows: Chapter 2 introduces the Unified Parallel C programming language. Chapter 3 defines commonly-used terms that are used throughout the document. The IBM UPC Compiler and Runtime System are described in Chapter 4. The locality analysis algorithm is presented in Chapter 5. Chapter 6 presents optimization algorithms that use the results of the locality analysis to optimize shared-object accesses. Chapter 7 presents techniques to optimize UPC parallel loops. A performance analysis of the locality optimizations and of the parallel loop optimization is performed

on several benchmarks and presented in Chapter 8. There is extensive work related to the material presented in this document. The related work is presented in Chapter 9, after all new contributions have been described. This order of presentation allows the related work to be contrasted to the new contributions. Chapters 10 and 11 present conclusions and future work. Appendix A presents compiler terminology that should be useful to readers that are not familiar with the compiler literature.

Chapter 2

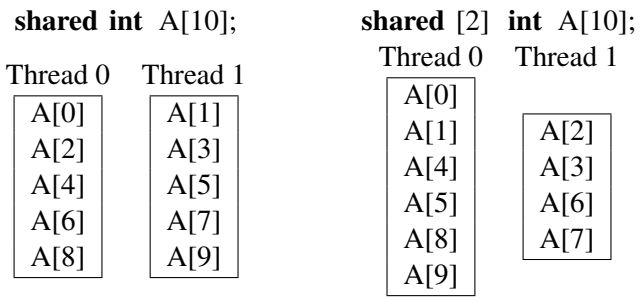
Unified Parallel C

Unified Parallel C (UPC) is an extension of the C programming language that is designed for parallel programs. UPC employs a Partitioned Global Address Space (PGAS) model, where the programmer can use a shared address-space programming model but the underlying architecture can be shared memory, distributed memory or a combination of both. The shared memory is partitioned, with a portion of it being local to each thread. The compiler and runtime system handle the movement of shared data between processors when necessary, based on the underlying architecture.

2.1 Shared Objects in UPC

The **shared** keyword is used in UPC to identify data that is to be shared among all threads. Every shared object has affinity with one, and only one, thread. The programmer uses a *blocking factor* to specify the *affinity* between shared objects and threads. If a shared object O_s has affinity with a thread T then T *owns* O_s. In an ideal UPC program, the majority of shared data accesses will be to shared data owned by the accessing thread. Such a data distribution reduces the amount of data movement between threads, thereby improving the performance of the program. When a thread T_1 uses a shared object that is owned by another thread T_2, and T_1 and T_2 map to different address partitions, the shared object has to be copied between T_1's address partition and T_2's address partition.

Shared scalars and structs are considered unique shared objects, while a shared array is a collection of shared objects. Thus, each shared-array element is a unique shared object; the terms shared-array element and shared object will be used interchangeably throughout the document. All shared scalars and structs have affinity with thread 0. Individual elements of a shared array are distributed among all threads based on the blocking factor. If no blocking factor is specified by the programmer, elements are distributed in a cyclic fashion, with the



(a) Default (cyclic) blocking factor (b) Blocking factor of 2

Figure 2.1: Shared-array declaration and data-affinity examples (2 threads)

first element owned by thread 0, the second element owned by thread 1, and so forth. If an infinite blocking factor is specified (denoted with empty square brackets: `[]`), the entire array has affinity with thread 0. If a blocking factor $\mathcal{B}$ is specified, $\mathcal{B}$ contiguous elements of the array are allocated to each thread (*i.e.* the first $\mathcal{B}$ elements are allocated to thread 0, the second $\mathcal{B}$ elements are allocated to thread 1 and so forth). Figure 2.1(a) shows the affinity of each array element given the default (cyclic) blocking factor while, Figure 2.1(b) uses a blocking factor of 2.

2.2 Shared and Private Pointers

UPC supports four types of pointer declarations:

1. **int** ∗pp;
2. **shared int** ∗ps;
3. **int** ∗**shared** sp;
4. **shared int** ∗**shared** ss;

Item 1 shows a private pointer pointing to private space (PP), which is a standard C pointer. Item 2 shows a private pointer pointing to shared space (PS). In this case, each thread has a private copy of ps that can be used to access shared objects. Item 3 shows a shared pointer pointing to private space (SP). This type of pointer is not encouraged because dereferencing it by a thread that does not own the private space may cause an access violation. To ensure that an access violation does not occur and that the correct data is obtained when this type of pointer is used, the programmer must be very careful when allocating memory and assigning the pointers. Item 4 shows a shared pointer pointing to

shared space (SS). Here, there is only one instance of ss with affinity to thread 0, but all threads have access to it.

In UPC, the blocking factor is considered part of the type system. Thus, two pointers to shared data, **shared int** *ps1 and **shared** [5] **int** *ps2 have different types. This is important when determining the aliasing in UPC programs – two pointers to shared data with different blocking factors are not aliased to each other.

2.3 UPC Built-in Functions

UPC provides three predefined identifiers to the programmer. **MYTHREAD** is an integer value that evaluates to a unique thread index for each thread at runtime. **THREADS** is an integer value that specifies the number of threads created to run the program. It is possible to specify the number of threads at compile time. Providing this information will allow the compiler to perform more aggressive optimizations. The number of threads can be defined at runtime (prior to program execution) using an environment variable. If the number of threads at runtime is different than the number of threads specified at compile time, the program terminates with an error message.

The **upc_threadof** function takes a pointer to a shared object as an argument and returns the thread index that has affinity to the shared object. This function is used to determine the owner of a shared object.

The remaining built-in functions are outside the scope of this document. A description of them can be found in the UPC Specifications [56].

2.4 Parallel Loops

The **upc_forall** statement distributes iterations of the loop among all threads. Instead of each thread executing all iterations of the loop, an iteration is conditionally executed by a thread based on an *affinity test*.

The affinity test is specified by the programmer using a fourth parameter in the **upc_forall** loop declaration. This parameter must contain either an integer, a pointer-to-shared, or the **continue** keyword. If no parameter is specified, the **continue** keyword is assumed. Table 2.1 illustrates the different types of affinity statements that can be used and the corresponding **for** loops. When the affinity parameter is an integer, an iteration i will be executed by a thread j if and only if the affinity parameter modulo the number of threads is equal to j. When a shared address is used, an iteration i of the loop will be executed by thread j if

```
upc_forall (i=0; i < THREADS; i++; i) {
  A[i] = MYTHREAD;
}
```

```
upc_forall (i=0; i < N; i++; &A[i]) {
  A[i] = MYTHREAD;
}
```

```
upc_forall (i=0; i < N; i++; continue) {
  A[i] = MYTHREAD;
}
```

(a) **upc_forall** loop

```
for (i=0; i < THREADS; i++) {
  if ( (i%THREADS) == MYTHREAD )
    A[i] = MYTHREAD;
}
```

```
for (i=0; i < THREADS; i++) {
  if (upc_threadof(&A[i]) == MYTHREAD)
    A[i] = MYTHREAD;
}
```

```
for (i=0; i < N; i++) {
  A[i] = MYTHREAD;
}
```

(b) Corresponding **for** loop

Table 2.1: Example **upc_forall** loops and their associated **for** loops

and only if the shared address has affinity with j. In this case, it is common to use the loop induction variable as an index into a shared array. When the **continue** keyword is used, or no statement is specified, the loop body is executed by all threads. Note that in this example, since A is a shared array, the use of the **continue** keyword causes a data race as seen in the third loop of Table 2.1(b).

2.5 Synchronization

Synchronization can be performed in several different ways. The **upc_barrier** statement denotes a synchronization point. A thread cannot begin executing the next statement after the **upc_barrier** until all threads have reached the **upc_barrier** statement.

Split-phase barriers are implemented using an **upc_notify** followed by an **upc_wait** statement. The **upc_notify** statement indicates when a thread T reaches a synchronization point. T continues executing instructions located between the **upc_notify** and **upc_wait** statements. T must halt execution when it reaches the **upc_wait** statement until all threads have executed the **upc_notify** statement. When a **upc_notify** statement has been encountered, the next synchronization statement to be executed must be a **upc_wait** [56].

The **upc_fence** statement ensures the completion of all shared-object accesses located lexically *before* the fence prior to the execution of any shared-object access located lexically *after* the fence.

2.6 UPC Memory Model

The memory model in UPC has two distinct access modes: *strict* and *relaxed*. In the relaxed memory model, shared-object accesses can occur in any order as long as data dependencies are preserved and all shared-object accesses have completed at the end of a synchronization point. Similarly, no shared-memory accesses can begin until all shared-memory accesses prior to a synchronization point have completed and are visible to all threads. This memory model is very flexible in that it provides the compiler with the most opportunities to reorder shared-object accesses. As long as traditional data dependencies are preserved, the compiler is free to move shared object accesses anywhere between synchronization points.

The strict memory model forces sequential consistency. All shared-memory accesses must complete in the specified order before execution continues. A strict shared-memory reference can also act like a fence in that any relaxed shared-memory access that occur before it cannot be moved after it. Furthermore, any relaxed shared memory accesses that occur before the strict access must complete before execution can proceed. In this sense, strict accesses can be used to perform memory synchronization. This model is very restrictive in terms of the optimizations that can be performed by the compiler.

The programmer has the ability to specify an access mode to a specific shared variable, to a region of code or to the entire program. That is, the programmer can declare a program to run in relaxed mode, but define specific shared variables, specific statements or regions of code that must adhere to the strict access mode. Accesses to these strict variables are identical to a **upc_fence** instruction: all shared accesses before the strict access must complete and no accesses to subsequent shared variables can be performed until after the strict access has completed. Similarly, shared variable accesses cannot be moved across a strict shared variable access. Unless otherwise stated, we will assume a relaxed memory model throughout this document.

2.7 Collectives

The UPC specification provides many collective operations (*e.g.,* broadcast, scatter, gather, exchange, permute, reduction, and sort) that can be used to simplify common operations in parallel programs [55]. The collectives must be executed by every UPC thread. The collective operations allow the programmer to specify the type of synchronization for both the beginning of the collective and the end of the collective using one of three modes.

1. *NO*: the collective may read or write as soon as a thread enters the collective function.

2. *MY*: the collective may read or write data only to threads that have entered the collective function.

3. *ALL*: the collective may read or write data only after all threads have entered the collective function.

The synchronization mode is specified in the flags parameter passed to the function call. Thus, the compiler has access to the synchronization mode and is typically able to determine the type of synchronization specified by the programmer.

2.8 Limitations of UPC

We have implemented several parallel algorithms — stencil computation and linear algebra operations such as matrix-vector and Cholesky factorization — in the UPC programming language. During this effort we identified several issues with the current language definition, such as:

- UPC has rudimentary support for data distributions. In UPC, shared arrays can only be distributed in a block cyclic fashion.
- UPC has a flat threading model. This model does not support subsets of threads.
- UPC focuses primarily on data parallel applications. While task-parallelism in UPC is possible, UPC task-parallel programs are not as elegant as the same programs in other programming languages.
- The definition of collectives in UPC has the following shortcomings: collectives cannot operate on subsets of threads; shared data are not allowed as a target for collective operations; a thread cannot concurrently participate in multiple collectives.

Section 4.2 presents an extension to the UPC language that we proposed to enhance data distribution. This change to UPC extends the way shared data can be distributed among threads and is extremely useful for certain types of scientific applications. In addition, this extension allows UPC programs to interface with existing libraries that require a specific data layout.

2.9 Chapter Summary

This chapter has presented a brief overview of the UPC programming language including (i) how shared objects are declared and laid out; (ii) the different types of pointers that are

available in UPC; (iii) how work is distributed using parallel loops; (iv) the UPC memory model; (v) how synchronization is performed; (vi) collective operations that can be used to simplify programming. The following chapter presents UPC-specific definitions and terminology that will be used throughout the remainder of the document.

Chapter 3

Definitions

This chapter introduces terminology that will be used throughout the document. It begins with a discussion of the the global view (logical layout) and local view (physical layout) of UPC shared arrays and defines terms used to describe the different parts of the shared arrays for the different views. These terms are used when describing the locality analysis and subsequent UPC optimizations. Definitions of general compiler terminology used in this document can be found in Appendix A.

3.1 UPC Shared Array Layout

Shared arrays in UPC can be viewed in two separate ways: (i) the *global* or *logical* view; (ii) the *local* or *physical* view. The global view encompasses the entire array. The physical view shows how the blocks of the shared array are allocated to each thread.

3.1.1 Global Shared Array Layout

The global view of a shared array is concerned with how elements of the array are logically distributed among each of the threads. In the global view, elements of a shared array A_s are grouped together in blocks of size $\mathcal{B}$ and divided among $\mathcal{T}$ threads.

When discussing the global view of a shared array, the following terminology is used.

Blocking factor The *blocking factor*, $\mathcal{B}$ for a shared object O_s describes the number of consecutive elements that have affinity to each thread.

Block group A *block group*, $\mathcal{G}$ is a group of blocks of elements for the shared object O_s such that each thread owns one block in the group. The size of a block group, $|\mathcal{G}|$, is computed by multiplying the blocking factor of the shared object by the number of threads, as seen in Equation 3.1.

$$|\mathcal{G}| = \mathcal{B} * \mathcal{T} \tag{3.1}$$

Given a linearized offset, pos, for a shared object, the block group index where pos is located is computed using equation 3.2. The linearized offset representing the first position of a block group is computed using equation 3.3. Note that equation 3.2 uses integer division and thus the division cannot be simplified when substituting equation 3.2 into equation 3.3.

$$Index(\mathcal{G}) = \left\lfloor \frac{pos}{|\mathcal{G}|} \right\rfloor \tag{3.2}$$

$$\mathcal{G}^{start} = Index(\mathcal{G}) * |\mathcal{G}| \tag{3.3}$$

Within a block group, the block of shared array elements owned by the executing thread T is specified as $Block_T$.

$$BlockStart_T = \mathcal{B} * \texttt{MYTHREAD} \tag{3.4}$$

$$BlockEnd_T = BlockStart_T + \mathcal{B} - 1 \tag{3.5}$$

The first and last indices for $Block_T$ are computed using equations 3.4 and 3.5 respectively. It follows from $BlockStart_T$ and $BlockEnd_T$ that the size of $Block_T$ is $\mathcal{B}$.

Thread group A *thread group* is a group of threads that maps to the same address partition. The thread group size, $\mathcal{T}_{AP}$, is the number of threads that map to an address partition. The subscript AP indicates that we are referring to the subset of threads that map to the same address partition. In a shared-memory environment, all threads belong to the same thread group and thus the thread group size is the total number of threads ($\mathcal{T}_{AP} = \mathcal{T}$). In a distributed memory environment, each thread maps to a different address partition ($\mathcal{T}_{AP} = 1$). In a hybrid environment, the thread group size is determined by the number of threads and the number of nodes. For simplicity, we assume the same thread group size for all address partitions (*i.e.* $\mathcal{T} \,\%\, \mathcal{T}_{AP} = 0$). We also assume a straightforward sequential mapping of threads to address partitions: given $\mathcal{T}$ threads, N nodes and a thread group size of $\mathcal{T}_{AP}$ ($\mathcal{T}_{AP} \leq \mathcal{T}$, $\frac{\mathcal{T}}{N} = \mathcal{T}_{AP}$) threads 0 to $\mathcal{T}_{AP} - 1$ are mapped to node 0, threads $\mathcal{T}_{AP}$ to $\mathcal{T}_{AP} + \mathcal{T}_{AP} - 1$ are mapped to node 1, *etc*.

Consider a shared array A_s with blocking factor $\mathcal{B}$ and Z elements, declared in UPC as:

shared [B] **int** A[Z];

Figure 3.1 shows the logical view of A_s, when run in an execution environment containing $\mathcal{T}$ threads and N address partitions. Assume that each thread is allocated K contiguous blocks of A_s (*i.e.* $Z > \mathcal{B} * \mathcal{T}$ and $\frac{Z}{\mathcal{B}} = K$).

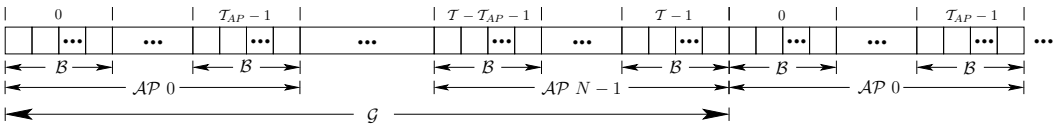


Figure 3.1: Global view of a UPC shared array

Figure 3.1 presents the elements of the array A_s grouped into segments of $\mathcal{B}$ elements (first division below the array). The thread that owns each segment is indicated above the array. The figure also shows the address partition that each thread group belongs to (second division below the array). Finally, the figure indicates where the first block group ends (third division below the array).

Given $\mathcal{T}_{AP}$ threads per address partition, $\mathcal{T}_{AP} \leq \mathcal{T}$, each address partition contains $\mathcal{AP} = \mathcal{B} \times \mathcal{T}_{AP}$ elements of A_s. The representation in Figure 3.1 is general. In a hybrid environment with multiple threads per address partition (*i.e.* many-to-one mapping) $1 < \mathcal{T}_{AP} < \mathcal{T}$. In a shared-memory environment $\mathcal{T}_{AP} = \mathcal{T}$. When $\mathcal{T}_{AP} = 1$ there is a single thread operating in each address partition (*i.e.* one-to-one mapping).

3.1.2 Local Shared Array Layout

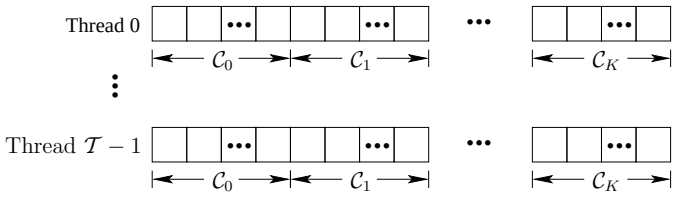


Figure 3.2: Physical view of a UPC shared array

The local view of a shared array is concerned with the layout of the shared array on each thread. Within the local storage of each thread a block is kept as a collection of contiguous memory locations called a *course*. Figure 3.2 shows the corresponding physical view of

the shared array in Figure 3.1. When discussing the logical view of a shared array, the following terminology is used.

threadof The threadof of a specific array location is the thread ID that owns the specified array element. For shared arrays, the threadof term is only used in reference to a specific array element (*i.e.* it does not make sense to discuss the threadof of an entire shared array).

courseof The courseof of a specific array element is the local block number (or course index) in which the element resides. The term courseof is only used in reference to a specific array element of a shared array.

phaseof The phaseof of a specific array element is the offset within the course where the element resides. The term phaseof is only used in reference to a specific array element of a shared array.

This physical layout of elements of the shared array is required by the UPC memory model. This layout allows the programmer to cast a local shared array element to a local pointer and then walk through all of the local elements of the shared array contiguously. The downside to this layout is the indexing computations that need to be generated to convert a shared array element A_s (global view) into a memory location (physical view). This indexing is discussed in more detail in Section 4.3.4.

3.2 UPC Terminology

Shared-object access A shared-object access is an access (read or write) to a variable marked as *shared* in a UPC program. The object can be any type (scalar, struct, array) and thus the access may or may not have an index or offset associated with it.

A shared-object access is performed using functions defined in the RTS. Each access (i) determines the location of the shared object in memory using the SVD and (ii) sets or gets the value of the shared object, depending on the access type. The SVD determines the memory location by performing several address translations using a unique *handle* or *fat pointer* that identifies the object. A single handle is used to identify all shared objects from the same shared array. Shared-object accesses are expensive to perform because of the required address translations.

Owning thread Given a shared object O_s, the *owning thread* or *owner thread* is the thread that O_s has affinity with.

Accessing thread Given an access to a shared object, the thread performing the access is the *accessing thread*.

Local shared-object access A local shared-object access occurs when a thread T accesses a shared object O_s that is located in the address partition of T. In an all-to-one mapping, all shared-object accesses will be local, while in a one-to-one mapping only shared-object accesses performed by the owner thread will be local. In a many-to-one mapping, some shared object accesses will be local based on the layout of the shared objects.

Remote shared-object access A remote shared-object access occurs when a thread T accesses a shared object O_s that is not located in the address partition of T. In an all-to-one mapping there are no remote shared-object accesses. In a one-to-one mapping, any shared-object access performed by a thread that does not own the shared-object will be remote. In a many-to-one mapping, the shared-object accesses may be remote, depending on the layout of the shared objects.

Direct pointer access A direct pointer access refers to directly accessing a shared object using its address. Direct pointer accesses can only be performed for local shared-object accesses. In other words, it is not possible to perform a *remote direct pointer access*.

Thread The UPC specification defines a *thread* as "*an instance of execution initiated by the execution environment at program startup*"[56]. Unless otherwise stated, a thread will refer to a UPC thread.

Node A node corresponds to a unique address partition in a given execution environment. A shared-memory environment will contain a single node while distributed and hybrid environments will contain multiple nodes.

Shared reference The compiler uses a *reference* to represent indirect accesses to memory (access through a pointer, array or a structure). A reference contains information about the base symbol being accessed, the offset (also referred to as the index) and the data type and length. A *shared reference* is a reference where the base symbol has been marked as shared. A shared reference for the shared array access A[i][j] is represented as $A_{i,j}$.

Iteration Space A loop nest L_{nest} can be described by its *iteration space*. The iteration space contains one point for every iteration of the loops in L_{nest}, where a point is a unique combination of loop induction variables.

Iteration Vector An *iteration vector* is used to represent points in the iteration space. Given a loop nest L_{nest} containing m loops, the iteration vector contains m integers representing the iteration number for each loop in L_{nest}, ordered by nesting level [4, 58]. The iteration number for a loop corresponds to a unique value for the loop induction variable, based on the loop bounds and increment. When L_{nest} contains a **upc_forall** loop it is a *parallel loop nest* and each iteration vector is executed by only one thread. We assume the **upc_forall** loop has either an integer or a pointer-to-shared affinity test; **upc_forall** loops that use the **continue** affinity test are not considered as parallel loops.

To determine the physical location of each array element accessed by a loop, the iteration vector is converted into a linearized offset from the base of the array. This is done by multiplying the iteration vector by a column vector representing the dimension sizes. Given an N-dimensional array where x_i represents the number of elements in dimension i, the dimension size vector dim_{size} is computed as follows:

$$dim_{size}[i] = \begin{cases} \prod_{j=i+1}^{N} x_j & 1 \leq i < N \\ 1 & i = N \end{cases}$$

0	3	6	9
12	15	18	21
24	27	30	33
36	39	42	45
48	51	54	57

$A[][][0]$

1	4	7	10
13	16	19	22
25	28	31	34
37	40	43	46
49	52	55	58

$A[][][1]$

2	5	8	11
14	17	20	23
26	29	32	35
38	41	44	47
50	53	56	59

$A[][][2]$

Figure 3.3: Layout of 3-dimensional array: **int** A [5][4][3]

Figure 3.3 shows an example of a 3-dimensional array with dimension sizes of 5, 4 and 3. The value in each element indicates the element's offset from the base of the array. The dimension size vector for this array is:

$$dim_{size} = \begin{bmatrix} 12 \\ 3 \\ 1 \end{bmatrix}$$

The iteration vector for element A[0][1][2] is [0 1 2], which is multiplied by dim_{size} to obtain a linearized offset of 5. Similarly, the iteration vector for element A[3][3][2] is [3 3 2] which represents a linearized offset of 47.

3.3 Chapter Summary

This chapter provides an overview of terminology used to describe UPC shared arrays. A shared array has two views: (i) a logical or global view; and (ii) a physical or local view. Typically the programmer will consider the logical view of the shared array, however the compiler must use the physical view when performing locality optimizations. The definitions presented here will be used extensively when describing the locality analysis and the locality optimizations in Chapters 5 and 6.

Chapter 4

The IBM UPC Compiler and Runtime System

The IBM XL UPC compiler, *xlupc*, is a full implementation of the UPC language version 1.2 supporting IBM pSeries® systems running the AIX® or Linux operating systems. The *xlupc* compiler builds on the well-known strength of the IBM XL compiler family and on a scalable runtime system designed and implemented as a collaboration between IBM's compiler technology group in Toronto and IBM Research.

The *xlupc* compiler is a full compiler providing extensive diagnostics and compile-time syntax checking of UPC constructs. As opposed to a source-to-source translator, a full compiler offers the advantage that the language semantics can be carried on from parsing through different levels of optimization and all the way to the code generator. Figure 4.1 shows the components of *xlupc* and UPC Runtime System.

The compiler Front-End (FE) parses the UPC source code and generates a three-address intermediate representation called *Wcode*. Wcode is a stack-based intermediate language that provides a well-established interface between IBM compiler components. The Toronto Portable Optimizer (TPO) performs high-level optimizations using the Wcode generated by the FE. The compiler back-end (TOBEY) generates object code based on the Wcode input from TPO. TOBEY performs various low-level optimizations including software pipelining and register allocation based on the optimization level. The UPC Runtime System (RTS) contains data structures and functions that are used during the runtime execution of a UPC program, similar to GASNet [13]. Of the components seen in Figure 4.1, the FE and TPO have been modified to handle UPC code while TOBEY remains unchanged. The UPC Runtime System was created specifically for the UPC compiler.

The FE transforms all UPC-specific source code into traditional Wcode with additional information. For shared symbols, this information includes the type (shared array, private

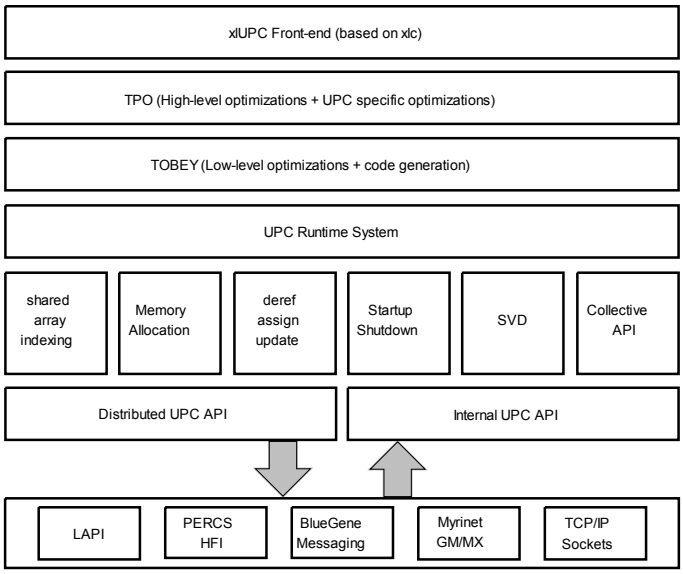


Figure 4.1: Components used in IBM's XL UPC compiler and runtime system

pointer to shared, shared-pointer to shared), the blocking factors (if applicable) and the access semantics to use (strict or relaxed). For **upc_forall** loops, the affinity test is specified. General information including the number of threads and number of nodes (if specified at compile time) is also included in the Wcode.

When compiling UPC programs TPO performs two different functions: *transformations* and *optimizations*. When performing transformations, TPO converts all shared-object accesses (reads and writes) into corresponding function calls defined in the RTS. These function calls are used to access the shared-objects using the SVD. **Upc_forall** loops are transformed into traditional C **for** loops with a branch guarding the execution of the loop body. When performing optimizations, UPC-specific optimizations are performed in addition to a subset of traditional optimizations. When compiling with optimizations, all optimizations are performed before the transformations and any UPC-specific code that is not optimized is converted using the transformations. When compiling without optimizations, the transformations are run to ensure the correct execution of the program.

The output of TPO is standard Wcode — TOBEY has not been modified to deal with UPC code. All UPC-specific portions of the program are either optimized or transformed by TPO. Likewise, the system linker/loader also has not been modified.

4.1 Overview of TPO

Toronto Portable Optimizer (TPO) originated as an interprocedural optimizer for RS/6000 machines. It has evolved since its conception into a compiler component that analyzes and transforms programs at both the procedure (intraprocedural) and whole program (interprocedural) level. The purpose of TPO is to reduce the execution time and memory requirements of the generated code. TPO is both machine- and source-language independent through the use of Wcode.

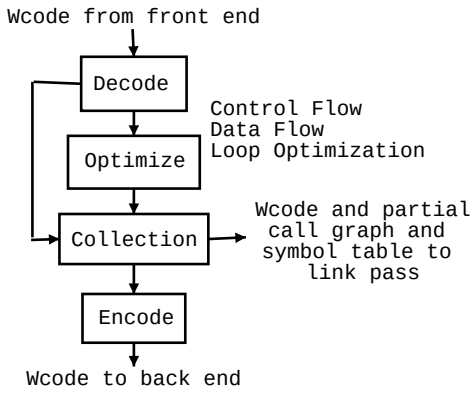


Figure 4.2: High-level control flow through TPO

TPO performs intraprocedural analysis on individual files at compile time. Intraprocedural and interprocedural analysis is performed on an entire program at link time. Figure 4.2 shows the high-level flow of control through TPO. At compile time, input into TPO is provided from the compiler front end. At link time, input into TPO is provided from object files and libraries, some of which may have been compiled by TPO. Object files compiled by TPO are supplemented by Wcode and a partial call graph and symbol table.

The three main groups of optimizations performed in TPO are control flow optimizations, data flow optimizations, and loop optimizations. While these optimizations can be performed at both compile time (intraprocedural scope) and link time (interprocedural scope) the UPC-specific optimizations are only applied at compile time. Unless stated otherwise, all discussions in this document pertain strictly to the intraprocedural analysis steps performed by TPO.

4.1.1 Loop Optimizations in TPO

The current version of TPO performs a subset of traditional loop optimizations when handling UPC code. In addition to the traditional loop optimizations, the locality analysis,

locality optimizations and the parallel loop-nest optimizations described in Chapters 5, 6, and 7 have been added to the loop optimization framework.

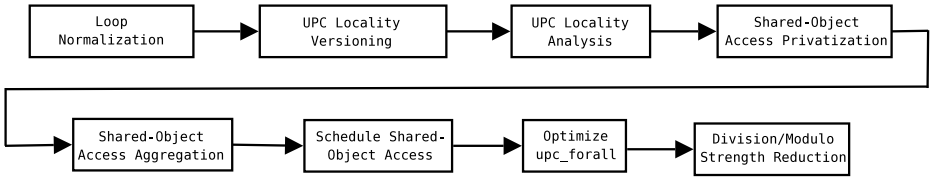


Figure 4.3: Loop optimizations in TPO

The execution sequence of the major loop optimizations performed in TPO is seen in Figure 4.3. The following sections discuss the traditional loop optimizations, providing examples to illustrate the code transformations that they perform.

Loop Normalization

Loop normalization modifies a countable loop to start at a specific lower bound and iterate, by increments of 1, to a specific upper bound. In TPO, normalized loops have a lower bound of 0. The upper bound is computed based on the original lower bound, upper bound, and increment of the loop to ensure that the loop iterates the correct number of times. Uses of the induction variable in the loop body are modified to maintain consistency with the original loop. Figure 4.4(a) shows an example of an un-normalized loop. The normalized version of the loop is seen in Figure 4.4(b).

```
1 for (i=10; i < n; i++) {
2   A[i] = A[i+2] * 2;
3   B[i+1] = A[i-1] + 3;
4 }
```

(a) Un-normalized loop

```
1 for (i=0; i < (n-10); i++) {
2   A[i+10] = A[i+12] * 2;
3   B[i+11] = A[i+9] + 3;
4 }
```

(b) Loop after normalization

Figure 4.4: Loop normalization example

Whenever possible, the compiler creates *bump normalized* loops. A loop is bump normalized if its first iteration is 0 and the induction variable is incremented by 1 in every iteration of the loop. Loops whose induction variables increment by 1 every iteration are described as having a *stride* of 1. A loop that is not countable cannot be bump normalized.

UPC Locality Versioning

UPC locality versioning operates on parallel loop nests that contain pointers to shared-objects. It creates two copies of the original parallel loop nest, guarded by a check to determine if the shared-object is a candidate for locality analysis. This check creates two control flow paths, the *true* path and the *false* path. The shared references in the true control flow path (*i.e.* the shared-object is a candidate for locality analysis) are added to a list and considered by the subsequent locality analysis. The shared references in the false control flow path are not added to a list and thus will not be considered by locality analysis.

```
1  int foo(shared int *A) {
2    int i;
3    upc_forall (i=0; i<N; i++; &A[i]) {
4      A[i] = MYTHREAD;
5    }
6    return 0;
7  }
```

(a) Original **upc_forall** loop

```
1  int foo(shared int *A) {
2    int i;
3    if (_is_shared_array(A) &&
4        upc_threadof(A)==0 &&
5        upc_phaseof(A)==0 ) {
6      upc_forall (i=0; i<N; i++; &A[i]) {
7        /* A is a candidate for
8             locality analysis    */
9        A[i] = MYTHREAD;
10     }
11   }
12   else {
13     upc_forall (i=0; i<N; i++; &A[i]) {
14       /* A is NOT a candidate
15          for locality analysis */
16       A[i] = MYTHREAD;
17     }
18   }
19   return 0;
20 }
```

(b) After UPC locality versioning

Figure 4.5: UPC locality versioning example

Figure 4.5 shows an example **upc_forall** loop before and after locality versioning. For the loop to be a candidate for locality analysis, the compiler must be able to prove that the shared reference A points to a shared array and begins at the beginning of a block group. The calls to **upc_threadof** and **upc_phaseof** return the thread and phase of the first element of A. If either of these values are non-zero, then the first element of A is not located at the start of a block group. If A does not point to a shared array (*i.e.* it points to a scalar), it is also not a candidate for locality analysis. The shared reference on line 9 of Figure 4.5(b) will be added to a list and considered for locality analysis while the shared reference on line 16 will not be considered.

UPC locality versioning was implemented by Ettore Tiotto at IBM and thus a detailed discussion falls outside the scope of this document.

Division/Modulo Strength Reduction

The division and modulo strength reduction optimization performs strength reduction optimizations on division and modulo operations that are created by the UPC optimizations. More information about strength reduction can be found in Appendix A.

4.1.2 Internal Representations in TPO

During the optimization pass, TPO decodes Wcode and builds several internal representations used during the optimization phase. A Control Flow Graph (CFG), Data Flow Graph (DFG), and Data Dependence Graph (DDG) are among the representations available throughout the optimization phase. The additional information passed by the FE about shared symbols is accessible from all of these representations. A data structure containing information specifically about loops is maintained separately from the CFG and DDG. A *loop descriptor* contains information about each loop, including (i) the lower and upper bounds (in numerical form if known at compile time or symbolic form otherwise); (ii) the induction variable used in the loop; and (iii) a list of references that occur in the loop. For references corresponding to shared pointers and shared arrays, information about the blocking factor, passed through the Wcode, can be accessed. Every procedure has a loop table that contains the loop descriptors for each loop found in the procedure.

TPO also contains a lexicographical list of statements for each procedure. A statement is a collection of one or more *expressions*, stored in a tree representation. Every expression contains an instruction and (when applicable) a symbol index.

For UPC-specific programs, TPO also maintains several data structures that contain information specific to shared symbols. These structures include an associative map that associates the shared symbol with the handle used by the RTS to identify the shared object. This map contains typical information about the shared object (such as data type and length), as well as UPC-specific information including the blocking factor and access type (strict or relaxed).

Loop Structure in TPO

The structure of loops in TPO corresponds to the general description of regions in the definitions section. A loop in the source code is transformed into five distinct parts in TPO, each having its own purpose.

Figure 4.6 gives a general overview of the parts of a loop structure in TPO. The *guard block* is the block immediately preceding entry into a loop. The *guard branch* protects

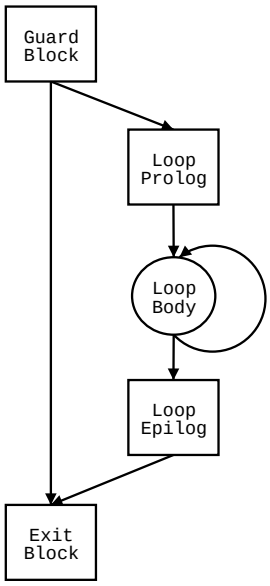


Figure 4.6: Loop structure in TPO

the execution of the loop and is the last statement in the guard block. If the guard branch evaluates to true, a jump instruction is executed and control jumps directly to the exit block, skipping the loop entirely. If the guard branch evaluates to false, all blocks representing the loop are executed at least once.

The *loop prolog* is located immediately below the guard branch of the loop, before the loop body. Since it is not included in the loop body, only instructions that are invariant to the loop are located in the loop prolog. The loop prolog normally contains the initialization of the induction variables used in the loop body, as well as any loop invariant code that has been moved out of the loop by the *loop invariant code motion* optimization.

The *loop body* contains all of the loop variant statements that were in the original loop plus any additional statements that were moved into the loop body by optimizations.[1] Note, in Figure 4.6, that the loop body can contain an arbitrary number of basic blocks with arbitrary control flow between them. The last statement of the last basic block in the loop body is called the *Latch Branch*. It is a test to determine if the current value of the induction variable is within the bounds set by the original loop. If it is, the latch branch is taken, resulting in the execution of another iteration of the loop. If the value is not within the bounds, the latch branch is not taken and control flows to the loop epilog.

The *loop epilog* block contains statements to be executed after the loop body has executed but only if the loop body was executed. Like the loop prolog, the loop epilog only

[1] Aggressive copy propagation is one example of an optimization that can move statements into a loop body.

contains statements that are invariant to the loop body. For instance, any statements that were identified as loop invariant but could not be put into the loop prolog due to data dependencies would be placed in the loop epilog. It is possible that the loop epilog remains empty in which case it is removed by a later optimization.

The beginning of the *exit block* contains the target of the guard branch. This is the first block executed if the guard condition protecting the loop evaluates to true and the jump is taken.

Figure 4.7 shows an example loop in C source code (a), and the loop representation within TPO (b). This representation ensures that no unnecessary instructions are executed if the loop body does not execute at least one iteration.

```
1 for (i=k; i < j; i++) {
2   A[i] = A[i] + 2;
3   B[i] = B[i-1] * 3;
4 }
```

(a) Source loop

```
1 if (k >= j) goto L1
2   i = 0
3   L2:
4     A[i+k] = A[i+k] + 2;
5     B[i+k] = B[i+k-1] * 3;
6     i = i + 1;
7   if (i < j-k) goto L2
8 L1:
```

(b) TPO representation

Figure 4.7: Example loop

The first branch (guard branch) in the code of Figure 4.7(b), protects the initial execution of the loop. This branch ensures that no code between the guard branch (line 1) and the branch target (line 8) is executed if the original loop would not have been executed at least once. If the compiler can ensure that the test condition always evaluates to true (*i.e.* the initial value of the induction variable is always less than the upper bound) the guard branch can be removed by later optimizations. However, the guard branch is always present while loop optimizations are performed. All loops in TPO are changed into *do* loops in which the exit condition is tested at the end of the iteration, instead of the beginning of the iteration. This transformation is always legal because the initial execution of the loop body is protected by the guard branch.

The body of the loop remains the same; the assignments to the two arrays A and B remain unchanged. The latch branch (line 7) tests whether the induction variable is within the original bounds. As long as it is, control continues to jump back to the first statement in the loop body (line 3). As soon as the induction variable has passed the upper bound, control falls through to the second label (line 8).

4.2 Multidimensional Blocking of UPC Arrays in the XL UPC Compiler

The IBM XL UPC compiler provides an extension to the UPC language to add support for *multiblocked* or *tiled* arrays. Tiled data structures are used to enhance locality (and therefore performance) in a wide range of HPC applications [12]. Multiblocked arrays can help UPC programmers to better express the parallelism, enabling the language to fulfill its promise of allowing both high productivity and high performance. Also, having this data structure available in UPC facilitates the use of library routines, such as the Basic Linear Algebra Subprograms (BLAS) [26], that already make use of tiled data structures.

```
1 shared double A[M][N];
2 for (i=1; i<M-1; i++) {
3   upc_forall (j=1; j<N-1; j++; &A[i][j]) {
4     B[i][j] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
5   }
6 }
```

Figure 4.8: Stencil computation on a 2-dimensional shared array

Consider a simple stencil computation on a 2-dimensional array that calculates the average of the four immediate neighbours of each element as seen in Figure 4.8. Since it has no data dependencies, this loop can be executed in parallel. However, the naive declaration of A above yields suboptimal performance. For example, A[i−1][j] will likely not be on the same UPC thread as A[i][j] and thus its access may require inter-node communication. A somewhat better solution allowed by UPC is a striped 2D array distribution, created by declaring the shared array as:

shared [N∗b] **double** A[M][N];

The blocking factor, $N * b$ causes the array to be allocated in contiguous blocks of size $N * b$. This however, limits parallelism to $\frac{N}{b}$ processors and results in $O(\frac{1}{b})$ of the array accesses to be remote. By contrast, a tiled layout allows $\frac{M \times N}{b^2}$ parallelism and requires only $O(\frac{1}{b^2})$ of the accesses to be remote. Typical MPI implementations of stencil computations tile the array and exchange "border regions" between neighbours before each iteration. This approach is also possible in UPC by declaring the shared array as:

struct block **double** tile [b][b]; ;
shared block A[M/b][N/b];

However, the declaration above complicates the source code because two levels of indexing are needed for each access. We cannot pretend that A is a simple array anymore. To allow a simpler implementation of multi-dimensional stencil computations, we propose a language extension that can declare a tiled layout for a shared array, as follows:

shared [b0][b1]...[bn−1] <type> A[d0][d1] ... [dn−1];

Array A is an n-dimensional multiblocked (or tiled) array with dimensions $d_0 \ldots d_{n-1}$ with each tile being an array of dimensions $b_0 \ldots b_{n-1}$. Tiles are understood to be contiguous in memory. Section 4.3.4 explains how the RTS determines the location of a multiblocked shared array element based on the indices used to specify the element.

4.2.1 Multiblocked Arrays and UPC Pointer Arithmetic

The address of any UPC array element can be taken with the **upc_addressof** function or with the familiar `&` operator. The result is called a *pointer-to-shared*, and it is a reference to a memory location somewhere within the space of the running UPC application. In our implementation a pointer-to-shared identifies the base array as well as the thread, course and phase of an element in that array.

UPC pointers-to-shared behave much like pointers in C. They can be incremented, dereferenced, compared, *etc*. The familiar pointer operators (`*`, `&`, `++`) are available. A series of increments on a pointer-to-shared will cause it to traverse a UPC shared array in row-major order. Pointers-to-shared can also be used to point to multiblocked arrays. Users can expect pointer arithmetic and operators to work on multiblocked arrays just like on regular UPC shared arrays. Multiblocked arrays can support affinity tests (similar to the **upc_threadof** function) and type casts the same way as regular UPC arrays do.

Dynamic allocation of UPC shared arrays can also be extended to multiblocked arrays. UPC memory allocation routines always return shared variables of type **shared void** $*$, thus multiblocked arrays can be allocated with such primitives as long as they are cast to the proper type.

4.2.2 Implementation Issues

The current implementation supports only statically-allocated multiblocked arrays. Dynamically allocated multiblocked arrays could be obtained by casting dynamically allocated data to a shared multiblocked type, making dynamic multiblocked arrays a function of correct casting and multiblocked pointer arithmetic. While correct multiblocked pointer arithmetic

is not conceptually difficult, implementation is not simple: to traverse a multiblocked array correctly, a pointer-to-shared will have to have access to all blocking factors of the shared type.

Another limitation of the current implementation is related to the cyclic distribution of blocks over UPC threads. An alternative would be to specify a processor grid over which blocks could be distributed. To do this, the equations to convert a shared array index into an offset (Section 4.3.4) would have to be modified to take thread distribution into consideration. We have not implemented this yet in the RTS.

4.3 The IBM UPC Runtime System

The IBM UPC Runtime System (RTS) was designed and implemented by a group of researchers at the IBM TJ Watson Research Center [7, 10]. It has been designed for scalability in large parallel machines, such as BlueGene/L. It exposes to the compiler an Application Program Interface (API) that is used to allocate and free shared data, to access shared data, and to perform synchronization.

The RTS uses a common API implemented by one of several *transport* layers. This API includes library-specific functions to get and set shared data and perform synchronization. The RTS currently supports three transport layers: (i) SMP using Pthreads, and two types of distributed memory processing using (ii) the Low-level Application Programming Interface (LAPI) and (iii) the Blue Gene®/L message layer. This document will focus on two transport layers: Pthreads and LAPI. For hybrid environments, the transport layer uses a combination of Pthreads and LAPI.

4.3.1 The Shared Variable Directory

The Shared Variable Directory (SVD) is a partitioned data structure used by the RTS to manage allocation, de-allocation and access to shared objects. It is designed to scale to a large number of threads while allowing efficient manipulation of shared objects. Every shared variable in a UPC program has a corresponding entry in the SVD. Each shared variable has a unique *handle* that is used by the RTS to locate the object through the SVD. This handle is conceptually a *fat pointer* that contains several fields. The handle is used by the SVD to determine the location of the shared object in memory. In the case of shared arrays, the combination of the array's handle and an index are used to locate the shared object. It is the responsibility of the compiler to manage the SVD entries when shared

objects are created or go out of scope. A more detailed discussion of the SVD can be found in [7, 10, 30].

4.3.2 Allocating a Shared-Object

Shared objects are allocated in contiguous segments in each address partition. Within an address partition, the shared object is further grouped into contiguous sections that have affinity with the threads that map to the address partition. To allocate a shared array A declared as:

shared [b] <type> A[N];

the RTS first computes the total number of blocks required for A using the number of elements (Equation 4.1). Based on the total number of blocks required, the RTS computes the number of blocks that must be allocated to each thread (Equation 4.2). The local size of the array that is owned by each thread is computed using Equation 4.3. The *sizeof* operator is the traditional C *sizeof* operator that computes the size of the specified parameter. Finally, the size that is allocated on each node is computed using Equation 4.4. Note that the total number of blocks and the number of blocks-per-thread are rounded up to the nearest integer when the array size is not a multiple of the blocking factor and the number of threads. This will cause additional array elements to be allocated that will never be accessed. However, this also makes the $local\_size$ uniform across all threads, which simplifies the computations to convert an array index into an offset (see Section 4.3.4).

$$nblocks(A) = \left\lceil \frac{num\_elts}{b} \right\rceil \tag{4.1}$$

$$blocks\_per\_thread(A) = \left\lceil \frac{nblocks}{\mathcal{T}} \right\rceil \tag{4.2}$$

$$local\_size(A) = blocks\_per\_thread(A) * b * sizeof(\texttt{type}) \tag{4.3}$$

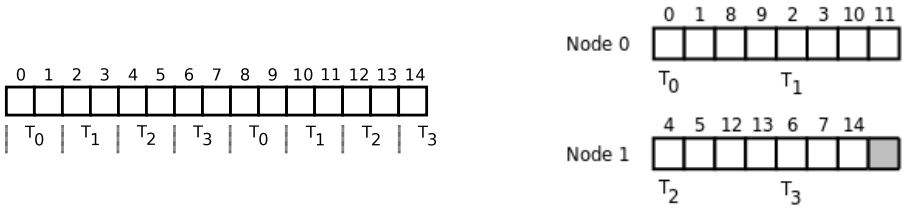
$$node\_size(A) = local\_size * threads\_per\_node \tag{4.4}$$

For example, consider a shared array declared as:

shared [2] **int** A[15];

compiled for 4 threads, with 2 threads-per-node.

Figure 4.9 shows the logical layout and physical layout per node and per thread for A. The total number of blocks is 8 and the number of blocks per thread is 2. Thus, each thread



(a) Logical data layout (all threads) (b) Physical data layout (per node and thread)

Figure 4.9: Shared and private data layouts for **shared** [4] **int** A[15], 4 threads, 2 threads per node

will *own* 4 elements of A and each address partition will contain 8 elements. Threads T_0 and T_1 map to node 0 while threads T_2 and T_3 map to node 1. Within each node, the blocks for each thread are allocated in one contiguous segment of memory. Thus, on node 0 all of the elements owned by thread T_0 are located before the elements owned by thread T_1. Similarly, on node 1 all elements owned by thread T_2 are located before the elements owned by thread T_3. The UPC specification requires all elements for a shared object that have affinity with a specific thread to be contiguous in memory. Our implementation places all elements for a shared object within an address partition into contiguous memory to facilitate the locality-aware optimizations. The final element owned by thread T_3, which corresponds to A[15] is allocated by the RTS but should never be accessed.

The RTS provides an interface to the compiler to obtain the base address for a given shared object on an address partition. This function always returns the address of the first element of a specified shared object in the current address partition. For example, if either thread T_0 or T_1 requests the base address of the shared array A in Figure 4.9, the RTS will return the address of the first element of the local shared array on node 0 (the element that corresponds to A[0]). Similarly, if either thread T_2 or T_3 requests the base address of A, the RTS will return the address of the first element of the local shared array on node 1 (the element that corresponds to A[4]).

4.3.3 Accessing a Shared-Object

Figure 4.10 shows the high-level control flow through the RTS when performing a shared array dereference. The process begins with the compiler issuing an xlupc_deref function call, specifying the handle of the shared array and the index. The RTS performs an SVD lookup to determine the location of the specified shared object. If the shared object is local, a memory copy is used to return the requested data to the caller. In this context, *local* indicates that the shared object is located in the address partition of the accessing thread.

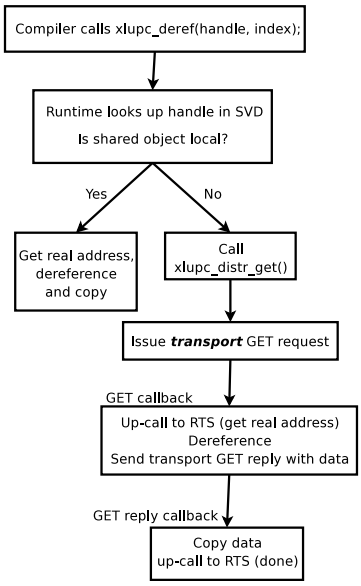


Figure 4.10: RTS control flow for a shared array dereference

Thus, the shared object will always be local for an all-to-one mapping and will be local for a many-to-one mapping if the owner thread and the accessing thread are located on the same node. For a one-to-one mapping, the shared object will only be local if it is owned by the accessing thread.

If the shared object is remote, a xlupc_distr_get () function call is issued by the RTS to call a routine that retrieves the data. The `GET` routine is responsible for communication with the owning thread and for the subsequent retrieval of the shared object. LAPI uses active messages with a *handle* routine that is run on the node that owns the shared object. This handler determines the location of the shared object, dereferences it and sends the data back in a subsequent message. Finally, when the reply has been received on the accessing node, the data is copied into the local buffer and a call to the RTS is issued to indicate that the transfer has completed. If one-sided communication is supported, as in LAPI active messages, then the `GET` routine uses it. Otherwise, the `GET` routine must use another communication protocol, such as point-to-point.

A similar process is used when assigning shared objects. Except that the `GET` commands are replaced with `PUT` commands. All other steps remain the same.

4.3.4 Index-to-Offset Conversion

As described in Section 3.1, a shared array can be viewed in two ways: the global view and the local view. A shared array element is specified using an *index* in the global view. To

access the underlying storage for a specific shared array element, the index is converted to an *offset* on the thread that owns the specific shared array element.

A UPC shared array declared as:

shared [b] <type> A[d0][d1]...[dn];

is distributed in memory in a block-cyclic manner with blocking factor b. Given an array index $\mathbf{v} = v_0, v_1, \ldots, v_{n-1}$, to locate element A[**v**] the RTS first calculates the linearized row-major index (as we would in C):

$$L(\mathbf{v}) = v_0 \times \prod_{j=1}^{n-1} d_j + v_1 \times \prod_{j=2}^{n-1} d_j + \cdots + v_{n-1} \tag{4.5}$$

The block-cyclic layout is based on this linearized index. The RTS then calculates the UPC *thread* on which array element A[**v**] resides using the *threadof* equation. The *local_threadof* is the local thread on the given node where the shared element is located. Within the local storage of this thread the array is kept as a collection of blocks. The *courseof* of an array location is the block number in which the element resides; the *phaseof* is its location within the block.

$$threadof\,(A_s, \mathbf{v}) = \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \,\%\, \mathcal{T} \tag{4.6}$$

$$local\_threadof\,(A_s, \mathbf{v}) = \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \,\%\, \mathcal{T}_{AP} \tag{4.7}$$

$$courseof\,(A_s, \mathbf{v}) = \left\lfloor \frac{L(\mathbf{v})}{b \times \mathcal{T}} \right\rfloor \tag{4.8}$$

$$phaseof\,(A_s, \mathbf{v}) = L(\mathbf{v}) \,\%\, b \tag{4.9}$$

Consider again the shared array in Figure 4.9. Given two shared array accesses, A[7] and A[8], the RTS first computes the thread that owns the elements of A using Equation 4.6. Using the linearized indices, 7 and 8, $b = 2$, $\mathcal{T} = 4$, and $\mathcal{T}_{AP} = 2$ the RTS determines that thread 3 owns A[7] and thread 0 owns A[8]. However, given that there are two threads per address partition, the $local\_threadof\,(\mathrm{A}, 7) = 1$ and $local\_threadof\,(\mathrm{A}, 8) = 0$. The $courseof$, which corresponds to the block index containing the array element, is computed using Equation 4.8. The course for the two array elements are 0 and 1 respectively. The $phaseof$, computed using Equation 4.9 determines the offset within the block. These are determined to be 1 and 0 for the two shared array elements.

To determine the offset (in bytes) from the base address for a shared array A_s with blocking factor $\mathcal{B}$ and type type, accessed at index **v**, the RTS uses Equation 4.10.

$$\begin{aligned} \mathit{offset}(A_s, \mathbf{v}) &= \mathit{local\_threadof}(A_s, \mathbf{v}) * \mathit{local\_size}(A_s) + \\ &\quad (\mathit{courseof}(A_s, \mathbf{v}) * \mathcal{B} + \mathit{phaseof}(A_s, \mathbf{v})) * \mathit{sizeof}(\texttt{type}) \end{aligned} \quad (4.10)$$

The *local_size* is computed using Equation 4.3. For the shared array in Figure 4.9, the *local_size* is computed to be 16, assuming **sizeof**(`int`) = 4. Thus, the offsets for the shared array accesses A[7] and A[8] are 20 bytes and 8 bytes respectively.

This example illustrates the difference between the logical and physical layouts of shared arrays. Two adjacent shared array elements in the global layout do not necessarily reside in adjacent memory locations in the local shared array layout. In fact, as seen in this example, adjacent shared array elements in the global layout do not necessarily map to the same address partition.

Multiblocked Arrays

The goal of multiblocked arrays is to extend UPC syntax to declare tiled arrays while minimizing the impact on language semantics. Therefore, the internal representation of multiblocked arrays should not differ significantly from that of standard UPC arrays. Consider a multiblocked array A with dimensions $\mathcal{D} = \{d_0, d_1, \ldots, d_{n-1}\}$ and blocking factors $\mathcal{B} = \{b_0, b_1, \ldots, b_{n-1}\}$. This array would be allocated in $k = \prod_{i=0}^{n-1} \left\lceil \frac{d_i}{b_i} \right\rceil$ blocks (or tiles) of $b = \prod_{i=0}^{n-1} b_i$ elements. We continue to use the concepts of *threadof*, *courseof* and *phaseof* to find array elements. However, for multiblocked arrays two linearized indices must be computed: one to find the block and another to find an element's location within a block. Note the similarity of Equations 4.12 and 4.11 to Equation 4.5:

$$B(A_s, \mathbf{v}) = \sum_{k=0}^{n-1} \left(\left\lfloor \frac{v_k}{b_k} \right\rfloor \times \prod_{j=k+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \right) \quad (4.11)$$

$$L_{in-block}(A_s, \mathbf{v}) = \sum_{k=0}^{n-1} \left((v_k \ \% \ b_k) \times \prod_{j=k+1}^{n-1} b_j \right) \quad (4.12)$$

Equation 4.11 computes the linearized block index for a specific shared array element A[**v**], where $\mathbf{v} = v_0, v_1, \ldots, v_{n-1}$ The *phaseof* of a multiblocked array element is its linearized in-block index, computed using Equation 4.12. The *courseof* and *threadof* are calculated with a cyclic distribution of the block index, as in the case of uni-dimensional UPC arrays.

$$threadof(A_s, \mathbf{v}) = B(A_s, \mathbf{v}) \% \mathcal{T} \tag{4.13}$$

$$courseof(A_s, \mathbf{v}) = \left\lfloor \frac{B(A_s, \mathbf{v})}{\mathcal{T}} \right\rfloor \tag{4.14}$$

$$phaseof(A_s, \mathbf{v}) = L_{in-block}(\mathbf{v}) \tag{4.15}$$

When handling multiblocked arrays, the RTS uses Equation 4.16 to determine the offset from the base address on a given thread.

$$offset(A_s, \mathbf{v}) = \left(courseof(A_s, \mathbf{v}) * \prod_{i=0}^{n-1} b_i + phaseof(A_s, \mathbf{v}) \right) * sizeof(\texttt{type}) \tag{4.16}$$

The dimensions of the multiblocked arrays do not need to be multiples of their respective blocking factors, just as the array dimension of a regular UPC array is not required to be a multiple of the blocking factor. Arrays are padded in every dimension to allow for correct index calculation.

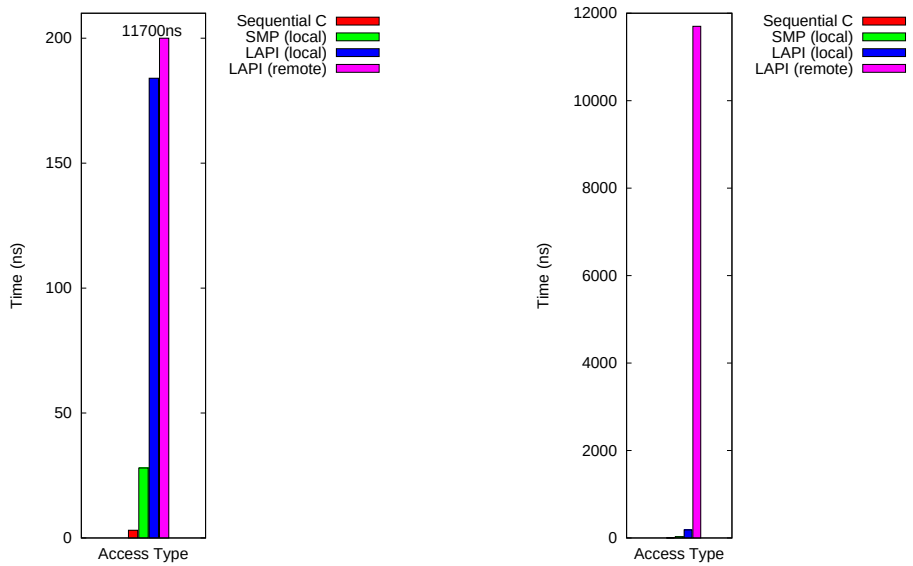
4.3.5 Shared-Object Access Timing

Figure 4.11 shows the times to access a shared array element using the different transports in the RTS (SMP, LAPI local and LAPI remote). In this context LAPI local refers to a different thread of the same address partition. Thus, the LAPI transport is still used to perform the access but no network communication is required. The time to access an array element in a C array is also given as a reference. These results were collected on a 1.9 GHz POWER5 machine. The tests were compiled with -O3 -qhot.

Figure 4.11(a) shows the difference in local access times. The difference between the sequential C time and the local SMP time (5X increase) is due to the additional calculations needed to determine the course and phase when computing the offset. The local LAPI access includes the same calculations to determine the thread, course and phase but also calls down to the transport layer to perform the actual access, resulting in a 6.5x overhead over the SMP access. Finally, the remote LAPI access (11700ns) requires network communication in addition to the address calculations, resulting in a 58x increase in time over a local LAPI access, as seen in Figure 4.11(b).

4.4 Chapter Summary

This chapter presented an overview of the IBM UPC Compiler and Runtime System. The discussion of TPO included an overview of the optimizations performed on UPC code and



(a) Close-up of local access times (b) Full-scale local and remote access times

Figure 4.11: Time to access a shared-array element

a discussion of some of the internal representations used by TPO. The concept of a *multi-blocked array*, a solution for one limitation of the UPC language, was introduced. Nishtala *et al.* have shown that multiblocked arrays can be combined with a new collective interface, which is another limitation identified in Section 2.8, to improve both performance and scalability of UPC programs [46]. The discussion of the IBM RTS included a description of the SVD, an overview of the process of accessing a shared-object and a description of how the RTS converts between a shared-array index and the corresponding address offset. This conversion is used throughout Chapter 6 when optimizing shared-object accesses. Finally, the cost of accessing local and remote shared objects through the RTS using two types of transports (SMP and LAPI) was discussed. These times demonstrate some of the overheads incurred when using the RTS access shared objects and provide motivation for the compiler to bypass the RTS and directly access the shared objects whenever possible.

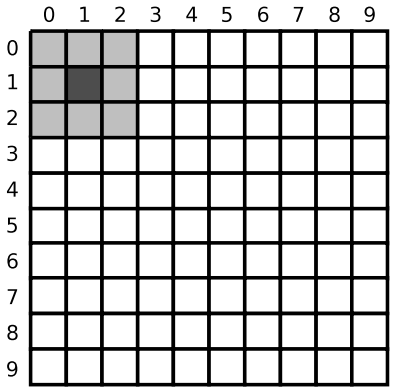
Chapter 5

Locality Analysis

As seen in the previous chapter, there is a significant overhead to accessing shared-objects using the RTS. Thus, to obtain reasonable performance it is crucial that the executing thread bypass the RTS and directly access local shared-objects whenever possible.

```
1  shared [10] double A[10][10];
2  shared [10] double B[10][10];
3  for (i=1; i < 9; i++) {
4    upc_forall(j=1; j < 9; j++; &A[i][j]) {
5      B[i][j] = (A[i-i][j-1] + A[i-1][j] +
6                 A[i-1][j+1] + A[i][j-1] +
7                 A[i][j]     + A[i][j+1] +
8                 A[i+1][j-1] + A[i+1][j] +
9                 A[i+1][j+1]) / 9.0;
10   }
11 }
```

(a) 9-point stencil example



(b) Access pattern

Figure 5.1: Example stencil computation in UPC

Consider the 9-point stencil example in Figure 5.1(a). This example uses two 10x10 shared arrays, distributed such that each row in the shared arrays have affinity with a thread. For a given element at position A[i][j] the values at the eight neighbouring positions are obtained and used to compute an average. This average is assigned to B[i][j]. Consider the case where this example is run on a distributed architecture with four threads. For every assignment to B[i][j] the compiler will generate nine calls to the RTS to retrieve elements of A and an additional call to the RTS to assign an element of B. Of the elements of A that are accessed, three are owned by the accessing thread (A[i][j−1], A[i][j], and A[i][j+1]) because of the affinity test and the distribution of A. The remaining six array elements are owned by other threads, resulting in remote accesses. Based on the access times presented

in Chapter 4.3.5, there will be a significant overhead to perform these accesses using the RTS. However, the compiler can convert the calls to the RTS for the three accesses that are owned by the accessing thread (A[i][j−1], A[i][j], and A[i][j+1]), into direct memory accesses, thereby achieving performance similar to the performance of a C pointer dereference for these accesses. Similarly, remote accesses that are owned by the same thread can be grouped together in a single access to reduce the number of calls to the RTS and hence the number of messages that are required. However, to perform these optimizations, the compiler must be able to identify the thread that owns the shared-objects being accessed. This chapter presents a new shared-object locality analysis developed for UPC shared arrays with multi-dimensional blocking factors. The algorithm uses a shared-object access with a known locality to determine the locality of other related shared-object accesses. This locality information is used by the locality optimizations presented in Chapter 6 to optimize the shared-object accesses. While this analysis is presented specifically for UPC, it should apply to any language that uses a partitioned global address space.

A previously published analysis allowed the compiler to distinguish between local and remote shared-object accesses by computing the relative *node* ID of each shared reference. All shared references that map to node 0 are local and thus candidates for privatization. All shared references that map to a node other than 0 are remote and are handled using calls to the RTS [9]. This analysis differs from that previous work because it provides more information about the owners of each shared reference. Instead of categorizing the shared references into local and remote, it tracks the relative thread ID for each reference. This provides the compiler with more precise information that can be used to perform additional locality optimizations described in Chapter 6.

5.1 Shared Object Properties

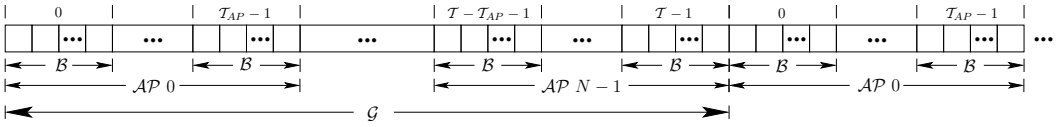


Figure 5.2: Shared-array layout in UPC

Figure 5.2 illustrates that $\mathcal{G} = \mathcal{B} \times \mathcal{T}$ elements of a shared array A_s are allocated to the threads in a cyclic fashion. Each array segment with $\mathcal{G}$ elements forms a block group. The memory layout pattern shown in Figure 5.2 is only dependent on the blocking factor

$\mathcal{B}$ and the number of threads that map to each address partition, $\mathcal{T}_{AP}$. Thus two shared arrays with the same blocking factor in the same program execution always have the same layout because the mapping of threads to address partitions is constant. This distribution is independent of the overall sizes of the shared arrays. Therefore, the analysis presented in this section does not require that the two shared-array elements be located in the same shared array. The only requirement is that the two shared arrays have the same blocking factor.

Given two shared references that represent accesses to elements of a shared array, the compiler can analyze the two indices to determine if both accesses refer to data owned by the same thread. In the following discussion, i and j represent indices into a shared array A_s while the normalized indices $i' = i \% |\mathcal{G}|$ and $j' = j \% |\mathcal{G}|$ represent the offset of i and j within their block group. $Owner(O_s)$ represents the thread that owns the shared object O_s.[1]

Property 5.1 below states that when the distance between the two normalized indices is zero, both array elements are owned by the same thread. The distance of two normalized indices is computed by subtracting the smaller index from the larger index. Property 5.2 addresses the case when the distance between the two normalized indices is less than the blocking factor. In this case it is possible for the two array elements to map to the same thread or to different threads. The relationship between the two indices determines which case occurs. Property 5.3 states that when the distance between the normalized indices is between the blocking factor and the group size, the two array elements are owned by different threads. It is not possible for the difference between the normalized indices to be larger than the group size.

Without loss of generality, all the properties below assume that the following relation holds:

$$i' \leq j' \tag{5.1}$$

Property 5.1 *If* $i' = j'$ *then* $Owner(A_s[i]) = Owner(A_s[j])$

Proof 5.1 $Owner(A_s[i]) = Owner(A_s[j])$ *if and only if* $\frac{i'}{\mathcal{B}} \% \mathcal{T} = \frac{j'}{\mathcal{B}} \% \mathcal{T}$ *which is trivially true from the premise that* $i' = j'$. ■

Property 5.2 *If the following condition is true:*

$$1 \leq j' - i' < \mathcal{B} \tag{5.2}$$

[1] Recall that since O_s and A_s[i] can be used interchangably (see page 2), $Owner(O_s) \equiv Owner(A_s[\mathbf{i}])$.

then

1. *If* $i' \% \mathcal{B} < j' \% \mathcal{B}$ *then* $Owner(A_s[i]) = Owner(A_s[j])$
2. *If* $i' \% \mathcal{B} \geq j' \% \mathcal{B}$ *then* $Owner(A_s[i]) \neq Owner(A_s[j])$

Proof 5.2

This proof will use the following relations that are true for any positive integers i', j', *and* $\mathcal{B}$*:*

$$\left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \times \mathcal{B} = i' - (i' \% \mathcal{B}) \tag{5.3}$$

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \times \mathcal{B} = j' - (j' \% \mathcal{B}) \tag{5.4}$$

From assumptions 5.1 and 5.2 it follows that:

$$\left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \leq \left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \leq \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor + 1 \tag{5.5}$$

Case 1: We want to prove that $Owner(A_s[i]) = Owner(A_s[j])$*. To do this, it is sufficient to show that* $A_s[i]$ *and* $A_s[j]$ *are located in the same block. Thus, we must show that* $\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor$*. For the sake of contradiction, assume that:*

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor + 1 \tag{5.6}$$

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \times \mathcal{B} = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \times \mathcal{B} + \mathcal{B} \tag{5.7}$$

Replacing 5.3 and 5.4 in 5.7:

$$j' - (j' \% \mathcal{B}) = i' - (i' \% \mathcal{B}) + \mathcal{B} \tag{5.8}$$

Using the assumption $i' \% \mathcal{B} < j' \% \mathcal{B}$ *in 5.8:*

$$j' - i' - \mathcal{B} = (j' \% \mathcal{B}) - (i' \% \mathcal{B}) > 0 \tag{5.9}$$

$$j' - i' > \mathcal{B} \tag{5.10}$$

Inequality 5.10 contradicts assumption 5.2. Thus it must be the case that

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \neq \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor + 1 \tag{5.11}$$

From 5.11 and 5.5 it follows that $\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor$*. Thus* $Owner(A_s[i]) = Owner(A_s[j])$*.*

Case 2: Here, we want to prove that $Owner(A_s[i]) \neq Owner(A_s[j])$. To do this, it is sufficient to show that $A_s[i]$ and $A_s[j]$ belong to different blocks. Thus, we want to show that $\left\lfloor \frac{j'}{B} \right\rfloor \neq \left\lfloor \frac{i'}{B} \right\rfloor$. For the sake of contradiction, assume that:

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \tag{5.12}$$

$$\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \times \mathcal{B} = \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \times \mathcal{B} \tag{5.13}$$

Replacing 5.3 and 5.4 in 5.13:

$$j' - (j' \% \mathcal{B}) = i' - (i' \% \mathcal{B}) \tag{5.14}$$

$$(i' \% \mathcal{B}) - (j' \% \mathcal{B}) = i' - j' \tag{5.15}$$

Using the assumption $i' \% \mathcal{B} \geq j' \% \mathcal{B}$ in 5.15:

$$0 \leq i' - j' \tag{5.16}$$

$$j' \leq i' \tag{5.17}$$

The relation 5.17 contradicts assumption 5.1. Thus, it must be the case that $\left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \neq \lfloor \frac{i'}{\mathcal{B}} \rfloor$, and thus $Owner(A_s[i]) \neq Owner(A_s[j])$. ■

Intuitively, Property 5.2 can be explained as follows: since the distance between the two shared objects is less than the size of a block, either $A_s[i]$ and $A_s[j]$ are in the same block (Case 1) or $A_s[i]$ is in block k and $A_s[j]$ is in block $k+1$ (Case 2). The positions of i and j within their own blocks is used to determine which of these two cases holds.

Property 5.3 *If $j' - i' \geq \mathcal{B}$ then $Owner(A_s[i]) \neq Owner(A_s[j])$*

Proof 5.3 *For the sake of contradiction assume that:*

$$\left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor = \left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \tag{5.18}$$

Replacing equations 5.3 and 5.4 in the hypothesis of this property:

$$j' - i' = \left\lfloor \frac{j'}{\mathcal{B}} \right\rfloor \times \mathcal{B} + (j' \% \mathcal{B}) - \left\lfloor \frac{i'}{\mathcal{B}} \right\rfloor \times \mathcal{B} + (i' \% \mathcal{B}) \geq \mathcal{B} \tag{5.19}$$

Replacing 5.18 into 5.19:

$$(j' \% \mathcal{B}) - (i' \% \mathcal{B}) \geq \mathcal{B} \tag{5.20}$$

But by definition, $(j' \% \mathcal{B}) < \mathcal{B}$ and $(i' \% \mathcal{B}) < \mathcal{B}$, therefore its difference cannot be greater or equal to $\mathcal{B}$. Thus, equation 5.18 is false and it must be true that $Owner(A_s[i]) \neq Owner(A_s[j])$. ■

5.2 Shared-Object Locality Analysis

The goal of shared-object locality analysis is to identify the relative thread ID that owns each shared reference in a **upc_forall** loop. This analysis will allow the compiler to perform specific optimizations on the shared references based on the locality information. The key to the locality analysis is the observation that for each dimension, a block can span at most two threads. Therefore, in each dimension the locality can only change in one place. We call this place the *cut*.

5.2.1 Cuts

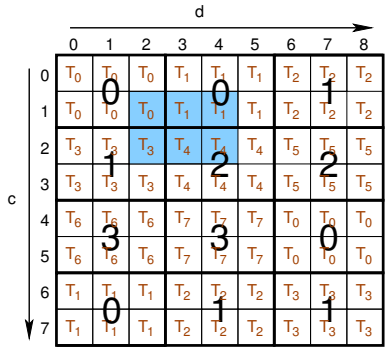
Given a **upc_forall** loop where the shared reference in the affinity test is formed by the current loop-nest index, every shared reference in the **upc_forall** loop body has a displacement with respect to the affinity expression. This displacement is specified by the distance vector $\mathbf{k} = [k_0, k_1, \ldots, k_{n-1}]$, where k_i represents the number of array elements between two shared references in dimension i.

Property 5.2 demonstrates that when the displacement distance between two shared references in a given dimension is less than the blocking factor in that dimension, the shared references can have two owners.

Definition 5.1 *The value of a cut in dimension i, Cut_i, is the distance, measured in number of elements, between the first element of a block and the transition between threads on that dimension.*

```
1 /* 8 threads and 2 threads/node */
2
3 shared [2][3] int A[8][9];
4
5 for(int c=0; c<7; c++){
6   upc_forall(int d=0; d<6; d++; &A[c][d]){
7     A[c+1][d+2] = c*d;
8   }
9 }
```

(a) Example UPC loop nest



(b) Multiblocked shared-array layout

Figure 5.3: A two-dimensional array example.

Consider the two-level loop nest that accesses a two-dimensional blocked array shown in Figure 5.3(a). Assume that this example is compiled for 8 threads and 4 nodes, resulting

in 2 threads-per-node. The layout of the array used in the loop is shown in Figure 5.3(b). Thin lines separate the elements of the array. Large bold numbers inside each block of 2×3 elements denote the node ID to which the block is mapped. Thick lines separate threads from each other. The grey area in the array represents all elements that are referenced by iterations of the **upc_forall** loop that are affine with &A[0][0]; cuts in this iteration space are determined by the thick lines (thread boundaries).

Finding the Cuts

In general, for dimension i the value of the cut is computed using Equation 5.21.

$$Cut_i = b_i - k_i \,\%\, b_i \tag{5.21}$$

Where k_i is the distance between the shared reference in the affinity statement and the shared reference in the **upc_forall** body in dimension i, and b_i is the blocking factor of the shared reference in dimension i.

In the example in Figure 5.3, the distance vector for shared array A is $[1, 2]$. Thus, the cut in the outer dimension (i=0) is $Cut_0 = (2-1) \,\%\, 2 = 1$. Similarly, the cut for the inner dimension (i=1) is $Cut_1 = (3-2) \,\%\, 3 = 1$. This indicates that the locality changes between c=1 and c=2 in the outer dimension and between d=1 and d=2 in the inner dimension.

Theorem 5.1 *Let A_s be an n-dimensional shared array with dimensions $d_0, d_1, \ldots, d_{n-1}$ and with blocking factors $b_0, b_1, \ldots, b_{n-1}$. Let p be an arbitrary dimension, let $\mathbf{w} = v_0, v_1, \ldots, v_p, \ldots, v_{n-1}$ and $\mathbf{y} = v_0, v_1, \ldots, v_p+1, \ldots v_{n-1}$ be two vectors such that $A(\mathbf{w})$ and $A(\mathbf{y})$ are elements of A. Notice that $\mathbf{w}$ and $\mathbf{y}$ only differ by 1 in dimension p. Let*

$$v'_i = v_i \,\%\, b_i - k_i \,\%\, b_i \tag{5.22}$$

If $v'_p \neq Cut_p - 1$ then $Owner(\mathbf{w}) = Owner(\mathbf{y})$. Note that $p < n-1$.

Proof 5.4 *The expressions for the owner of elements $\mathbf{w}$ and $\mathbf{y}$ are given by:*

$$Owner(\mathbf{w}) = L(\mathbf{w}) \,\%\, \mathcal{T} \quad and \quad Owner(\mathbf{y}) = L(\mathbf{y}) \,\%\, \mathcal{T} \tag{5.23}$$

where $L(\mathbf{w})$ is the linearized block index for $\mathbf{w}$ as defined in Equation 4.5. $L(\mathbf{w})$ and $L(\mathbf{y})$ can be written as:

$$L(\mathbf{w}) = \sum_{i=0}^{n-1} \left\lfloor \frac{v_i}{b_i} \right\rfloor \times \prod_{j=i+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{5.24}$$

$$L(\mathbf{y}) = L(\mathbf{w}) + \left(\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor - \left\lfloor \frac{v_p}{b_p} \right\rfloor \right) \times \prod_{j=p+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{5.25}$$

Substituting equations 5.21 and 5.22 in the assumption that $v'_p \neq Cut_p - 1$, results in:

$$v_p \% b_p - k_p \% b_p \neq b_p - k_p \% b_p - 1 \tag{5.26}$$

$$v_p \% b_p \neq b_p - 1 \tag{5.27}$$

For any integers v_p and b_p it is true that:

$$v_p \% b_p < b_p \tag{5.28}$$

$$v_p \% b_p = v_p - \left\lfloor \frac{v_p}{b_p} \right\rfloor \times b_p \tag{5.29}$$

From 5.27 and 5.28, it follows that:

$$v_p \% b_p < b_p - 1 \tag{5.30}$$

Therefore adding one to v_p is equivalent to adding one to the module $v_p \% b_p$:

$$(v_p + 1) \% b_p = v_p \% b_p + 1 \tag{5.31}$$

Using relation 5.29 for both v_p and $v_p + 1$ and substituting in equation 5.31:

$$v_p + 1 - \left\lfloor \frac{v_p + 1}{b_p} \right\rfloor \times b_p = v_p - \left\lfloor \frac{v_p}{b_p} \right\rfloor \times b_p + 1 \tag{5.32}$$

which can be simplified to

$$\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor = \left\lfloor \frac{v_p}{b_p} \right\rfloor \tag{5.33}$$

Substituting this result in equation 5.25 results that $L(\mathbf{w}) = L(\mathbf{y})$ and therefore $Owner(\mathbf{w})$. $Owner(\mathbf{y})$. ■

Theorem 5.1 can be stated informally as follows: two adjacent elements, $A(\mathbf{w})$ and $A(\mathbf{y})$, of an n-dimensional shared array A_s must have the same owner unless $A(\mathbf{w})$ is on one side of a cut and $A(\mathbf{y})$ is on the other side of the cut.

5.2.2 Algorithm

The locality analysis algorithm considers one parallel loop nest at a time. At each nest level, the compiler computes the cuts for each shared reference using the properties in Section 5.1. Using these cuts, the iteration space of the loop is refactored into regions. Regions are created using cuts such that a shared reference is owned by the same thread for every iteration of the loop that falls into the region. Each region is a subset of the original iteration space and the collection of all regions will form the iteration space of the original

loop. For each region, the compiler keeps track of one value of the loop induction variable that falls into the region. We call this information the *position* of the region. When all cuts have been determined and the corresponding regions created, the compiler computes the owner of each shared reference in each region using the position information. The owner of the shared reference will always remain the same in the given region.

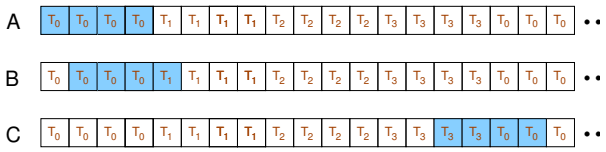
The locality analysis works on parallel loop nests that contain a **upc_forall** loop. The affinity test used in the **upc_forall** loop provides the analysis with a shared reference with a known locality, meaning the thread that executes that loop iteration will "own" the shared reference used in the affinity test. The locality analysis determines the relative position of all other shared references inside the loop, relative to the affinity test, and uses that information to compute therelative thread ID of the owners. A **upc_forall** loop can use one of two types of affinity test: integer or pointer-to-shared. A *typical* integer affinity test will contain the loop induction variable, which may be modified by a constant factor (*i.e.* **upc_forall** (**int** i=0; i $<$ N; i++; i)). A *typical* pointer-to-shared affinity test will use the loop induction variable to access a shared array (*i.e.* **upc_forall** (**int** i=0; i $<$ N; i++; A[i]), where A is a shared array). Again, this index can be modified by a constant value. If the affinity test uses a non-affine function or contains a constant that cannot be determined at compile time, the **upc_forall** loop is not considered for locality analysis. This is discussed further in Section 5.2.3.

The example in Figure 5.4 will be used to illustrate how the compiler uses the properties in Section 5.1 to determine the relative thread ID that owns a shared reference. Assume that this code is compiled for four threads and a machine that runs one thread per address partition. Thus, referring to Figure 5.2, we have $\mathcal{B} = 4$, $\mathcal{T}_{AP} = 1$, $\mathcal{T} = 4$, $\mathcal{G} = 16$ and $\mathcal{AP} = 4$. The shared array A contains N elements distributed in blocks of four elements per thread. For this example, assume that N is greater than 32. Thread 0 owns A[0] to A[3], A[16], to A[19]; thread 1 owns A[4] to A[7], A[20] to A[23] and so forth.

Figure 5.4(b) shows the access pattern for the three shared arrays on the first four iterations of the **upc_forall** loop: j=0, j=1, j=2, and j=3. These four iterations will be executed by thread 0. In iteration j=0, the statements on lines 9 and 10 are local to thread 0 (A[i][0] and B[i][1] respectively) while the statement on line 11 is not local (C[i][14] is owned by thread 3). For iteration j=3, the statements on lines 9 and 11 are local to thread 0 (A[i][3] and C[i][17] are owned by thread 0) while the statement on line 10 is not (B[i][4] is owned by thread 1). The compiler must decide for which values of j the shared references are owned by `MYTHREAD` and for which values the shared references are owned by a dif-

```
1  shared [4] int A[N][N];
2  shared [4] int B[N][N];
3  shared [4] int C[N][N];
4
5  int main() {
6    int i,j;
7    for (i=0; i < N; i++) {
8      upc_forall(j=0; j < N-14; j++; &A[i][j]) {
9        A[i][j] = 0;
10       B[i][j+1] = j*2;
11       C[i][j+14] = j-3;
12     }
13   }
14 }
```

(a) UPC loop nest



(b) Access patterns for shared arrays A, B, and C

Figure 5.4: UPC parallel loop containing 3 shared references

ferent thread. Relating these shared references to the properties of Section 5.1, the shared reference in the affinity test (A[i][j]) corresponds to $A_s[i]$ and the shared references in the **upc_forall** loop body (A[i][j], B[i][j+1], and C[i][j+14]) correspond to $A_s[j]$.

Locality analysis is done on the n-dimensional blocks of the multiblocked arrays present in the loop. For conventional UPC shared arrays declared with a single blocking factor b, the analysis uses blocking factors of 1 in all dimensions except the inner-most dimension, where b is used. For example, a shared array declared as:

shared [4] **int** A[N][N];

is implicitly translated into a shared array declared as:

shared [1][4] **int** A[N][N];

We start with the original loop nest as a single region. This region is analyzed and the cuts are computed. The iteration space is then split according to the cuts generated. Each time the iteration space is split, two new regions are created. The newly generated regions are again analyzed and split recursively until no more cuts are required. As the iteration space is being split, the compiler tracks the relative owner of each shared reference in each

region. When all of the regions have been generated, we use the position of the region, and the position of the shared reference within the region to determine the relative thread that owns the shared reference. The shared reference map is then constructed, associating each shared reference with the relative thread that owns it. The shared reference map is implemented as a hash table where the owner of the shared reference is used as the index. For each index, a list of shared references owned by the index is stored. Since a shared reference can span at most two threads in each dimension, the shared reference map will contain at most $2 * |SharedRefList| * \mathcal{D}$ entries, where $|SharedRefList|$ is the number of shared references in the original **upc_forall** loop that are candidates for locality analysis and D is the number of dimensions of a shared reference.

```
LocalityAnalysis(Procedure p)
1. NestSet ← GatherForallLoopNests(p)
2. foreach loop nest L in NestSet
Phase 1 - Gather Candidate Shared References
3.        l_forall ← upc_forall loop found in loop nest L
4.        nestDepth ← depth of L
5.        AffStmt ← Affinity statement used in l_forall
6.        SharedRefList ← CollectSharedReferences(l_forall, AffStmt)
Phase 2 - Restructure Loop Nest
7.        FirstRegion ← InitializeRegion(L)
8.        L_R ← FirstRegion
9.        L_R^final ← φ
10.       while L_R not empty
11.           R ← Pop head of L_R
12.           CutList ← GenerateCutList(R, SharedRefList)
13.           if R.nestLevel < nestDepth − 1
14.             L_R ← L_R ∪ GenerateNewRegions(R, CutList)
15.           else
16.             L_R^final ← L_R^final ∪ GenerateNewRegions(R, CutList)
17.           end if
18.       end while
Phase 3 - Build Shared Reference Map
19.       Shared Reference Map M ← φ
20.       foreach R in L_R^final
21.           foreach r_s in SharedRefList
22.               r_s^owner ←ComputeOwner(r_s, R)
23.               M[r_s^owner] ← M[r_s^owner] ∪ r_s
24.           end for
25.       end for
26. end for
```

Figure 5.5: Locality analysis for UPC shared references

The LocalityAnalysis algorithm in Figure 5.5 begins by collecting all top-level loop nests that contain a candidate **upc_forall** loop. To be a candidate for locality analysis, a **upc_forall** loop must be normalized (lower bound begins at 0 and the increment is 1) and

must use a pointer-to-shared or integer argument for the affinity test. A **upc_forall** loop is normalized in the same manner as defined in Chapter 4.1.1. The algorithm then proceeds to analyze each loop nest independently (step 2).[2] In Figure 5.4, the loop nest on line 7 is collected by the GATHERFORALLLOOPNEST algorithm and placed into $NestSet$.

Phase 1 of the per-loop nest analysis algorithm finds the **upc_forall** loop l_{forall} in the loop nest. A loop nest can only contain a single **upc_forall** loop. If it contains multiple **upc_forall** loops, the outermost **upc_forall** loop behaves as the **upc_forall** loop and the inner loops behave as **for** loops. The affinity statement, $AffStmt$, used in l_{forall} is also obtained. Finally the COLLECTSHAREDREFERENCES procedure collects all candidate shared references, in l_{forall}. In order to be a candidate for locality analysis, a shared reference must have the same blocking factor as the shared reference used in the affinity test. The compiler must also be able to compute the displacement vector $k = r_s - r_s^{aff}$ for the shared reference, the vectorized difference between the indices of the shared reference in the loop body and of the shared reference in the affinity statement.

In the example in Figure 5.4 the loop on line 8 is identified as the **upc_forall** loop in step 3. The shared references on lines 9, 10, and 11 are collected as candidates for locality analysis; the computed displacement vectors are $[0, 0]$, $[0, 1]$, and $[0, 14]$ respectively.

Phase 2 of the algorithm restructures the loop nest by splitting the iteration space of each loop into *regions* where the locality of shared references remains constant. Each region has a *statement list* associated with it, i.e. the lexicographically ordered list of statements as they appear in the program. Each region is also associated with a *position* in the iteration space of the loops containing the region.

In the example in Figure 5.4 the first region, R_0, contains the statements on line 8 to 12. The position of R_0 is $[0]$, since the iteration space of the outermost loop contains the location 0. Once initialized, the region is placed into a list of regions, $\mathcal{L}_R$ (step 8).

The algorithm iterates through all regions in $\mathcal{L}_R$. For each region, a list of cuts is computed based on the shared references collected in Phase 1. A cut represents a transition between threads that own the shared reference in the given region. The GENERATECUTLIST algorithm first determines the loop-variant induction variable, iv, in R that is used in r_s. The use of iv identifies the dimension in which to obtain the blocking factor and displacement when computing the cut. Equation 5.21 from Section 5.2.1 is used to compute the cut.

The GENERATECUTLIST algorithm sorts all cuts in ascending order. Duplicate cuts

[2] We use the word step to refer to the lines presented in Figure 5.5.

and cuts outside the iteration space of the region ($Cut = 0$ or $Cut \geq b_i$) are discarded. Finally, the current region is cut into multiple iteration ranges, based on the cut list, using the GENERATENEWREGION algorithm. New regions are formed using the body of the outermost loop in the current region. That is, the first statement in the outer loop becomes the first statement in the new region and the last statement in the outer loop becomes the last statement in the new region. Newly created regions are cut using a branch statement containing a *cut expression* of the form $iv \% b_i < Cut$ (the modulo is necessary to ensure that the cut always falls within a block). The body of the new region is replicated under each cut expression. If the cut list is empty a new region is still created; however, it does not contain any cut expressions and thus no branches are inserted and the statements in the region are not replicated.

Step 13 determines if the region R is located in the innermost loop in the current loop nest (*i.e.* there are no other loops inside of R). If R contains innermost statements, the regions generated by GENERATENEWREGIONS are placed in a separate list of final regions. $\mathcal{L}_R^{final}$. This ensures that at the end of Phase 2, the loop nest has been refactored into several iteration ranges and the final statement lists (representing the innermost loops) are collected for use in Phase 3.

The second phase iterates through the example in Figure 5.4 twice. The first region R_0 and the $CutList = \phi$, calculated by GENERATECUTLIST, are passed to the GENERATENEWREGIONS algorithm in step 14. Since the cut list is empty, no cuts are created. However, a new region, R_1 is created and added to L_R. R_1 contains the statements on lines 9, 10, and 11 and has a position of $[0]$ associated with it.

The new region R_1 is popped from $\mathcal{L}_R$ and passed to the GENERATECUTLIST algorithm. The three statements in R_1, A[i][j], B[i][j+1], and C[i][j+14] generate three cuts: 0, 3, and 2 respectively. These three cuts are sorted in ascending order and then used by GENERATENEWREGIONS to create regions R_2, R_3, and R_4 as seen in Figure 5.6. Thus region R_3 is created by the cut 2, which was generated by the statement C[i][j+14]. The position of the three new regions are $[0, 0]$, $[0, 2]$, and $[0, 3]$ respectively. Since R_1 is the innermost region, the new regions R_2, R_3, and R_4 are added to the final region list, $\mathcal{L}_R^{final}$ (step 16).

Phase 3 of the algorithm uses the position information stored in each of the final regions to compute the position of each shared reference in that region. This information is then used to build the shared reference map. The COMPUTEOWNER algorithm computes the position of the shared reference using the position of the region and the displacement vector

```
1  shared [4] int A[N][N];
2  shared [4] int B[N][N];
3  shared [4] int C[N][N];
4
5  int main() {
6    int i,j;
7    for (i=0; i < N; i++) {
8      upc_forall(j=0; j < N-14; j++; &A[i][j]) {
9        if (j < 2) {
10         /*** Begin Region 2, position [0,0] ***/
11         A[i][j]    = 0;   // shared reference A1
12         B[i][j+1]  = m*2; // shared reference B1
13         C[i][j+14] = m-3; // shared reference C1
14         /********** End Region 2 **********/
15       }
16       else if (j < 3) {
17         /*** Begin Region 3, position [0,2] ***/
18         A[i][j]    = 0;   // shared reference A2
19         B[i][j+1]  = m*2; // shared reference B2
20         C[i][j+14] = m-3; // shared reference C2
21         /********** End Region 3 **********/
22       }
23       else {
24         /*** Begin Region 4, position [0,3] ***/
25         A[i][j]    = 0;   // shared reference A3
26         B[i][j+1]  = m*2; // shared reference B3
27         C[i][j+14] = m-3; // shared reference C3
28         /********** End Region 4 **********/
29       }
30     }
31   }
32 }
```

Figure 5.6: UPC parallel loop after cuts

of the shared reference (step 22). The linearized block index of the shared reference is computed using this position. Finally, the owner is computed using the linearized block index and the number of threads, as seen in Equation 5.23. Note that since the position of the shared reference is relative to `MYTHREAD`, the owner is also relative to `MYTHREAD`. The shared reference map is a data structure that associates relative thread IDs with shared references that are owned by the ID. Thus, each shared reference is added to the shared reference map using the shared reference owner as an index.

Shared Reference	Region Position	Displacement Vector	Reference Position	Linearized Block Index	Owner
A1	$[0, 0]$	$[0, 0]$	$[0, 0]$	0	`MYTHREAD`
B1	$[0, 0]$	$[0, 1]$	$[0, 1]$	0	`MYTHREAD`
C1	$[0, 0]$	$[0, 14]$	$[0, 14]$	3	`MYTHREAD`+3
A2	$[0, 2]$	$[0, 0]$	$[0, 2]$	0	`MYTHREAD`
B2	$[0, 2]$	$[0, 1]$	$[0, 3]$	0	`MYTHREAD`
C2	$[0, 2]$	$[0, 14]$	$[0, 16]$	4	`MYTHREAD`
A3	$[0, 3]$	$[0, 0]$	$[0, 3]$	0	`MYTHREAD`
B3	$[0, 3]$	$[0, 1]$	$[0, 4]$	1	`MYTHREAD`+1
C3	$[0, 3]$	$[0, 14]$	$[0, 17]$	4	`MYTHREAD`

Table 5.1: Calculations to determine the owner of shared references

Table 5.1 shows the calculations the compiler performs to determine the relative owner of the shared references in Figure 5.6. The shared references A1, B1, and C1 in region R_2 are computed to have positions $[0, 0]$, $[0, 1]$ and $[0, 14]$ based on the position of R_2 ($[0, 0]$) and the displacement vectors for the shared references ($[0, 0]$, $[0, 1]$, and $[0, 14]$). The owner of the shared references are then computed using the reference positions and Equation 5.23. References A1 and B1 are determined to be owned by `MYTHREAD` while reference C1 is owned by `MYTHREAD`+3. The shared reference map is then updated by adding A1 and B1 to the shared reference list associated with `MYTHREAD` and adding C1 to the shared reference list associated with `MYTHREAD`+3. The value of `MYTHREAD` is replaced with the numeric value 0 when building the map. Thus, all shared references owned by `MYTHREAD` will be associated with relative thread 0. Similarly, C1 will be associated with relative thread 3 in the shared reference map. Note that the shared reference map does not indicate the absolute owner of the shared reference but rather the relative owner from the executing thread. This may be a subtle distinction, but the shared reference map cannot be used to determine the thread that owns the shared reference at compile time. That is, the shared reference map does not say that C1 is always owned by the UPC thread with ID 3. Figure 5.7 shows the

shared reference map computed by the compiler for Figure 5.6.

Thread		Shared References
0	$\rightarrow$	A1, A2, A3, B1, B2, C2, C3
1	$\rightarrow$	B3
2	$\rightarrow$	ϕ
3	$\rightarrow$	C1

Figure 5.7: Shared-reference map after locality analysis

5.2.3 Discussion

When an integer affinity test that contains only the loop induction variable is used in the **upc_forall** loop, we will assume that the blocking factor of the shared array is 1. Thus, any shared-array reference in the loop with a blocking factor not equal to 1 is not analyzed by the locality analysis algorithm. When the unmodified loop induction variable is used as the affinity test, a thread T_i will execute iterations i, $i + \texttt{THREADS}$, $i + (2 * \texttt{THREADS})$ and so forth. Similarly, when a blocking factor of 1 is used on a shared array A, thread i will own elements i, $i + \texttt{THREADS}$, $i + (2 * \texttt{THREADS})$ and so forth. Thus, when the loop induction variable is used as the affinity test, any access inside the loop to a shared array with a blocking factor of one will be local, making this assumption safe. The loop induction variable used in the affinity test corresponds to $A_s[i]$ from the properties in Section 5.1 and the shared references in the loop body still correspond to $A_s[j]$.

The locality analysis uses the properties in Chapter 5.1 to determine the relative thread ID of the shared objects. If the compiler cannot determine the difference between i and j in these properties (*i.e.* the distance between the affinity test and the shared reference is not known at compile time) the corresponding reference is not a candidate for locality analysis. This situation occurs when a compile-time unknown value is used to create one of the indices. For example, if the shared-array element is accessed with an index of $i + k$ where k is not known at compile time, the locality analysis cannot determine the relative owner of the shared reference. It is possible to extend the locality analysis to perform symbolic analysis on the difference between the references; however, this has been left as future work.

The properties in Chapter 5.1 also assume that the blocking factor is known at compile time (a requirement of the UPC language). However, the locality analysis is still applicable if the blocking factor is not known at compile time. In this case, the compiler has to perform symbolic analysis, similar to the symbolic analysis required if the difference between

the shared references is not known at compile time. Thus, this analysis is applicable to other PGAS languages, even if they relax the constraint of specifying the blocking factor at compile time.

Consider a single-blocked, two-dimensional shared array in the current UPC specification declared as **shared** [BF] **int** A[ROWS][COLUMNS]. The blocking factor applies to the elements of the innermost dimension of the array (the analysis assumes the blocking factor for the outer dimensions is one). However, if BF is a multiple of one of the inner dimension (*i.e.* BF = 2∗COLUMNS), the block will wrap around the outer dimension, creating a multi-blocked array. In this situation, the compiler treats the shared array as a multi-blocked shared array where the blocking factor of the outer dimension is computed as $\texttt{BF}_{outer} = \frac{\texttt{BF}_{inner}}{\texttt{COLUMNS}}$. Note that the compiler only converts single-blocked arrays to multi-blocked arrays for the purpose of locality analysis and only when the blocking factor is a multiple of the inner dimension (*i.e.* $\texttt{BF \% COLUMNS} = 0$).

On the other hand, if the blocking factor in the innermost dimension is greater than the size of the innermost dimension, but is not a multiple of the size of the innermost dimension (*i.e.* $\texttt{BF} > \texttt{COLUMNS}$ and $\texttt{BF \% COLUMNS} \neq 0$), the array is not considered for locality analysis. This is a limitation in the current algorithm because cuts are computed and generated on each dimension (from outermost to innermost) independently. In the above example it is necessary to cut the outer dimension in addition to the inner dimension, however that is not discovered until the outer dimension has already been processed. One possible solution for this is to change the algorithm to first compute all cuts and then generate the corresponding code. This modification would allow the algorithm to handle this situation. However, the need for this modification has not been found in any of the benchmarks at this point.

The shared reference map contains an additional entry, *UNKNOWN*, that represents all shared references that could not be correctly sorted by the locality analysis. For example, if the shared reference has a distance that cannot be computed at compile time (*e.g.,* A[i+X]) the locality analysis cannot determine the relative thread that owns the shared reference and thus it is associated with the *UNKNOWN* entry in the map.

5.3 Chapter Summary

This chapter presented a new compiler analysis that will identify the relative thread ID that owns a specific shared reference given a shared reference with a known locality. The affinity test in a **upc_forall** loop provides a reference with known locality, thereby making

this analysis applicable to parallel loop nests in UPC. After the analysis is complete, the compiler has a shared reference map that maps shared references to the relative thread ID that owns the reference. This information can be used by various locality optimizations to optimize the code generated by the compiler, as discussed in the next chapter.

Chapter 6

Shared-Object Locality Optimizations

The Shared-Object Locality Analysis described in Chapter 5 builds a shared reference map that associates shared references with the relative thread ID that owns them. This information is used by the compiler to perform four different locality optimizations on shared objects: (i) privatizing local shared objects accesses; (ii) coalescing remote shared-object accesses owned by the same thread; (iii) scheduling remote shared-object accesses to overlap communication with computation; and (iv) updating remote shared-object accesses. Each of these optimizations is new within the context of UPC and PGAS programming models. While each optimization is presented for UPC specifically, they are all applicable to any language that employs a PGAS programming model.

6.1 Shared-Object Access Privatization (SOAP)

Shared-object accesses that are owned by the accessing thread can be optimized to bypass the RTS system. This is done by converting the *handle* used to describe the shared object into a *direct pointer access* that uses the local address of the shared object.

Figure 6.1(a) shows the STREAM triad kernel written in UPC using an integer affinity test. Since the default (cyclic) blocking factor is used to distribute the three shared arrays, the integer affinity test is used to parallelize the kernel. Figure 6.1(b) shows the naive transformation of shared accesses into calls to the RTS to access the shared arrays. This transformation results in three function calls, two to read the values of the b and c shared array and a third to write the computed value to a.

Figure 6.2 shows the rate of data transfer to memory (memory bandwidth) measured by the STREAM triad kernel using the naive transformation on a 64-way POWER5 machine

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; i) {
    a[i] = b[i] + SCALAR*c[i];
  }
}
```

(a) Original UPC code

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; i) {
    __xlupc_deref_array(c_h, tmp1, i);
    __xlupc_deref_array(b_h, tmp2, i);
    tmp3 = tmp2 + 3.0*tmp1;
    __xlupc_assign_array(a_h, tmp3, i);
  }
}
```

(b) After transformations

Figure 6.1: UPC STREAM triad kernel

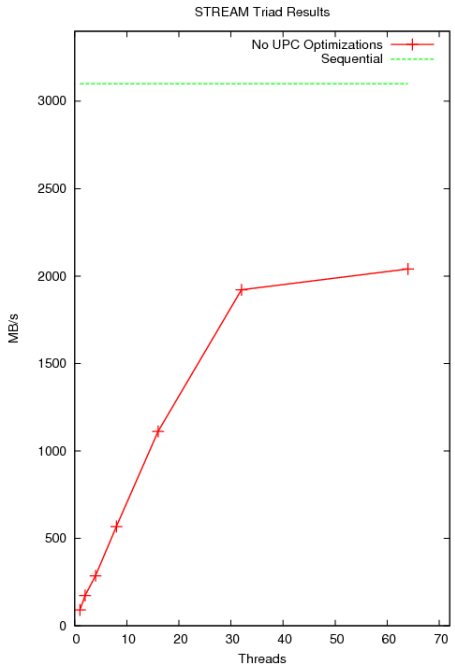


Figure 6.2: STREAM triad results using the naive transformations

(1.6 GHz processors) with 512 GB of RAM running AIX 5.3. The memory bandwidth for the sequential version of STREAM triad kernel is 3100 MB/s. While the benchmark does scale nicely as the number of UPC threads increase, the overall results are significantly lower than the sequential version.

6.1.1 Algorithm

To perform Shared-Object Access Privatization (SOAP) the compiler converts a fat pointer into a traditional C pointer. This conversion requires support from the RTS to retrieve the base address of the shared object in memory. A direct pointer access is then performed using the base address and an offset computed using the original index into the shared array.

PRIVATIZESOA($ForallLoop\ L_{forall}$, $SharedReferenceMapArray\ \mathcal{A}$)
1.**foreach** local thread T
2. $\mathcal{M}_T \leftarrow \mathcal{A}[T]$
3. **foreach** shared reference r_s in $\mathcal{M}_T$
4. $stmt \leftarrow$ statement containing r_s
5. $r_{handle} \leftarrow$ SVD handle for r_s
6. $L^{Preheader}.Add(r_{address} \leftarrow$ `BaseAddress(`r_{handle}`)`$)$
7. $lix \leftarrow$ Linearized IVs used by r_s
8. $offset \leftarrow (courseof(r, lix) * r_{blocking\_factor} + phaseof(r, lix)) * r_{elt\_sz}$
9. **if** r_s is a *def* **then**
10. $data \leftarrow$ data to store to r_s
11. **if** $data.DataType$ is intrinsic **then**
12. $newStmt \leftarrow store_{ind}(r_{address} + offset, data)$
13. **else**
14. $newStmt \leftarrow$ `memcpy`$(r_{address} + offset, data, data.size)$
15. **end if**
16. **else**
17. $dst \leftarrow$ location to store data from r_s
18. **if** $r_s.DataType$ is intrinsic **then**
19. $newStmt \leftarrow load_{ind}(r_{address} + offset, dst)$
20. **else**
21. $newStmt \leftarrow$`memcpy`$(dst, r_{address} + offset, data.size)$
22. **end if**
23. **end if**
24. $stmt \leftarrow newStmt$
25. **end for**
26.**end for**

Figure 6.3: Algorithm to privatize local shared-object accesses

The PRIVATIZESOA algorithm in Figure 6.3 uses the $SharedReferenceMapArray\ \mathcal{A}$ generated by locality analysis to identify local shared references. For all architectures, `MYTHREAD` will always be local and thus $\mathcal{A}[0]$ will always be evaluated. For hybrid architectures, threads 0 to $\mathcal{T}_{AP}$ will be local and thus considered in step 1. For SMP architectures, all threads map to the same address partition and thus step 1 will iterate through all threads.

Step 5 obtains the handle (fat pointer) used by the RTS to identify the shared reference r_s. Step 6 inserts a call to a function in the RTS to obtain the base address of r_s using the SVD. The base-address initialization is placed in the loop preheader. The loop preheader contains statements that should only be executed if the loop body executes but do not need to be executed in every iteration of the loop. It is typically used to initialize loop invariant variables.

In steps 7 and 8 the compiler generates code to compute the offset from the base address at runtime. First, the linearized index is computed based on all induction variables used by r_s (step 7). Next, code to compute the offset is generated using Equation 4.10 from Section 4.3.4.

The algorithm then determines the type of reference that r_s represents. If r_s is a definition of (store to) a shared object, the data stored to r_s is obtained (step 10). If the data type of the reference (*i.e.* the type of the shared array) is an intrinsic, a new statement containing an indirect store is generated to store the data to the memory location $r_{address} + offset$ (step 12). If the data type is not intrinsic, a new statement containing a call to `memcpy` is used to copy the data to $r_{address} + offset$ (step 14).

If r_s is a *use* of (load from) a shared object, the destination of the load is obtained. If the type of the shared data represented by r_s is intrinsic, an indirect load is used to obtain the data, which is stored to the destination (step 19). If the type is not intrinsic, `memcpy` is used to copy the shared data from $r_{address} + offset$ to the destination (step 21). Finally, step 24 replaces the statement containing r_s with the new statement.

```
1  #define SCALAR 3.0
2  shared double a[N];
3  shared double b[N];
4  shared double c[N];
5
6  void StreamTriad() {
7    int i;
8    aBase = __xlupc_base_address(a_h);
9    bBase = __xlupc_base_address(b_h);
10   cBase = __xlupc_base_address(c_h);
11   upc_forall(i=0;i<N;i++;&a[i]) {
12     aOffset = ((((i%THREADS)%threadsPerNode)*local_size) + (i/THREADS))*8;
13     bOffset = ((((i%THREADS)%threadsPerNode)*local_size) + (i/THREADS))*8;
14     cOffset = ((((i%THREADS)%threadsPerNode)*local_size) + (i/THREADS))*8;
15     *(aBase+aOffset) = *(bBase+bOffset) + SCALAR*(*(cBase+cOffset)) ;
16   }
17 }
```

Figure 6.4: UPC STREAM triad kernel after PRIVATIZESOA

Figure 6.4 shows the code generated when the PRIVATIZESOA algorithm is applied to

the STREAM kernel in Figure 6.1(a). The shared reference map generated by the locality analysis will show that all three references map to `MYTHREAD`. Each of these references is then considered in turn. The base addresses for the three shared arrays, a, b, and c arrays are obtain on lines 8, 9, and 10. The base addresses are loop invariant and thus the call is placed outside of the loop. The offsets from the base are computed on lines 12, 13 and 14. The offsets are based on the induction variable i and thus must remain inside the loop. However, the offset computations contain both division and modulo operations and thus are later optimized by the division and modulo strength reduction optimization. The compiler was able to simplify the offset computations, instead of generating the more general form presented in Equation 4.10, because the blocking factor for the shared arrays is 1. Finally, the local shared-memory is directly accessed using the base address and offsets on line 15.

Three offset computations are inserted (for aOffset, bOffset, and cOffset) because the algorithm is run on each shared reference independently. However, since the three shared arrays have the same blocking factor and are accessed at the same index, these offsets will always be the same. A subsequent pass of common sub-expression elimination will remove the redundant computations of bOffset and cOffset.

6.1.2 Results

The STREAM benchmark was used to measure the impact of privatizing shared-object accesses. The benchmark was run on the same machine as used to collect the results presented in Figure 6.2. In all experiments, all UPC-specific optimizations are disabled except for the privatization optimization being tested. Thus, the **upc_forall** loop in the STREAM triad kernel is not optimized. The STREAM triad kernel using an *integer* affinity test was used in these experiments because the unoptimized integer affinity test has a much lower overhead than the unoptimized pointer-to-shared affinity test.

Figure 6.5 shows the memory bandwidth measured when the STREAM triad kernel is compiled with and without the PRIVATIZESOA algorithm. These results demonstrate that there is a significant benefit to privatizing local shared references.

The performance of the optimized UPC STREAM Triad kernel run with 1 thread is approximately 11% worse than the sequential C performance. This is due to the extra computations required to determine the offsets to use in the direct pointer access. While the compiler is able to optimize these computations and reduce the overhead, it cannot remove them completely. When the UPC version is increased to 2 threads, the measured memory bandwidth increases to approximately 3085 MB/s resulting in roughly the same

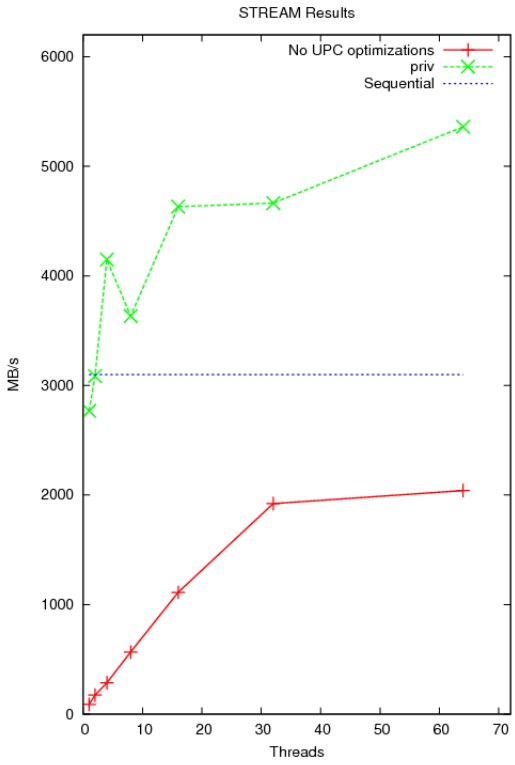


Figure 6.5: Transfer rate for the UPC STREAM triad kernel

performance as the sequential version. Thus, the extra work required by the additional offset computations is mitigated by the addition of a second thread. As more threads are added, the performance improves.

6.1.3 Discussion

The results in Figure 6.5 demonstrate a significant performance improvement by privatizing shared-object accesses. However, since the **upc_forall** loop is not optimized, it contains a branch that is executed by every thread in every iteration of the loop. Not only does this branch add instructions that must be executed in every iteration of the loop, it hinders the ability of other optimizations to further optimize the loop. Thus, in order to achieve good performance, the compiler must be able to remove this branch in addition to privatizing the shared-object accesses.

The implementation of the PRIVATIZESOA algorithm was designed to focus on shared arrays. In principle it is possible to privatize accesses to dynamically allocated storage represented using pointers to shared data. However, before the compiler can privatize such accesses it must verify that the pointer points to shared data that is distributed across all threads. The algorithm to privatize shared accesses will not compute the offset positions correctly for shared objects that are allocated to a single node (*e.g.,* structures).

The PRIVATIZESOA algorithm does not change the order of accesses to shared data and thus is always safe to perform, even in the presence of aliasing. That is, if a shared array A_s is aliased with other shared symbols, it is always safe to privatize accesses to A_s because the privatization does not change the order of the accesses relative to other shared accesses.

6.2 Shared-Object Access Coalescing (SOAC)

Every shared-object access is translated into a function call to the RTS. On distributed and hybrid architectures, access to shared objects located on a different node (*i.e.* remote accesses) result in the exchange of messages between the accessing thread and the owner thread. Thus, there is a direct correlation between the number of remote shared references and the number of messages that are sent during the execution of a program on a distributed or hybrid architecture. When two or more shared objects are owned by the same thread, accesses to the shared objects can be combined into a single access. When the shared objects are members of the same shared array, this optimization is called *message coalescing*; when the shared objects are members of different shared arrays, this optimization is called

message aggregation [34]. This work focuses on message coalescing, thus requiring all shared-objects to be members of the same shared array.

If the accesses are uses of the shared objects, a coalesce RTS function call retrieves the shared objects from the owning thread and places them into a temporary buffer owned by the accessing thread. The uses are then modified to access the temporary buffer. Similarly, if the accesses are definitions, the accesses are modified to write to the temporary buffer owned by the accessing thread. A single coalesce RTS function call then updates all of the shared objects on the owning thread. The goal of SOAC is to reduce the number of accesses to remote shared objects, thereby reducing the number of messages required during the execution of the program on a distributed or hybrid architecture. The SOAC optimization is responsible for (i) identifying opportunities for coalescing; (ii) managing the temporary buffers, including modifications of shared references to read-from/write-to the buffers; and (iii) inserting calls to the coalesce RTS functions.

```
1  shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
2  shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
3  int Sobel() {
4    int i, j, gx, gy;
5    double gradient;
6    for (i=1; i < ROWS-1; i++) {
7      upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
8        gx =   (int) orig[i-1][j+1] - orig[i-1][j-1];
9        gx += ((int) orig[i][j+1]   - orig[i][j-1]) * 2;
10       gx +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
11       gy =   (int) orig[i+1][j-1] - orig[i-1][j-1];
12       gy += ((int) orig[i+1][j]   - orig[i-1][j]) * 2;
13       gy +=  (int) orig[i+1][j+1] - orig[i-1][j+1];
14       gradient = sqrt((gx*gx) + (gy*gy));
15       if (gradient > 255) gradient = 255;
16       edge[i][j] = (BYTE) gradient;
17     }
18   }
19 }
```

Figure 6.6: UPC Sobel benchmark

Figure 6.6 shows the Sobel edge detection kernel in UPC. The two shared arrays representing the image are blocked such that each row has affinity with a single UPC thread. The kernel contains a total of 13 accesses to shared arrays (12 to orig and one to edge). Figure 6.7 shows the resulting code generated by the compiler without coalescing.

Figure 6.8 shows the results of running the Sobel benchmark on a 4000x4000 image on a hybrid system using up to 4 nodes and 1 thread-per-node (*i.e.* a purely distributed system). The sequential time of a C implementation of Sobel for this image size is 1.217s.

Table 6.1 shows the number of local and remote accesses performed for the Sobel ker-

```
1  shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
2  shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
3  int Sobel() {
4    int i, j, gx, gy;
5    double gradient;
6    for (i=1; i < ROWS-1; i++) {
7      upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
8        __xlupc_deref_array(orig_h, tmp1, (i-1)*COLUMNS+j+1);
9        __xlupc_deref_array(orig_h, tmp2, (i-1)*COLUMNS+j-1);
10       __xlupc_deref_array(orig_h, tmp3, i*COLUMNS+j+1);
11       __xlupc_deref_array(orig_h, tmp4, i*COLUMNS+j-1);
12       __xlupc_deref_array(orig_h, tmp5, (i+1)*COLUMNS+j+1);
13       __xlupc_deref_array(orig_h, tmp6, (i+1)*COLUMNS+j-1);
14       __xlupc_deref_array(orig_h, tmp7, (i+1)*COLUMNS+j-1);
15       __xlupc_deref_array(orig_h, tmp8, (i-1)*COLUMNS+j-1);
16       __xlupc_deref_array(orig_h, tmp9, (i+1)*COLUMNS+j);
17       __xlupc_deref_array(orig_h, tmp10, (i-1)*COLUMNS+j);
18       __xlupc_deref_array(orig_h, tmp11, (i+1)*COLUMNS+j+1);
19       __xlupc_deref_array(orig_h, tmp12, (i-1)*COLUMNS+j+1);
20
21       gx =   (int) tmp1 - tmp2];
22       gx += ((int) tmp3 - tmp4) * 2;
23       gx +=  (int) tmp5 - tmp6;
24       gy =   (int) tmp7 - tmp8;
25       gy += ((int) tmp9 - tmp10) * 2;
26       gy +=  (int) tmp11 - tmp12;
27       gradient = sqrt((gx*gx) + (gy*gy));
28       if (gradient > 255) gradient = 255;
29       tmp13 = (BYTE) gradient;
30       assign_array(edge_h, tmp13, i*COLUMNS+j);
31     }
32   }
33 }
```

Figure 6.7: Transformed Sobel benchmark

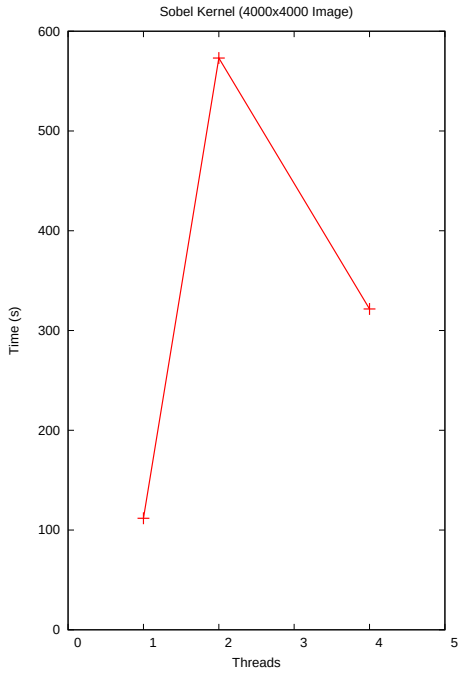


Figure 6.8: Execution time for the Sobel kernel

nel with the same image. The accesses are broken down into assignments (writes) and dereferences (reads). For a single thread, there are no remote accesses because all elements of the shared arrays have affinity with thread 0. Thus, the difference in execution time between the sequential version (1.217s) and the UPC version with 1 thread (111s) is not caused by remote accesses. Instead, this difference is due to the overhead of the parallel loops and the calls to the runtime system to access the shared arrays. These overheads are discussed further in Chapter 7. As the number of nodes increases, the number of remote accesses also increases because the shared arrays are now distributed across more threads. Each of these remote accesses will result in a message being sent across the network interconnect to access the remote data. This overhead is seen in Figure 6.8 as the time for running the Sobel kernel with 2 threads is higher than the time when running on a single thread. Even though the work is distributed among 2 threads, the overhead from accessing remote shared-objects cause the benchmark to run slower. The increase in execution time clearly demonstrates that the cost of communication in a distributed or hybrid system can significantly impact performance.

The access measurements in Table 6.1 include the initialization of the original image

Nodes	Local Accesses			Remote Accesses		
	Assign	Deref	Total	Assign	Deref	total
1	31984004	143872032	175856036	0	0	0
2	23984004	39968008	63952012	8000000	103904024	111904024
3	21320004	37304008	58624012	10664000	106568024	117232024
4	19984004	35968008	55952012	12000000	107904024	119904024

Table 6.1: Remote assigns and derefs

and the serialization of the edge (final) image. Both of these operations must be performed by a single thread and thus some of the accesses will be remote. On the other hand, the execution times presented in Figure 6.8 only measure the time spent in the Sobel kernel and does not include the initialization and serialization of the images. Thus, the number of local and remote accesses in Table 6.1 do not correspond directly to the execution times in Figure 6.8. However, the instrumentation results clearly show a large number of remote accesses.

6.2.1 Algorithm

The interface for coalescing shared object accesses provided by the RTS allows the compiler to specify a shared symbol, a stride to use when collecting shared references, the number of shared references to collect and the temporary buffer to use. Separate calls are used to differentiate between reads and writes. To use this interface, the compiler enforces the following restrictions on coalescing candidates: (i) all candidates must have the same base symbol; (ii) all candidates must be owned by the same thread. (iii) all candidates must have the same type (read or write). Conditions (i) and (ii) ensure that a single (constant) stride can be used to describe all shared references that are coalesced into a single access. Requiring coalesced shared references to have the same symbol simplifies the coalescing in two ways. First, in the function calls to the RTS only one shared symbol needs to be specified. An extension to this algorithm would be to coalesce several shared symbols and offsets into a single message. This extension is left as possible future work. The second advantage is that all of the coalesced data will have the same type and length. This restriction simplifies the conversion of the original shared accesses to correctly use the temporary buffer. Condition (iii) is required because a single coalescing call cannot mix read and write instructions.

Another requirement for coalescing is that two shared references cannot be separated by a synchronization point (*e.g.,* a **upc_barrier** or a strict access). This is required to com-

ply with the memory semantics specified by the UPC Language. As a result, **upc_forall** loops that contain synchronization points are not included in the analysis for shared-object coalescing. This restriction is overly conservative because it is possible to have shared references that can safely be coalesced in a **upc_forall** loop that contains synchronization points. However, coalescing such references would require additional data and control flow analysis to ensure that the coalesced calls do not cross synchronization points.

The algorithm iterates through all remote threads, collecting shared references as candidates for coalescing. For each remote thread T, the map generated by locality analysis is obtained (step 2). This map contains all of the shared references that the compiler can prove that T owns. For each type of reference (use or def) and each shared symbol s, a list of coalescing candidates is created using the COALESCECANDIDATES routine (step 5). This routine collects all shared references with the same shared symbol and of the specified type, eliminating *conflicting* and *aliased* references. If a loop contains a use and a definition of the same shared variable, the corresponding shared references *conflict*. This algorithm only considers non-conflicting shared references to simplify placement of the coalesced access (conflicting shared references cause data dependencies that must be preserved when placing the coalesced shared-object access). This restriction can be relaxed, however, coalescing conflicting shared references will require a context sensitive placement algorithm that preserves data dependencies. If a shared reference is aliased to another shared reference (through a pointer-to-shared) then it is not safe to perform coalescing because coalescing can change the order of accesses to shared objects. Thus, all aliased shared references are also eliminated by the COALESCECANDIDATES routine. This is also an overly restrictive constraint and could be relaxed in the future using more detailed control flow and data flow analysis.

After step 8, $RefList_T^{ref\_type}$ contains two or more shared references that all have the same type (def or use), all access the same base symbol and are all owned by the same thread. It is not necessary for all shared references in $RefList_T^{ref\_type}$ to have the same stride between them. Thus the next step of the algorithm is to determine the best stride to use when coalescing the shared references in $RefList_T^{ref\_type}$. This is done by performing a pair-wise comparison between all shared references in $RefList_T^{ref\_type}$ to determine the stride between them (steps 10 to 13). $StrideCounts$ is a hash table where the stride s is used as a key and the hash value is the number of times s occurs. The MAXIMUM function (step 14) returns the position containing the maximum value in $strideCounts$. This maximum value corresponds to the stride that occurred the most among all the shared

```
COALESCESOA(ForallLoop L_forall, SharedReferenceMapArray A)
1.foreach remote thread T
2.      M_T ← A[T]
3.      foreach ref_type ∈ def, use
4.          foreach shared symbol s
5.              RefList_T^ref_type ← COALESCECANDIDATES(ref_type, s, M_T)
6.              if || RefList_T^ref_type || < 2 then
7.                continue
8.              end if
9.              Initialize strideCounts to 0
10.             foreach pair of unique references r_i and r_j in RefList_T^ref_type
11.                 s ← |r_i − j_j|
12.                 Increment strideCounts[s]
13.             end for
14.             stride ← MAXIMUM(strideCounts)
15.             initCands ← φ
16.             CoalesceList ← initCands
17.             foreach pair of shared references r_i and r_j in RefList_T^ref_type
18.                 if |r_i − r_j| = stride then
19                    foreach candList in CoalesceList
20.                       if r_i ∈ candList then
21.                         candList ← candList + r_j
22.                       elsif r_j ∈ candList then
23.                           candList ← candList + r_i
24.                       else
25.                           newList ← r_i
26.                           newList ← newList + r_j
27.                           CoalesceList ← CoalesceList + newList
28.                       end if
29.                   end for
30.               end if
31.           end for
32.           foreach candList ∈ CoalesceList
33.               if ||candList|| < 2 then
34.                 continue
35.               end if
36.               candList ← SORTCANDIDATES(candList, s)
37.               Allocate tmpBuffer with ||candList|| elements
38.               if ref_type is def then
39.                 InitBuffer ← COALESCEDEF(tmpBuffer, candList)
40.                 InsertPos ← BODYEND(L_forall)
41.                 Insert InitBuffer after InsertPos
42.               else
43.                   InitBuffer ← COALESCEUSE(tmpBuffer, candList)
44.                   InsertPos ← BODYBEGIN(L_forall)
45.                   Insert InitBuffer before InsertPos
46.               end if
47.               foreach shared reference r_s in candList
48.                     r_s.base ← tmpBuffer
49.                     r_s.offset ← position of r_s in candList
50.                     Mark r_s as not shared
51.               end for
52.           end for
53.       end for
54.   end for
55. end for
```

Figure 6.9: Algorithm to coalesce remote shared-object accesses

references in $RefList_T$. Selecting the stride with the most occurrences ensures that the maximum number of shared references in $RefList_T^{ref\_type}$ will be coalesced.

The next phase of the algorithm is to sort all of the shared references into groups of candidates that can be coalesced together. This is done using a union-find based algorithm where two references r_i and r_j with the desired stride are placed into an existing list (steps 21 and 23) or added to a new list (steps 24 to 27). A *candList* is a list of references and *CoalesceList* is a list of *candList*s. It is not possible for a reference to appear in two different candidate lists because $RefList_T^{ref\_type}$ did not contain any duplicate references. Similarly, every reference in $RefList_T^{ref\_type}$ is guaranteed to be placed in one candidate list.

The final phase of the algorithm traverses each list of candidates, creating the coalesced calls and remapping the shared references. No coalescing opportunities exist for lists of size 1 and thus they are ignored (step 33).

Step 36 sorts the candidates in $candList$ such that for every two adjacent candidates $cand_i$ and $cand_{i+1}$ in $candList$, the distance between $cand_i$ and $cand_{i+1}$ is $stride$ (*i.e.* $cand_{i+1} - cand_i = stride$). In principle it is possible to sort the candidate list as it is being created (*i.e.* insert candidates so the list remains sorted) however for implementation reasons this was added as an explicit step. This sorting is done to facilitate the remapping of shared references into the temporary buffer since the position of the reference within the group will determine the position of the data in the temporary buffer. Upon completion of step 36, the candidates in $candList$ are arranged in ascending order based on the distance between the shared references.

Next, the temporary buffer is allocated based on the number of shared references in the candidate list (step 37). The call to the coalesce RTS function is then created. The generated call contains (i) the starting position used for coalescing (the first entry in $candList$), (ii) the stride between candidates, (iii) the number of entries to retrieve, (iv) a temporary buffer containing the results. When ref_type is use, a GET call is generated and placed at the beginning of the loop body; when ref_type is def a PUT call is generated and placed at the end of the loop body. All shared references in $candList$ are then remapped to use this temporary buffer. If ref_type is use, the data will be loaded from $tmpBuffer$; if ref_type is def, the data will be stored to $tmpBuffer$. Since the candidates in $candList$ are sorted in ascending order, the position of the candidate in $candList$ corresponds to the position of the data in the temporary buffer.

Consider again the Sobel kernel from Figure 6.6 compiled for 2 threads and 2 nodes (hybrid architecture with 1 thread per node) and an image size of 4000x4000 (*i.e.* ROWS

Thread		Shared References
0	$\rightarrow$	$orig_{i,j-1}$, $orig_{i,j}$, $orig_{i,j+1}$, $edge_{i,j}$
1	$\rightarrow$	$orig_{i-1,j-1}$, $orig_{i-1,j}$, $orig_{i-1,j+1}$, $orig_{i+1,j-1}$, $orig_{i+1,j}$, $orig_{i+1,j+1}$

Figure 6.10: *SharedReferenceMapArray* for the Sobel kernel

=4000, COLUMNS=4000). Figure 6.10 shows the *SharedReferenceMapArray* created by the locality analysis. Since there is only one remote thread, the outer foreach loop of the CoalesceSOA algorithm will only execute once (step 1). The shared reference map for thread 1, $\mathcal{M}_1$, will contain six shared references corresponding to the shared array accesses: orig[i−1][j−1], orig[i−1][j], orig[i−1][j+1], orig[i+1][j−1], orig[i+1][j], and orig[i+1][j+1]. Since all shared references are uses and have the same base symbol $RefList_T^{ref\text{-}type}$ will contain these six references after step 5.

stride	count
1	4
2	2
7998	1
7999	2
8000	3
8001	2
8002	1

Figure 6.11: *strideCounts* array created by CoalesceSOA

Figure 6.11 shows the stride counts array computed in steps 10 to 13. The stride corresponds to the distance, in number of array elements, between two shared references. The stride is computed by *flattening* the indices in the shared references and subtracting them. For example, consider the shared references orig[i−1][j−1] and orig[i+1][j+1]. The *flattened* indices for these two references are (i−1)∗COLUMNS + j−1 and (i+1) ∗COLUMNS + j+1 respectively. Subtracting these two indices results in a stride of 2∗ COLUMNS + 2, where COLUMNS = 4000. Thus, the stride between orig[i−1][j−1] and orig[i+1][j+1] is 8002.

Since $strideCounts[1]$ contains the highest value, 1 is selected as the stride for coalescing. Two candidate lists are then created (steps 17 to 31) where $candList_1$ = [orig[i−1][j−1] orig[i−1][j] orig[i−1][j+1]] and $candList_2$ = [orig[i+1][j−1] orig[i+1][j] orig[i+1][j+1]]. The first candidate list is then sorted and a temporary buffer containing three elements is created. The data type and length of the elements in the temporary

buffer are based on the data type and length of the shared references. Since the references in $candList_1$ are *uses*, a coalesced GET call is created and placed at the beginning of the loop body. The uses of the references in $candList_1$ are replaced with references to the temporary buffer using their position in the candidate list to determine the offset into the temporary buffer. For example, $candList_1[1]$ is orig[i−1][j] so orig[i−1][j] is replaced with tmpBuffer[1].

```
1  shared [COLUMNS] BYTE orig[ROWS][COLUMNS];
2  shared [COLUMNS] BYTE edge[ROWS][COLUMNS];
3  int Sobel() {
4    int i, j, gx, gy;
5    double gradient;
6    for (i=1; i < ROWS-1; i++) {
7      upc_forall (j=1; j < COLUMNS-1; j++; &orig[i][j]) {
8        __xlupc_coalesce_get(tmp1, orig, ((i-1)*COLUMNS)+j-1, 3, 1);
9        __xlupc_coalesce_get(tmp2, orig, ((i+1)*COLUMNS)+j-1, 3, 1);
10       gx = (int) tmp1[2] - tmp1[0];
11       gx += ((int) orig[i][j+1]   - orig[i][j-1]) * 2;
12       gx += (int) tmp2[2] - tmp2[0];
13       gy = (int) tmp2[0] - tmp1[0];
14       gy += ((int) tmp2[1] - tmp1[1]) * 2;
15       gy += (int) tmp2[2] - tmp1[2];
16       gradient = sqrt((gx*gx) + (gy*gy));
17       if (gradient > 255) gradient = 255;
18       edge[i][j] = (BYTE) gradient;
19     }
20   }
21 }
```

Figure 6.12: UPC Sobel edge detection kernel after COALESCESOA

Steps 19 to 53 are then repeated for $candList_2$. Figure 6.12 shows the Sobel kernel after coalescing the remote shared accesses. Note that the shared array accesses on lines 11 and 18 are not modified. These accesses are identified as local accesses by the locality analysis and thus placed in the shared reference map for relative thread 0. As a result they are not considered for coalescing.

6.2.2 Results

The impact of coalescing shared-object accesses was measured using the Sobel kernel. The benchmark was compiled and run with one thread-per-node (distributed architecture) on a 4-node POWER5 cluster. Each node contains 16 POWER5 cores (1.9 GHz), 64GB of RAM and runs AIX 5.3. All UPC-specific optimizations are disabled except for the coalescing optimization being tested.

Figure 6.13 shows the execution times of the Sobel kernel when compiled with and without the COALESCESOA algorithm. We observe a 33% decrease in execution time on

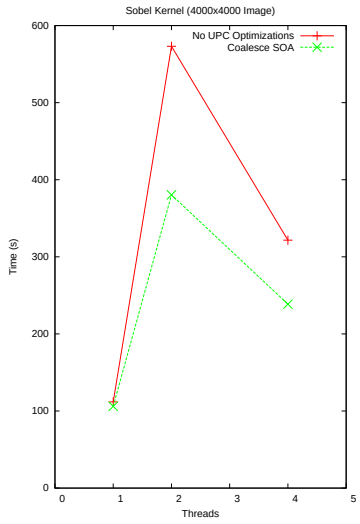


Figure 6.13: UPC Sobel kernel with coalescing

2 threads and a 25% decrease in execution time on 4 when SOAC is applied. Figure 6.14 shows the decrease in the number of remote dereferences when SOAC is applied. These results indicate that the COALESCESOA algorithm is able to reduce the number of remote shared-object accesses thereby increasing the performance of the Sobel benchmark.

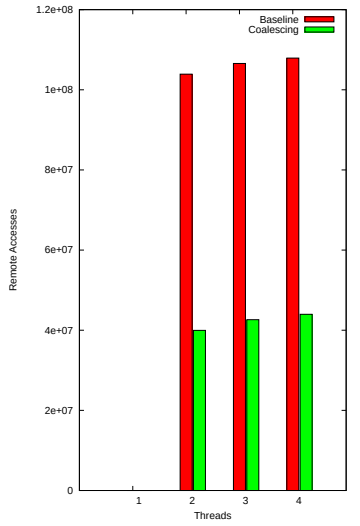


Figure 6.14: Remote dereferences with coalescing

6.2.3 Discussion

In principle, it is possible to create opportunities for coalescing by unrolling loops. However, the loop unrolling mechanism in TPO is currently disabled. Once the loop unrolling mechanism has been tested with **upc_forall** loops, work will be done to use it to create additional coalescing opportunities.

There are 2 costs associated with coalescing that could potentially impact performance: (i) allocation and management of temporary buffers; (ii) potential for creating large messages (which could impact performance of the network). Both of these costs will occur if too many accesses are coalesced into a single access. At this point we have not found any cases in existing benchmarks where coalescing all possible opportunities causes performance to degrade. As such, no heuristics have been implemented to control the use of coalescing. However, heuristics will become necessary once the compiler is able to unroll loops to create coalescing opportunities. The heuristics to unroll loops and create coalescing opportunities will have to take into account the potentially negative costs associated with coalescing in order to prevent a negative performance impact.

As mentioned previously, message coalescing requires the shared objects to be members of the same shared array. When the shared objects are members of different arrays, *message aggregation* can be used to combine multiple shared-object accesses that map to the same thread into a single access. The shared reference map contains all of the necessary information in order for the compiler to generate aggregated calls to the RTS. However, an RTS interface to represent aggregated accesses has not been established at this point and is being considered for future work.

6.3 Shared-Object Access Scheduling

The Shared-Object Access Scheduling (SOAS) optimization attempts to overlap the communication required to move shared objects between address partitions with other (unrelated) computation. The motivation is to reduce the amount of time a processor spends idle waiting for the communication to complete.

Consider the synthetic UPC benchmark in Figure 6.15. This benchmark performs a sum on private data and then updates the sum based on the value of a remote shared object. Finally, a local access is performed to write the sum to another shared object. Figure 6.16 shows the code that is generated by the existing UPC transformations. The local sum is computed on lines 8 to 10 and then the value of the remote data is retrieved (line 11). The

```
1  shared double MySharedData[SHARED_DATA_SIZE];
2  shared double RemoteData[SHARED_DATA_SIZE];
3  double MyPrivateData[PRIVATE_DATA_SIZE];
4  void Compute() {
5    int i, j, k;
6    double sum=0.0;
7    upc_forall (i=0; i < SHARED_DATA_SIZE; i++; &MySharedData[i]) {
8      for (j=0; j < PRIVATE_DATA_SIZE; j++) {
9        sum += MyPrivateData[j];
10     }
11     MySharedData[i] = sum * RemoteData[(i+1)%(SHARED_DATA_SIZE)];
12   }
13 }
```

Figure 6.15: Synthetic UPC benchmark to illustrate scheduling shared-object accesses

__xlupc_deref_array call will return when the remote value has been retrieved, at which point the execution continues. In this example, if the retrieval of the remote shared data is initiated before the loop on line 8, then the time taken to retrieve the data could be overlapped with the computation of the local sum.

```
1  shared double MySharedData[SHARED_DATA_SIZE];
2  shared double RemoteData[SHARED_DATA_SIZE];
3  double MyPrivateData[PRIVATE_DATA_SIZE];
4  void Compute() {
5    int i, j, k;
6    double sum=0.0;
7    upc_forall (i=0; i < SHARED_DATA_SIZE; i++; &MySharedData[i]) {
8      for (j=0; j < PRIVATE_DATA_SIZE; j++) {
9        sum += MyPrivateData[j];
10     }
11     __xlupc_deref_array(RemoteData_h, &tmp, (i+1)%SHARED_DATA_SIZE);
12     tmp2 = sum * tmp;
13     __xlupc_assign_array(MySharedData_h, &tmp2, i);
14   }
15 }
```

Figure 6.16: Naive transformation of a synthetic benchmark

There are three restrictions that must be obeyed when scheduling remote shared-object accesses: (i) shared-object accesses cannot move past a synchronization point; (ii) traditional data dependencies must be preserved; (iii) the non-blocking call, the corresponding wait statement and the shared reference must all be control-flow equivalent.

To ensure the first condition, remote shared-object accesses cannot be moved across barriers, strict accesses, or any collective operation that implies synchronization. Similarly, strict accesses are not candidates for remote shared-object scheduling. To ensure that this criteria is met, the remote shared-object access scheduling algorithm works at the scope of a loop nest that does not contain any synchronization points or strict accesses. A tra-

ditional data dependence graph for the loop nest is used to ensure that no traditional data dependencies are violated.

The third condition ensures that the scheduling of shared-object accesses does not cause a shared-object access to occur when it would not have occurred in the original code. The scheduler also ensures that the non-blocking call and the wait call inserted by the compiler match correctly.

When scheduling remote shared-object accesses, the compiler uses non-blocking versions of the calls in the RTS. These calls are referred to as *post* calls because they simply post the required message to the communication subsystem and return immediately. Each post call returns a pointer to a handle that the compiler uses in the subsequent wait call that it inserts before the data is used (in the case of a read) or before the data is re-written (in the case of a write). The wait call halts execution until the post command has completed.

To maximize the amount of computation between the call and the wait statement, the scheduler first attempts to insert *prefetch* non-blocking calls in the loop body such that the current iteration executes a non-blocking call to retrieve the shared-object used in the next iteration. If the scheduler is unable to prefetch the shared-object, it schedules the non-blocking call within the basic block containing the shared reference.

6.3.1 Algorithm

The scheduling of remote shared-object accesses attempts to overlap the communication required to move shared objects between address partitions with other (unrelated) computation. The motivation is to reduce the amount of time a processor spends idle waiting for the communication to complete. Parallel loop nests that do not contain synchronization points will provide the scope to which SOAS will be applied. That is, only parallel loop nests that do not contain barriers, collectives that imply synchronization or shared references that are marked as strict will be candidates for SOAS. In the future, these restrictions can be relaxed, however, more analysis will be necessary to ensure that the scheduled accesses do not cross any synchronization point. Traditional data dependences must also be preserved when scheduling remote shared-object accesses.

The algorithm to schedule remote shared-object accesses is shown in Figure 6.17. The algorithm takes a candidate **upc_forall** loop and the shared reference map generated by locality analysis. The algorithm begins by obtaining the affinity statement for the **upc_forall** loop (step 1). Next it iterates through all remote threads in the shared reference map and for each remote thread T the algorithm obtains the list of shared references owned by T.

```
SCHEDULESOA(ForallLoop L_forall, SharedReferenceMapArray A)
1. A_stmt ← Affinity statement used in L_forall
2. foreach remote thread T
3.         M_T ← A[T]
4.         foreach ref_type ∈ def, use
5.                 RefList_T^ref_type ← SCHEDULECANDIDATES(ref_type, M_T)
6.                 foreach shared reference symbol r_s in RefList_T^ref_type
7.                         wait_sym ← φ
8.                         if A_stmt dominates r_s and r_s postdominates A_stmt and ref_type = use then
9.                           index_first ← Generate first index expression
10.                          index_next ← Generate next index expression
11.                          post_first ← GENERATEPOST(index_first, wait_sym, r_s)
12.                          post_next ← GENERATEPOST(index_next, wait_sym, r_s)
13.                          INSERTPOST(post_first, ref_type, L_forall^header)
14.                          guard ← INSERTPOSTGUARD(index_next, r_s)
15.                          INSERTPOST(post_next, ref_type, guard)
16.                          INSERTWAIT(wait_sym, r_s)
17.                        else
18.                          block ← basic block containing r_s
19.                          index ← r_s^index
20.                          post ← GENERATEPOST(index, wait_sym, r_s)
21.                          wait ← GENERATEWAIT(post)
22.                          INSERTPOST(post, ref_type, block)
23.                          INSERTWAIT(wait, r_s)
24.                        end if
25.                end for
26.        end for
27.end for
```

Figure 6.17: Algorithm to schedule remote shared-object accesses

The SCHEDULECANDIDATES algorithm collects all shared references of the specified type (*use* or *def*) into a list. The SCHEDULECANDIDATES algorithm does not need to check for strict shared references because loops containing such references are not candidates for the SCHEDULESOA algorithm.

The algorithm proceeds by attempting to schedule each candidate shared reference. The required condition for scheduling shared object accesses is that the *post* instruction, the corresponding *wait* instruction and the original shared reference all must be control-flow equivalent. Thus, in order to schedule the access such that the current iteration of the **upc_forall** loop *prefetches* the shared reference used in the next iteration, the shared reference must be control-flow equivalent with the affinity statement. Step 8 checks for this condition. If A_{stmt} and r_s are control flow equivalent, then r_s is a candidate for prefetching. Steps 9 to 16 generate the code for prefetching the shared reference. The index of the first iteration of the **upc_forall** loop is computed (step 9) and used in the first *post* instruction, placed in the loop header (step 13). The index of the shared reference in the next iteration is computed based on the current index and the blocking factor of the shared reference used in the affinity statement. If the **upc_forall** loop uses an integer affinity test then a blocking factor of 1 is assumed. If the blocking factor is 1 then $index_{next}$ is the index of the next shared array element owned by the executing thread. This is computed using:

$$index_{next} = index_{current} + \texttt{THREADS} \tag{6.1}$$

If the blocking factor is greater than one, then the next index must be determined at runtime based on the position of the current index within the block. To determine this, the compiler inserts the code seen in Figure 6.18.

```
1 offset = 1;
2 if (currIndex % BlockingFactor == BlockingFactor - 1)
3   offset = offset + (BlockingFactor * (THREADS-1));
4 nextIndex = currIndex + offset;
```

Figure 6.18: Code to compute the next index at runtime

Before prefetching the shared reference from the next iteration of the loop, the compiler must ensure that the next iteration will be executed. This is done by inserting a guard branch to compare the next index of the shared reference with the upper bound of the loop. The post instruction is then placed under the guard branch (step 15). The GENERATEPOST algorithm generates the *post* instruction for the given shared reference. It take as a parameter the

symbol used by the wait statement. If this symbol is uninitialized, the algorithm creates a new symbol and initializes it. If the symbol is already initialized, a new symbol is not created. Therefore all post calls store the return value to the same symbol.

If the compiler was not able to prefetch the shared reference, the algorithm defaults to scheduling the shared-object access within the basic block containing the shared reference (Steps 17 to 24). These steps are similar to the ones used for prefetching, except that the index used in the *post* call is from the current loop iteration and the generated instructions are placed within the basic block containing r_s. No guard branches are necessary before the call because the index used in the post call is from the current iteration.

```
1  shared double MySharedData[SHARED_DATA_SIZE];
2  shared double RemoteData[SHARED_DATA_SIZE];
3  double MyPrivateData[PRIVATE_DATA_SIZE];
4  void Compute() {
5    int i, j, k;
6    double sum=0.0;
7    wait = __xlupc_deref_array_post(RemoteData_h, MYTHREAD);
8    upc_forall (i=0; i < SHARED_DATA_SIZE; i++; &MySharedData[i]) {
9      for (j=0; j < PRIVATE_DATA_SIZE; j++) {
10       sum += MyPrivateData[j];
11     }
12     offset=THREADS;
13     __xlupc_wait(wait);
14     tmp2 = tmp;
15     if (i+THREADS) < SHARED_DATA_SIZE)
16       wait = __xlupc_deref_array_post(RemoteData_h,&tmp,
17                                       (i+THREADS+1)%SHARED_DATA_SIZE);
18     MySharedData[i] = sum * tmp2;
19   }
20 }
```

Figure 6.19: Optimized synthetic UPC benchmark

Consider again the synthetic benchmark presented in Figure 6.15. The locality analysis identifies the owner of shared reference $RemoteData_{(i+1)\%(SHARED\_DATA\_SIZE)}$ as *UNKNOWN* and places it into the shared reference map accordingly. The owner is *UNKNOWN* because the compiler cannot compute the modulo operation at compile time and therefore the affinity distance is not a compile-time constant. This reference will be a candidate for SCHEDULESOA because shared references with owners marked as *UNKNOWN* are treated as remote. The affinity statement and the $RemoteData_{(i+1)\%(SHARED\_DATA\_SIZE)}$ reference are control flow equivalent. Thus the compiler inserts a prefetech call into the **upc_forall** loop. The $index_{first}$ is computed to be `MYTHREAD` and used in the post call inserted in the loop header. Figure 6.19 shows the optimized synthetic UPC benchmark.

Figure 6.20 shows two shared references, A_i and B_i that are not control-flow-equivalent

```
shared int A[N];
shared int B[N];

int foo() {
  int i, x;
  upc_forall (i=0;i<N;i++;&A[i]) {
    if (i % M) {
      x = A[i];
    }
    else {
      x = B[i];
    }
  }
  ...
}
```

(a) Original UPC code

```
shared int A[N];
shared int B[N];

int foo() {
  int i, x;
  upc_forall (i=0;i<N;i++;&A[i]) {
    if (i % M) {
      waitSym =
        __xlupc_deref_array_post(A_h,i);
      __xlupc_wait(waitSym);
      x = A[i];
    }
    else {
      waitSym =
        __xlupc_deref_array_post(B_h,i);
      __xlupc_wait(waitSym);
      x = B[i];
    }
  }
  ...
}
```

(b) After SCHEDULESOA

Figure 6.20: Non-control-flow-equivalent shared references

with the affinity statement in the **upc_forall** loop. In this case, the compiler does not attempt to prefetch shared references A_{i+1} or B_{i+1} because it cannot determine which will be used next. In this situation, the compiler inserts the non-blocking calls at the beginning of the basic block containing the shared reference, as seen in Figure 6.20(b). In this example, there would be no gain from inserting the calls because the basic blocks containing the shared references are small. Also, if the control-flow paths containing the shared references are long, it may be beneficial to separate the non-blocking calls and the subsequent wait statements by moving the non-blocking calls up the control flow path. While doing this, the compiler must maintain the control-flow relationship between the non-blocking call and the shared reference, *i.e.* the non-blocking call must remain control-flow equivalent with the shared reference. Moving code to separate the non-blocking calls from wait statements, in the presence of control flow, has been left for future investigation.

6.3.2 Results

Figure 6.21 shows the relative improvement obtained from using the SCHEDULESOA algorithm on the synthetic benchmark in Figure 6.15. The benchmark used sizes of 2000, 200000, and 2000000 for the private and shared arrays in the benchmark (*e.g.,* pd2ksd200k represents a private array of 2000 and a shared array of 200000). These variations measure the impact that the different data sizes have on performance. By adjusting the size of the

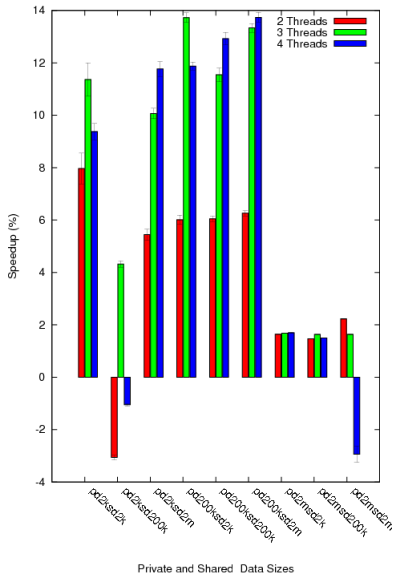


Figure 6.21: Improvement over baseline from SCHEDULESOA algorithm

private array, the amount of computation between the prefetch call and the subsequent wait is altered. Similarly, altering the size of the shared data determines the number of prefetch calls that are issued in a given program execution.

The relative improvements shown in Figure 6.21 demonstrate that there is a benefit to scheduling shared-object accesses to overlap the communication with the computation. In this example, this benefit appears to be greatest for private array sizes of 2000 and 200000 elements. When the size is increased to 2000000, the cost of the private computation dominates the execution and the communication becomes a secondary concern. For private arrays sizes of 2000 and 200000 there is not as much benefit from scheduling shared-object accesses for 2 threads as there is for 3 and 4 threads. We do not have an explanation for this surprising result. [1] Similarly, we have no explanation for the slowdown for 2 and 4 threads using a private array of 2000 and a shared array of 200000.

6.3.3 Discussion

Software prefetching is an optimization where the compiler inserts machine instructions that begin to *prefetch* into the data cache. The compiler attempts to insert the prefetch instructions such that the data will be available when it is first used [4, 6, 58]. The SCHEDULESOA algorithm is similar to software prefetching in that it attempts to hide the latency of accessing remote shared data by overlapping it with unrelated computation. Despite the

[1] The speedups reported in Figure 6.21 are the average of 8 runs.

advantages of software prefetching, Allen and Kennedy list three disadvantages: (i) it increases the amount of instructions that must be executed; (ii) it can cause useful data cache lines to be evicted; (iii) data brought into the cache can be evicted before use, or never used. As a result of these disadvantages, the compiler must perform analysis to determine when software prefetching is profitable and must try to reduce the occurences of these situations. The SCHEDULESOA algorithm does not have to deal with these disadvantages. The blocking calls in the RTS to access shared data are implemented using a post call followed immediately by a call to the corresponding wait function. Thus, there are no additional instructions executed by using the non-blocking calls to the RTS instead of the blocking calls. When retrieving a shared object using non-blocking calls to the RTS, the calls will not overwrite (or otherwise destroy) existing data that may be used. Finally, the non-blocking calls to the RTS to access shared objects are only executed if the original blocking calls would have been executed. Thus, it is not possible to needlessly call the RTS and cause shared data to be retrieved if it is not going to be used. As a result, the issues addressed by the SCHEDULESOA algorithm are significantly different from existing techniques developed for software prefetching.

The SCHEDULESOA algorithm currently only attempts to schedule shared reference accesses across loop iterations when the shared references are *uses* (*i.e.* the algorithm only attempts to *prefetch* data). In theory it is possible for the compiler to perform an analogous optimization when the shared references are *defs*. In this case, the compiler would insert a non-blocking post call immediately after the shared reference is written, and place the corresponding wait command before the shared reference is written in the next iteration. A final wait command would be placed in the loop prolog to be executed once after the loop has finished execution. This final wait would ensure that the shared reference in the last iteration of the loop is written correctly. This extension to remote shared-object access scheduling has been left for future work.

The compiler currently does not support scheduling coalesced shared-object accesses. While the RTS does provide the functionality, the compiler is currently not able to generate the non-blocking coalesced calls and to insert the subsequent wait commands. This implementation has been left for future work.

6.4 Remote Shared-Object Updates

When the compiler can identify an update performed by a thread T_i to a remote shared object O_s owned by a thread T_j, $i \neq j$, it is not necessary for T_i to perform a remote shared read of O_s, update O_s and then perform a remote shared write of O_s.[2] Instead, a special update function defined in the RTS can be used. This update function sends a message from T_i to T_j containing the update instructions. The update is applied by T_j. This update mechanism reduces the number of messages that need to be sent, resulting in a performance improvement [10].

6.5 Chapter Summary

This chapter introduced four locality-based optimizations that the compiler can perform to improve performance of UPC programs. Each optimization requires the locality analysis presented in Chapter 5 to create the shared reference map. Shared-object access privatization allows the compiler to generate code that directly accesses shared objects located on the same node as the accessing thread instead of generating calls to the RTS to access the data. Shared-object access coalescing allows the compiler to combine two or more remote shared-object accesses into a single call to the RTS, thereby reducing the amount of communication required to access the remote data. Shared-object access scheduling allows the compiler to overlap the calls to access remote shared objects with other computation in an attempt to hide the cost of accessing remote shared objects. Finally, updating remote shared objects allows the compiler to generate special calls to the RTS that cause the update instructions to be performed on the remote node containing the shared-object. Chapter 8 evaluates the effects of each of these optimizations on a set of UPC benchmarks.

[2] A remote shared read followed by a remote shared write would require two messages from T_i to T_j and one message from T_j to T_i.

Chapter 7

Parallel Loop Nests

The **upc_forall** statement distributes iterations of the loop among all UPC threads. Instead of each thread executing all iterations of the loop, an iteration is conditionally executed by a thread based on an *affinity test*. UPC supports two types of affinity test statements: *integer* and *pointer-to-shared*. According to the semantics of an integer affinity test, a thread executes the current iteration of the loop if the integer value used in the affinity statement modulo the number of threads is equal to the thread ID. According to the semantics of a pointer-to-shared affinity test a thread executes the current iteration of the loop if it owns the shared object specified in the affinity statement.

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; i)
    a[i] = b[i] + SCALAR*c[i];
}
```

(a) Integer affinity test

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; &a[i])
    a[i] = b[i] + SCALAR*c[i];
}
```

(b) Pointer-to-shared affinity test

Figure 7.1: UPC STREAM triad kernel

Figure 7.1 shows the STREAM triad kernel written in UPC using integer and pointer-to-shared affinity tests respectively [44]. For the integer affinity test, the loop body is executed by T if and only if i%**THREADS** is equal to T's ID. For the pointer-to-shared affinity test, the loop body is executed by a thread T if and only if T owns a[i]. Figure 7.2 shows the naive transformation of **upc_forall** loops to C **for** loops. In this example every thread will execute the same iterations for both loops because the default (cyclic) blocking factor is used to distribute the shared arrays.

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  for (i=0; i < N; i++) {
    if ( (i % THREADS) == MYTHREAD )
      a[i] = b[i] + SCALAR*c[i];
  }
}
```

(a) Equivalent integer affinity test

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  for (i=0; i < N; i++) {
    if (upc_threadof(&a[i]) == MYTHREAD)
      a[i] = b[i] + SCALAR*c[i];
  }
}
```

(b) Equivalent pointer-to-shared affinity test

Figure 7.2: UPC STREAM triad kernel with equivalent **for** loops

To determine the overhead of the naive transformation of parallel loops, we compare the performance of the UPC STREAM triad kernels in Figure 7.1 to an equivalent C implementation using a **for** loop run on a single thread. All loops access shared arrays, allowing us to isolate the overhead of the **upc_forall** loop structure. Table 7.1 contains the memory bandwidth, measured in MB/s, reported by the STREAM triad kernel run on a 1.6 GHz POWER5 shared-memory machine with 64 (SMT-enabled) processors. All tests were compiled with UPC optimizations and loop optimizations enabled. All shared accesses were privatized because the experiments were conducted on an SMP machine. For all experiments, the number of UPC threads was specified at compile-time.

Threads	C **for** loop	**upc_forall** (integer affinity)	**upc_forall** (pointer-to-shared affinity)
1	3253.2005	2690.7	102.2
2		2414.7	121.3
4		3425.3	151.7
8		2687.5	172.9
16		3739.7	180.0
32		4504.8	185.0
64		4964.4	183.6

Table 7.1: UPC STREAM benchmark using different types of loops (MB/s)

The memory bandwith measurements reported in Table 7.1 demonstrate that there is overhead when using both integer affinity and pointer-to-shared affinity tests. However, the overhead for integer affinity is much less severe then the overhead for pointer-to-shared. When run with a single thread, the integer affinity test becomes **if** ((i%1)== 0) which

is always true and thus the compiler is able to remove the branch.[1] Even with the branch removed the **upc_forall** loop is still 17% slower than the C **for** loop. This is attributed to the additional computations inserted, by the PRIVATISESOA code, to compute the offset used in the direct pointer access. With 2 threads, the modulo operation in the integer affinity test introduces some overhead, and reduces the overall performance by approximately 25%.

As more threads are added, each thread has less work to do and the performance increases. When run with 4 threads there is a 5% improvement over sequential performance. When run with 64 threads, the performance improves by 53% over the sequential C version. An improvement of 53% is disappointing considering that the benchmark is running on 64 threads.

The cost of the pointer-to-shared affinity test is much higher and results in a performance degradation of 97% on a single thread and almost 94% on 64 threads. The compiler is not able to simplify the affinity branch in any way because this affinity test is not directly related to the number of threads. Therefore, the branch is executed in every iteration of the loop. Furthermore, the branch contains a function call to the RTS that has to consult the layout of the shared array to determine which thread owns the shared array element.

An execution profile of the STREAM benchmark using the pointer-to-shared affinity test reveals that for one thread approximately 88% of the execution time is spent in calls to the RTS while only 5% of the execution is spent in the STREAM triad kernel. When run with 32 threads, approximately 96% of the execution time is spent in calls to the RTS and only 3% of the time is spent in the STREAM triad kernel. These measurements indicate that the calls to the **upc_threadof** function contributes significantly to the execution time of the benchmark and must be eliminated to achieve reasonable performance.

The naive transformations described above divide the iteration space of the loop nest among all threads by inserting guard branches into the loop body. In many cases the compiler can modify the lower bound, upper bound and increment of the **upc_forall** loop to achieve the same behaviour without the high overhead of the guard statement. However, to preserve the original semantics of the program, the transformation of the **upc_forall** loop must ensure that the iteration space of the modified loop nest is identical to the iteration space of the original loop.

[1] When compiling for a single UPC thread, the compiler replaces `MYTHREAD` with 0.

7.1 Integer Affinity Tests

Consider again the STREAM triad kernel from Figure 7.1 written using an integer affinity test. In this case a thread T will execute an iteration of the loop if the affinity test modulo the number of threads is equal to T's ID. We only consider **upc_forall** loops that use the loop induction variable in the affinity test (the most common case). Consider a **upc_forall** loop L_{forall} with upper bound N. The iteration space for L_{forall} on thread T, $\mathcal{I}_T$ is:

$$\mathcal{I}_T = \{\texttt{MYTHREAD}, \texttt{MYTHREAD} + \texttt{THREADS}, \texttt{MYTHREAD} + (2 * \texttt{THREADS}), \ldots, K\}, K < N \leq K + \texttt{THREADS} \quad (7.1)$$

Thus, $\mathcal{I}_T$ contains at most $\lceil \frac{N}{\texttt{THREADS}} \rceil$ elements where the m^{th} element is $\texttt{MYTHREAD} + ((m-1) * \texttt{THREADS})$. The last element, K, guarantees that i_T falls within the original loop bounds. The **upc_forall** loop can be modified to execute this iteration space on each thread by changing the lower bound of the loop to MYTHREAD and the increment of the loop to THREADS. The upper bound of the loop remains the same.

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  upc_forall(i=0; i<N; i++; i)
    a[i] = b[i] + SCALAR*c[i];
}
```

(a) Original forall loop

```
#define SCALAR 3.0
shared double a[N];
shared double b[N];
shared double c[N];

void StreamTriad() {
  int i;
  for (i=MYTHREAD; i<N; i+=THREADS)
    a[i] = b[i] + SCALAR*c[i];
}
```

(b) Optimized for loop

Figure 7.3: UPC STREAM triad kernel with integer affinity test

Figure 7.3 shows the original STREAM benchmark with an integer affinity test and the optimized **for** loop.

7.2 Pointer-to-Shared Affinity Tests

The algorithm to optimize **upc_forall** loops with pointer-to-shared affinity is a strip-mining type algorithm where the **upc_forall** loop is replaced by a two-level loop nest. The outer loop, L_{outer}, iterates over blocks in the shared array and the inner loop, L_{inner}, iterates within the block. Let T be the UPC thread that is executing the new loop nest. The outer and inner loops are combined to access all elements in the shared array used in the affinity test that are owned by T and within the bounds of the original **upc_forall** loop.

```
shared [2] double A[6][6];

void UpperTriangularLoop() {
  int i,j;
  for (i=0; i < 6; i++) {
    upc_forall (j=i; j < 6; j++; &A[i][j]) {
      A[i][j] = (double) MYTHREAD;
    }
  }
}
```

(a) UPC upper-triangular loop nest

0	0	1	1	0	0
-1	1	0	0	1	1
-1	-1	1	1	0	0
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	0
-1	-1	-1	-1	-1	1

(b) Upper-triangular shared array with 2 threads

Figure 7.4: Upper-triangle loop nest

Consider the upper-triangular loop nest in Figure 7.4. This loop nest initializes the elements in the upper-triangular portion of a 2D array to the thread ID that owns the element. The affinity test in the **upc_forall** loop ensures that the initialization for each array element A[i][j] is performed by the thread that owns the element. The shared array A is distributed so that each thread owns two consecutive elements in the shared array (blocking factor of 2). Assume that the shared array A is initialized to contain -1 in every element. Figure 7.4(b) shows the 6x6 shared array after the loop nest has run using 2 UPC threads.

```
shared [2] double A[6][6];

void UpperTriangularLoop() {
  int i,j;
  for (i=0; i < 6; i++) {
    upc_forall (j=0; j < 6-i; j++; &A[i][i+j]) {
      A[i][i+j] = (double) MYTHREAD;
    }
  }
}
```

Figure 7.5: Triangular loop nest after loop normalization

The pointer-to-shared optimization is designed to work with normalized **upc_forall** loops. A normalized loop is a countable loop that has been modified to start at 0 and to iterate, by increments of 1, up to a specific upper bound. Figure 7.5 shows the normalized UpperTriangular loop nest. The accesses to the shared arrays (in the affinity test as well as the loop body) have been modified from A[i][j] to A[i][i+j] as part of the normalization process.

7.2.1 New Loop-Nest Structure

0	0	1	1	0	0
1	1	0	0	1	1
0	0	1	1	0	0
1	1	0	0	1	1
0	0	1	1	0	0
1	1	0	0	1	1

(a) Shared array A

B_0	B_0	B_0	B_0	B_1	B_1
B_1	B_1	B_2	B_2	B_2	B_2
B_3	B_3	B_3	B_3	B_4	B_4
B_4	B_4	B_5	B_5	B_5	B_5
B_6	B_6	B_6	B_6	B_7	B_7
B_7	B_7	B_8	B_8	B_8	B_8

(b) Block groups of A

Figure 7.6: Thread and block ownership of shared array A

Figure 7.6 shows the shared array A from the upper-triangle loop of Figure 7.4 and the corresponding blocks of A. The element numbers in 7.6(a) correspond to the thread that owns the shared array element. The element numbers in 7.6(b) correspond to the block group that contains the shared array element. For every row of the shared array (every invocation of the **upc_forall** loop), different parts of a block group are modified. Thus, the optimization cannot assume that every block group of a shared array will be modified by a **upc_forall** loop. Similarly, the optimization must also be able to handle cases where only parts of a block group are modified. For example, the second iteration of the i loop ($i = 1$) in Figure 7.4 starts at position A[1][1] and thus shared array element A[1][0] (located in the end of block group B_1) is not modified. The lower bound of L_{outer} must start at the index of the first block owned by thread T (within the iteration space of the original loop). The bounds of L_{outer} and L_{inner} are computed at runtime to ensure that the iteration space of the optimized loop nest is identical to the iteration space of the original loop nest.

The modified outer loop iterates over blocks owned by the thread T and the modified inner loop iterates within blocks. Thus, the bounds of the outer loop are used to determine the blocks that are traversed. The lower bound of the outer loop determines the first block that is traversed by thread T, the increment determines the next block that is traversed, and the upper bound determines the last block that is traversed. Similarly, since the inner loop traverses within a block, the lower bound of the inner loop determines the position within the block where thread T begins, the increment determines the next position and the upper bound determines how many elements in the block are traversed. Prior to the execution of the new loop nest, thread T must determine (i) the first block that the loop nest will iterate over; and (ii) the offset within the block at which to begin iterating. The first block over which the loop nest iterates determines the lower bound of L_{outer} while the offset within the block determines the lower bound of L_{inner}. These bounds are computed by

generating code to convert the affinity test into a linearized array offset. The position of the linearized array offset in the block group is used to determine where thread T should begin its execution.

StripMineForallLoop($ForallLoop\ L_{forall}$)
1. $r_{aff} \leftarrow$ shared reference used in affinity test
2. $s_{curr} \leftarrow$ statement before $L_{forall}^{GuardBranch}$
3. $\mathcal{B} \leftarrow r_{aff}^{BlockingFactor}$
4. $s_{new} \leftarrow$ Generate($pos_{first}^{norm} = LinearOffset(r_{aff})\%\ (\mathcal{B} \times \mathbf{THREADS})$)
5. LinkAndAdvance(s_{new}, s_{curr})
6. $s_{new} \leftarrow$ Generate($BlockStart_T = \mathcal{B} \times$ MYTHREAD)
7. LinkAndAdvance(s_{new}, s_{curr})
8. $s_{new} \leftarrow$ Generate($BlockEnd_T \leftarrow BlockStart_T + \mathcal{B} - 1$)
9. LinkAndAdvance(s_{new}, s_{curr})
10. $s_{new} \leftarrow$ Generate($\mathcal{G}^{start} = \left\lfloor \frac{pos_{first}^{norm}}{\mathcal{B} \times \mathbf{THREADS}} \right\rfloor \times \mathcal{B} \times \mathbf{THREADS}$)
11. LinkAndAdvance(s_{new}, s_{curr})
12. $s_{new} \leftarrow$ Generate(**if** $pos_{first}^{norm} \leq BlockStart_T$)
13. LinkAndAdvance(s_{new}, s_{curr})
14. $s_{new} \leftarrow$ Generate($outerLB = \mathcal{G}^{start} + BlockStart_T$)
15. LinkAndAdvance(s_{new}, s_{curr})
16. $s_{new} \leftarrow$ Generate($innerLB = 0$)
17. LinkAndAdvance(s_{new}, s_{curr})
18. $s_{new} \leftarrow$ Generate(**elseif** $BlockStart_T < pos_{first}^{norm} \leq BlockEnd_T$)
19. LinkAndAdvance(s_{new}, s_{curr})
20. $s_{new} \leftarrow$ Generate($outerLB = \mathcal{G}^{start} + BlockStart_T$)
21. LinkAndAdvance(s_{new}, s_{curr})
22. $s_{new} \leftarrow$ Generate($innerLB = LinearOffset(r_{aff})\%\ \mathcal{B}$)
23. LinkAndAdvance(s_{new}, s_{curr})
24. $s_{new} \leftarrow$ Generate(**else**)
25. LinkAndAdvance(s_{new}, s_{curr})
26. $s_{new} \leftarrow$ Generate($outerLB = \mathcal{G}^{start} + (\mathcal{B} \times \mathbf{THREADS}) + BlockStart_T$)
27. LinkAndAdvance(s_{new}, s_{curr})
28. $s_{new} \leftarrow$ Generate($innerLB = 0$)
29. LinkAndAdvance(s_{new}, s_{curr})
30. $L_{outer} \leftarrow$ new loop
31. $L_{outer}^{LowerBound} \leftarrow outerLB$
32. $L_{outer}^{UpperBound} \leftarrow L_{forall}^{UpperBound} + LinearOffset(r_{aff})$
33. $L_{outer}^{Increment} \leftarrow \mathcal{B} \times \mathbf{THREADS}$
34. $L_{inner} \leftarrow$ new loop
35. $L_{inner}^{LowerBound} \leftarrow innerLB$
36. $L_{outer}^{Body} \leftarrow L_{inner}$
37. $L_{inner}^{Body} \leftarrow L_{forall}^{Body}$
38. Link L_{outer} after s_{curr}
39. Remove L_{forall}

Figure 7.7: Algorithm to strip-mine **upc_forall** loops with pointer-to-shared affinity

Figure 7.7 shows the algorithm used to strip-mine a **upc_forall** loop that contains a pointer-to-shared affinity test. The algorithm uses two assist routines to perform the strip-mining. The GENERATE routine generates a statement based on the input expression. The LINKANDADVANCE routine takes two reference parameters s_{new} and s_{curr} representing

pointers to statements. The statement referenced by s_{new} is placed in the statement list immediately following statement referenced by s_{curr}. The statement pointer s_{curr} is then advanced to point to the statement referenced by s_{new}.

$$LinearOffset(A[i][i+j]) = [i \quad i+j] \times \begin{bmatrix} 6 \\ 1 \end{bmatrix} \tag{7.2}$$

Let pos_{first} be the first position of the shared array accessed by the **upc_forall** loop. The first position is determined by computing (at runtime) the linearized offset of the array element used in the affinity test using equation 7.2, where the column vector represents the dimension sizes of the shared array A. The lower bound of the **upc_forall** loop is used as the value of the induction variables defined by the **upc_forall** loop. Thus, equation 7.2 is simplified to

$$LinearOffset(A[i][i+j]) = [i \quad i] \times \begin{bmatrix} 6 \\ 1 \end{bmatrix}, \quad j = 0$$

For example, pos_{first} for the first iteration of the i loop ($i = 0$) is $(0 \times 6) + (0 \times 1) = 0$; pos_{first} for the second iteration of the outer loop ($i = 1$) is $(1 \times 6) + (1 \times 1) = 7$. Table 7.2 shows the pos_{first} values for the different values of i when executing the upper-triangle loop in Figure 7.4.

i	pos_{first}	pos_{first}^{norm}	$\mathcal{G}$	$\mathcal{G}^{start}$
0	0	0	0	0
1	7	3	1	4
2	14	2	3	12
3	21	1	5	20
4	28	0	7	28
5	35	3	8	32

Table 7.2: Runtime values for various values of i

The bounds of the new loops are determined by computing the position of the next block from pos_{first} owned by thread T. This is done by normalizing pos_{first} with respect to the size of the block group to determine the relative position in the block group. The normalized first position, pos_{first}^{norm} , is then compared to $BlockStart_T$. Table 7.2 shows the normalized first positions and block group start positions for the different values of i. The result of this comparison will fall into one of three categories:

Case 1 $pos_{first}^{norm} \leq BlockStart_T$

If the normalized first position is less than the first index in the block owned by thread T, then the lower bound of L_{outer} is set to the beginning of the block owned by thread T in the current block group ($\mathcal{G}^{start} + BlockStart_T$). This means that the shared array element accessed in the first iteration of the **upc_forall** loop is located before the block owned by thread T and thus thread T should begin its execution at the beginning of its first block index in the current block group. An example of this case is seen in Figure 7.4(b) when thread 1 is executing the first iteration of the i loop (iteration vector [0 0]). From Table 7.2, $pos_{first} = 0$, $pos_{first}^{norm} = 0$ and $\mathcal{G} = 0$. Because $Block_1^{start} = 2$, the lower bound of the outer loop will be set to 2. Thus, T will begin its execution at the start of the first block. As such, the lower bound of the inner loop is set to 0.

Case 2 $BlockStart_T < pos_{first}^{norm} \leq BlockEnd_T$

If pos_{first}^{norm} falls between $BlockStart_T$ and $BlockEnd_T$ then the iteration vector of the **upc_forall** loop begins in the block owned by thread T. In this case, the lower bound of the outer loop is also set to begin at the beginning of its first block ($\mathcal{G}^{start} + BlockStart_T$). However, since the first position is located in the block owned by thread T, not all iterations of the block will be executed. Thus, the lower bound of the inner loop will be set to the correct offset within the block. An example of this is seen in Figure 7.4(b) when thread 1 is executing the second iteration of the i loop (iteration vector [1 1]). From Table 7.2, $pos_{first} = 7$ and $pos_{first}^{norm} = 3$ while $BlockStart_T = 2$ and $BlockEnd_T = 4$. In this case, the lower bound of the outer loop will be set to $4 + 2 = 6$. Because the first iteration vector of thread T is located within the thread's first block, the entire first block will not be traversed. As a result, the lower bound of the inner loop is set to the offset within the block where the **upc_forall** loop begins ($pos_{first} \ \% \ \mathcal{B} = 1$).

Case 3 $BlockEnd_T < pos_{first}^{norm}$

Finally, if pos_{first}^{norm} is greater than $BlockEnd_T$ then the first block to be traversed by thread T is in the next block group. Thus, the lower bound of the outer loop is set to $BlockStart_T$ in the next block group (*i.e.* lower bound $= \mathcal{G} + |\mathcal{G}| + BlockStart_T$). Because the execution will begin at the start of the block, the lower bound of the inner loop is set to 0. This case occurs in Figure 7.4(b) when thread 0 is executing the third iteration of the i loop (iteration vector [2 2]). In this case, $pos_{first} = 14$ and

thus the forall loop begins execution in block group 3. However, $pos_{first}^{norm} = 2$ and $BlockEnd_T = 1$ indicating there are no blocks in the current block group for thread 0 to execute. Thus, the outer loop begins at block 4.

The upper bound of the outer loop is computed by adding the upper bound of the original **upc_forall** loop to pos_{first}. Because the original **upc_forall** loop is normalized, its upper bound represents the number of iterations that are executed by the loop. Adding that number to pos_{first} will ensure that the same number of iterations are executed by L_{outer}. For example, on the fifth iteration of the i loop ($i = 4$), $pos_{first} = 28$ and the upper bound of the original **upc_forall** loop is 2. Thus, the upper bound of L_{outer} is set to 30. When thread 1 executes the fifth iteration of the i loop, the lower bound of L_{outer} is set to 30 (using Case 1). As a result, thread 1 will not execute any iterations of L_{outer} because the lower bound is equal to the upper bound.

Because the outer loop traverses over blocks of shared array elements owned by thread T, the increment of L_{outer} must move the induction variable to the next block owned by the executing thread. Thus, the increment is set to $\mathcal{B} \times$ THREADS, where $\mathcal{B}$ is the blocking factor of the shared array used in the affinity test of the original **upc_forall** loop.

```
1  shared [2] double A[6][6];
2
3  void UpperTriangularLoop() {
4    int i,j;
5    for (i=0; i < 6; i++) {
6      outerUB = i*6 + 6;
7      if ( ((i*6)+(i+0) % (2*2) < (MYTHREAD*2)) {
8        outerLB = (MYTHREAD*2) + (((i*6+i+0)/(2*2)) * (2*2));
9        innerLB = 0;
10     }
11     else if ( (i*7) % (2*2) < (MYTHREAD*2 + 2) ) {
12       outerLB = (MYTHREAD*2) + (((i*6+i+0)/(2*2)) * (2*2));
13       innerLB = (i*7) % 2;
14     }
15     else {
16       outerLB = (MYTHREAD*2) + (((i*6+i+0)/(2*2)) * (2*2)) + (2*2);
17       innerLB = 0;
18     }
19     for (k=outerLB; k < outerUB; k++) {
20       innerUB = min(i*6+6, k+2);
21       for (l=innerLB; l < innerUB-k; l++) {
22         A[i][((k+l) % 36) % 6] = (double) MYTHREAD;
23       }
24       innerLB = 0;
25     }
26   }
27 }
```

Figure 7.8: Upper-triangle loop nest after pointer-to-shared optimization

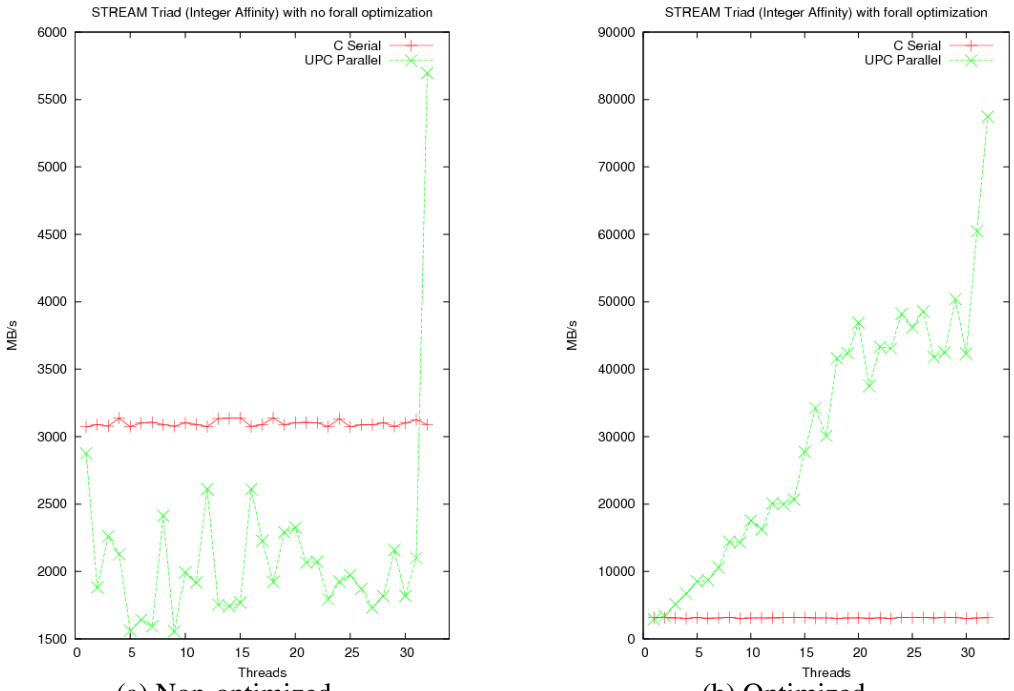
The indices used in the new loops are based on linearized offsets computed using the linearized offset of the affinity test at the beginning of the forall loop execution. Thus, the bounds of L_{outer} and L_{inner} are based on linearized offsets, not on array indices. Therefore, the induction variables used in L_{outer} and L_{inner} must be converted back into array indices before they are used. This conversion is done using modulo arithmetic, as seen on line 22 of Figure 7.8. While this conversion adds additional overhead, it allows the accesses to remain a traditional array access in the internal representation in the compiler. When a loop contains multiple occurrences of this conversion, the compiler uses common subexpression elimination to compute the array index once for each iteration of the loop, thus reducing the overhead. An alternative approach would be to convert the array accesses into pointer arithmetic where A[i][(k+l)%36%6] = (**double**)**MYTHREAD** would be replaced with $*$(A + (k+l)) = (**double**) **MYTHREAD**. However, doing this conversion transforms the array access into a pointer dereference in the internal representation in the compiler and can prevent certain optimizations. The ideal solution is to convert the bounds of L_{outer} and L_{inner} from linearized offsets back to array indices resulting in the induction variables representing array indices.

7.3 Experimental Evaluation

The results for the **upc_forall** optimization were collected on a 64-way POWER5 (1.6 GHz) machine running AIX 5.3 with 512GB of RAM.

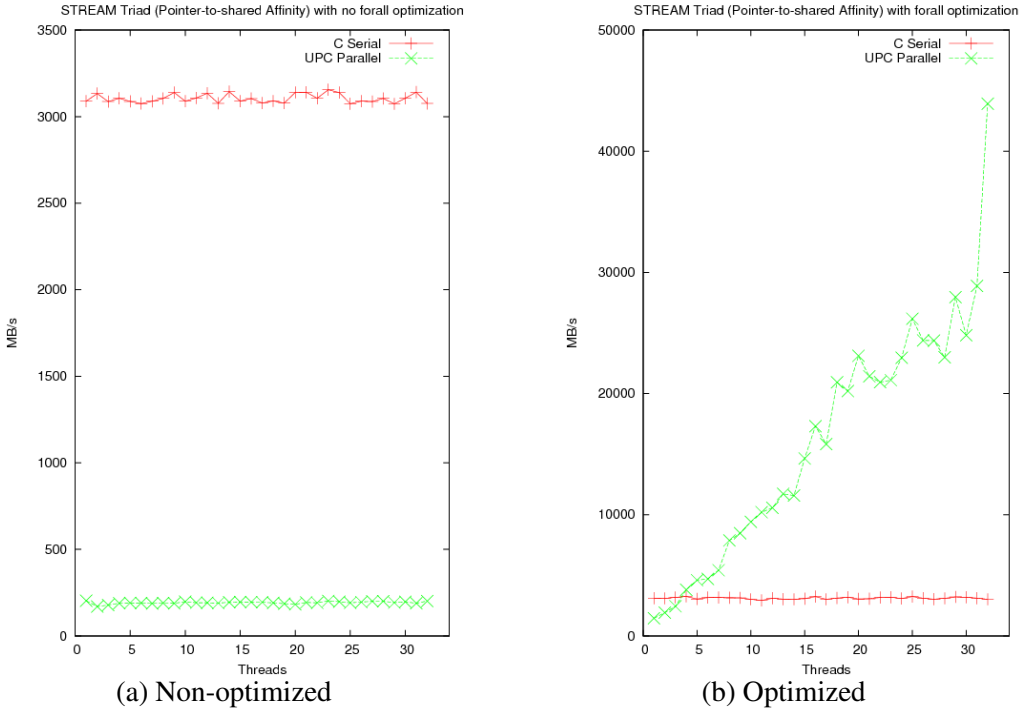
Figure 7.9 shows the results for the STREAM triad kernel using an integer affinity test. These graphs show the memory bandwidth (in MB/s) obtained based on the number of UPC threads (higher number is better). The results in Figure 7.9(a) correspond to the naive transformation, as described earlier (Table 7.1). Figure 7.9(b) shows the result of the integer optimization described in Section 7.1. The results demonstrate a significant improvement in the runtime performance of the STREAM triad benchmark when integer affinity is used.

Figure 7.10 shows the memory bandwidth measured by the STREAM triad kernel using a pointer-to-shared affinity test. The memory bandwidth shows a 38% degradation in performance for 2 threads and a 23% degradation for 3 threads. However, when run with 4 threads, there is a 17% improvement in performance. The performance degradation on 2 and 3 threads is due to the extra branches that are inserted before the new loop to compute the bounds as well as the conversion of the new induction variables from linearized array offsets back into array indices.



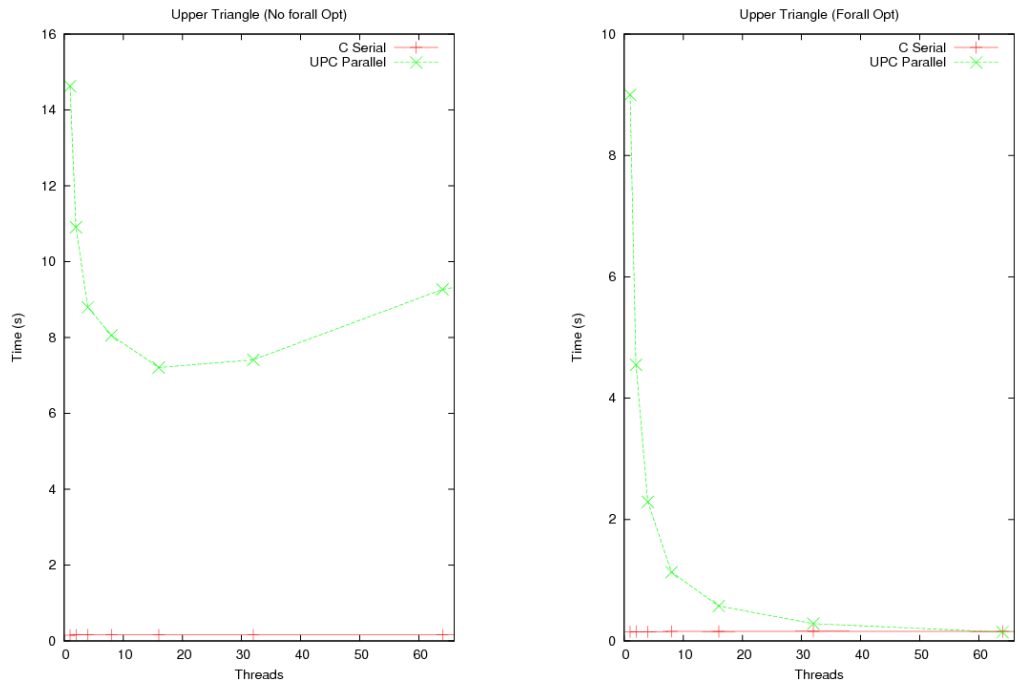
(a) Non-optimized (b) Optimized

Figure 7.9: STREAM triad (integer affinity)



(a) Non-optimized (b) Optimized

Figure 7.10: STREAM triad (pointer-to-shared affinity)



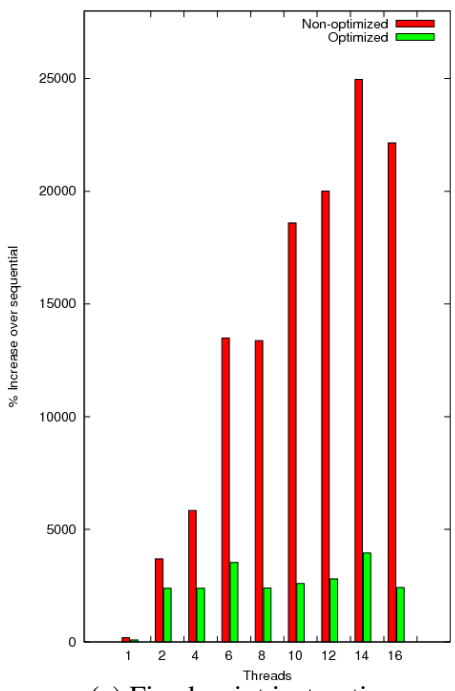
(a) Non-optimized (b) Optimized

Figure 7.11: Upper-triangle matrix (pointer-to-shared affinity)

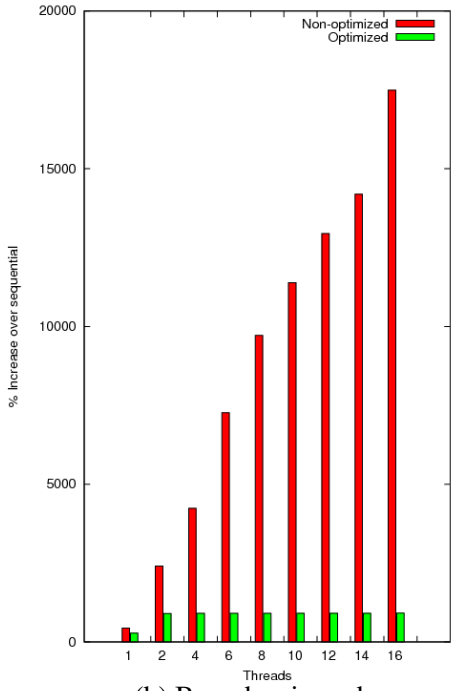
Figure 7.11 shows the execution times for running the upper-triangle initialization kernel (Figure 7.4) on a 11000x11000 (968MB) shared array distributed with a blocking factor of 2. The results measure the time taken to perform the upper-triangular access, and thus a lower time indicates better performance. The relative performance of the non-optimized UPC code is significantly worse than the sequential C code. The best performance is obtained with 16 threads (7.412s) but this performance is still much worse than the sequential performance (0.157s). Even with the improved code transformations 64 threads must be running for the performance of the UPC version to compare with the performance of the sequential version (0.145s and 0.155s respectively). These times demonstrate that there is still a significant cost to executing the branch instructions created by the forall optimization in some benchmarks.

Figures 7.12, 7.13, and 7.14 show the increase in the number of fixed-point instructions completed and the number of branch instructions issued when running the non-optimized and the optimized versions of the STREAM and upper-triangle benchmarks. These graphs show the percentage increase of the two events over the sequential versions of the benchmarks. The following observations can be made from these results:

1. There is a large increase in both the number of fixed-point instructions completed

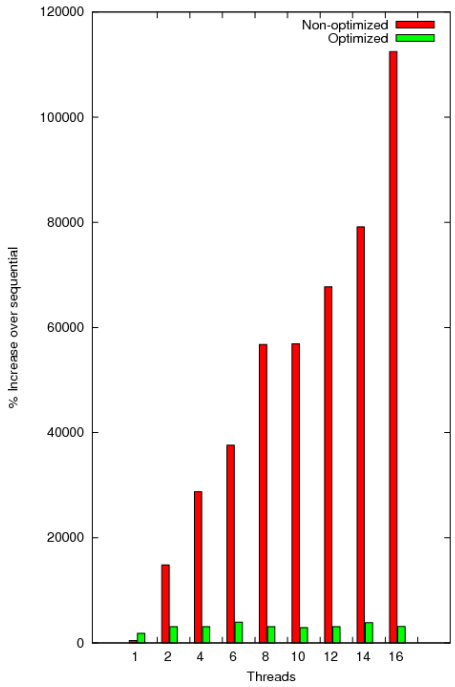


(a) Fixed-point instructions

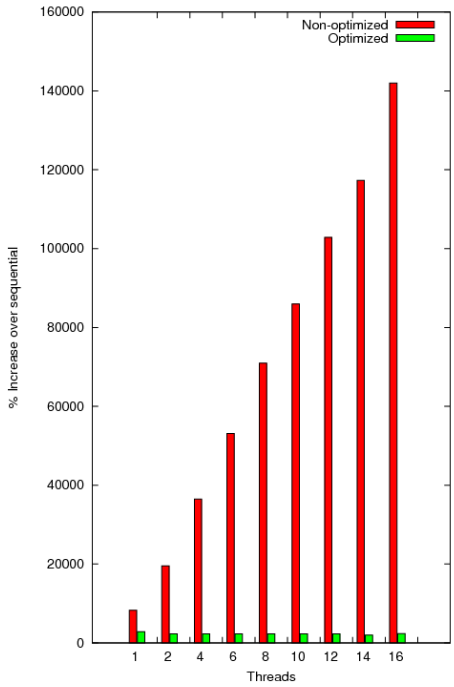


(b) Branches issued

Figure 7.12: Hardware performance counters for STREAM benchmark with integer affinity

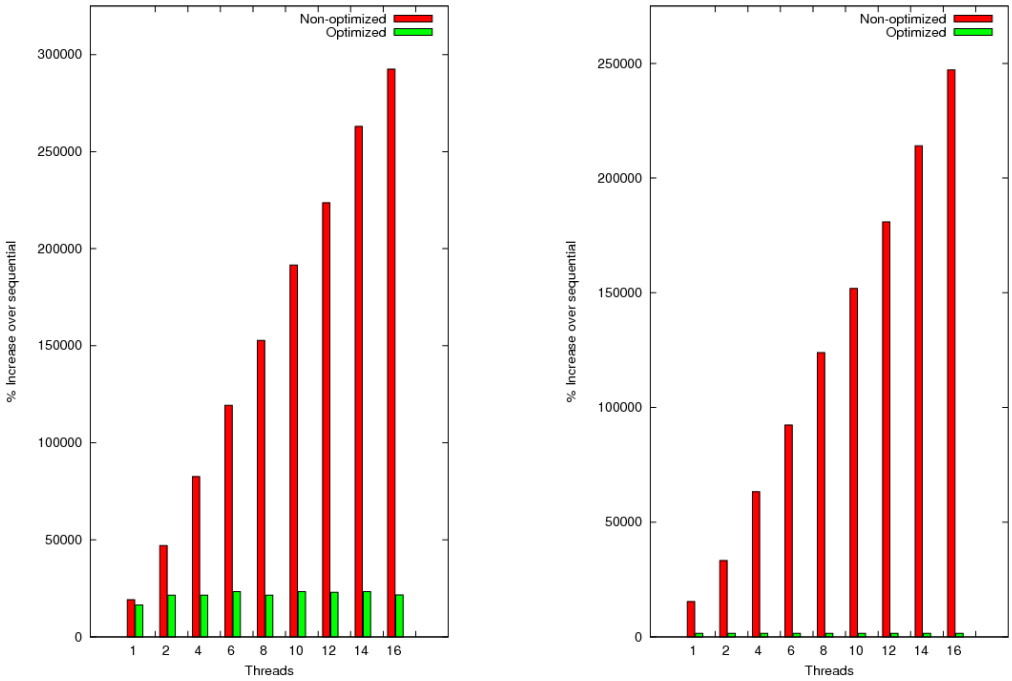


(a) Fixed-point instructions



(b) Branches issued

Figure 7.13: Hardware performance counters for STREAM benchmark with pointer-to-shared affinity



(a) Fixed-point instructions (b) Branches issued

Figure 7.14: Hardware performance counters for upper-triangle benchmark

and the number of branches issued between the sequential and the UPC versions of the benchmark. This increase is due to the calculations that are used for addressing the shared arrays and for computing the affinity of shared array elements. Both these operations use integer division and modulo operations in the computations.

The increase in the number of branch instructions for the non-optimized versions of the benchmarks (4.4X increase for the STREAM benchmark with integer affinity test, 83X for STREAM benchmark with pointer-to-shared affinity test, and 154X for upper-triangle benchmark) are partially due to the affinity tests and the conditional execution of the loop bodies. The difference in increase for the STREAM benchmark when integer affinity and pointer-to-shared affinity is used supports this explanation. The integer affinity test involves fewer branches and thus results in a smaller percent increase than the pointer-to-shared affinity test.

For the optimized versions of the benchmarks, there are fewer additional branch instructions executed: 2.8X increase for optimized STREAM benchmark with integer affinity, 29X increase for optimized STREAM benchmark with pointer-to-shared affinity, and 16X increase in the optimized upper-triangle benchmark. The additional increase for optimized benchmarks with integer affinity tests could not be associated

with a single location in the program. The additional increase for optimized benchmarks with pointer-to-shared affinity is partially a result of the extra branches added while optimizing the affinity test.

2. The percent increase in both events for the non-optimized versions of the benchmarks increases as the number of threads increases indicating that the same operations are being performed on all threads. This repetition of needless operations results in poor parallel performance because the same work is being done on all threads and is not distributed among all threads. While the optimized version also has a significant increase in the two events measured, this increase remains constant when more threads are added. This trend indicates that the operations are being distributed across all threads, resulting in increased parallel performance.

3. In the optimized versions of the benchmarks, there is an increase in the number of fixed-point instructions completed when going from 1 to 2 threads (13X increase for STREAM with integer affinity, 66% increase for STREAM with pointer-to-shared affinity, and 30% increase for upper-triangle). Many of the index and affinity computations contain a division or modulo by the number of threads. When the number of threads is 1, the compiler is able to simplify these computations, resulting in fewer fixed-point instructions executed. However, when the number of threads is 2 or more, these computations must be executed, resulting in an increase in fixed-point instructions completed.

7.4 Chapter Summary

This chapter described code transformations implemented in the compiler that can improve the performance of parallel loop nests in UPC. Both types of affinity tests (integer and pointer-to-shared) can be improved using different techniques. The results presented demonstrate that the optimizations are successful in improving the performance of benchmarks. However, these results include the SOAP optimization presented in Chapter 6.1. The following chapter investigates the impact of improving the code for parallel loop nests with and without SOAP on a variety of benchmarks.

Chapter 8

Performance Analysis

This chapter uses several UPC benchmarks to investigate the performance impact of the code transformations presented in Chapters 6 and 7. The benchmarks were selected based on the requirements of the locality analysis. Thus, every benchmark uses shared arrays and contains **upc_forall** loops. Results for OpenMP and the Message Passing Interface (MPI) are included for benchmarks for which such implementations are available.

Results are presented for SMP, distributed and hybrid environments. The SMP machine used is a 64-way POWER5 machine (1.6 GHz processors) with 512 GB of RAM running AIX 5.3. All experiments were run in exclusive mode on a quiet machine (*i.e.* the benchmark being tested was the only application running on the machine).

Three separate machines were used for the distributed and hybrid environments: *c132*, *v20*, and *bgl*.

c132 is a 4 node IBM Squadron^{TM}cluster. Each node is an 8-way POWER5 machine running at 1.9 GHz and 16 GBytes of memory. All results for c132 were previously published in [9].

v20 is a 32-node cluster where each node is a 16-way POWER5 machine (1.9 GHz processors) with 64 GB of RAM running AIX 5.3. The nodes are connected with 1GB ethernet and a dual-plane HPS switch. The cluster is partitioned into 4 shared nodes (general access) and 28 non-shared nodes (exclusive access). Due to limited access to the full cluster, all initial experimental results were run using the 4 shared nodes. Also, in certain cases the non-optimized benchmarks were only run on the 4 shared nodes due to an extremely high execution time.

bgl is the BlueGene/L installation at Lawrence Livermore National Labs (LLNL). It contains a total of 131072 processors. In these experiments one UPC thread is scheduled

for each BlueGene/L processor. All results for bgl were previously published in [10].

For experiments in a distributed environment the benchmarks were run with one UPC thread per node; for experiments in a hybrid environment the benchmarks were run with one UPC thread for every processor in the node (*e.g.,* on c132 the benchmarks were run with 8 threads-per-node and on v20 the benchmarks were run with 16 threads-per-node). The Parallel Operating Environment (POE) was used to run the experiments in the distributed and hybrid environments on all machines. The *LoadLeveler* batch scheduling software was used to schedule the execution of each experiment.

All experiments used a development version of the UPC compiler and a performance version of the UPC Runtime (debugging was disabled). The optimizations that are applicable in an SMP environment are Shared-Object Access Privatization (*privatization*) and optimizing parallel loop nests (*forall opt*); all combinations of these two optimizations are performed and the results reported. All the optimizations presented in Chapters 6 and 7 have the potential to impact performance on distributed and hybrid architectures. Thus, all of these optimizations were tested on these two architectures. However, due to limited machine access, it was not possible to study the performance of all combinations of these optimizations. Thus, the optimizations are applied successively, in the following order:

1. forall opt;
2. forall opt + privatization;
3. forall opt + privatization + coalescing;
4. forall opt + privatization + coalescing + scheduling;
5. forall opt + privatization + coalescing + scheduling + remote update;

The performance of other combinations of the optimizations was not studied on the distributed and hybrid environments. Results using options that do not apply to the tested environments have been omitted (*i.e.* results for shared-object access coalescing on SMP architectures were not collected). However, any optimization that should benefit a given architecture, but does not, will be discussed.

Unless stated otherwise, all experiments were designed to measure *strong scaling*. That is, the problem size is fixed, the number of UPC threads is adjusted and the resulting performance difference is reported. Unless stated otherwise, all results presented are from a single run and not the average of several runs.

8.1 The STREAM Benchmark

The STREAM benchmark uses 4 types of operations and 3 arrays to measure sustainable memory bandwidth [44]. Each of the four operations is performed in a separate loop:

1. Copy: elements of one array are copied into a different array (c[i] = a[i])

2. Scale: elements of one array are multiplied by a scalar and assigned to a different array (b[i] = SCALAR $*$ c[i])

3. Add: elements of two different arrays are added together and assigned to a third array (c[i] = a[i] + b[i])

4. Triad: elements of one array are multiplied by a scalar value and added to a second array element. The result is stored in a third array (a[i] = b[i] + SCALAR$*$c[i])

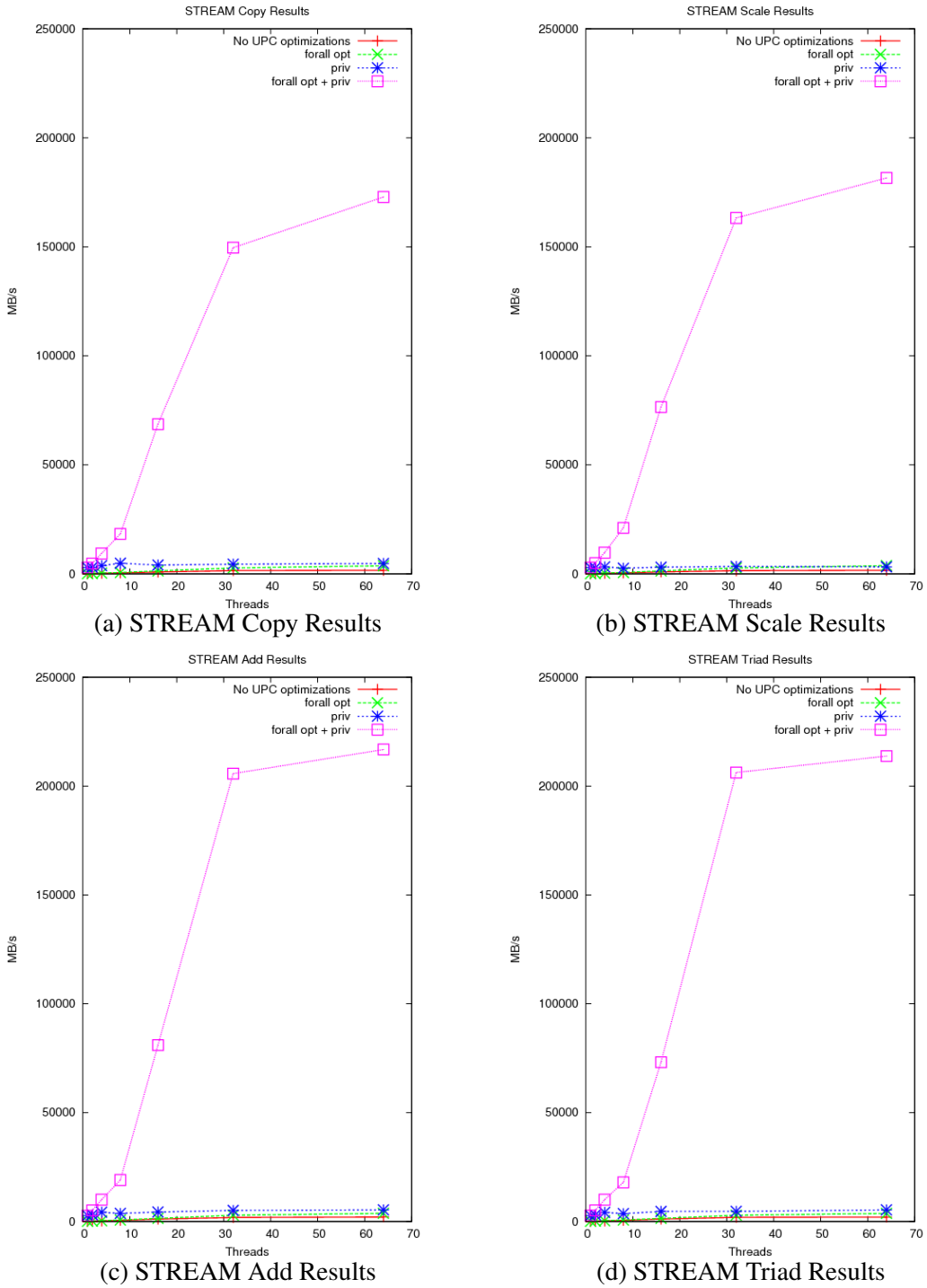
The results are verified by performing a sum reduction on each of the three arrays. The resulting values are compared with a pre-computed expected result based on the initial values in the array and on the number of times each operation is performed.

In the UPC version of STREAM all three arrays are shared. Each shared array contains 20000000 double-precision floating point values, for a total memory footprint of 480MB. Each of the operations is implemented using a **upc_forall** loop. Since the benchmark is designed to measure memory bandwidth, every access should be local. As such, the default (cyclic) blocking factor is used, making both the pointer-to-shared affinity test and the integer affinity test valid to use with the **upc_forall** loops. The *IntegerStream* benchmark was written using an integer affinity test for each **upc_forall** loop and the *PTSStream* benchmark was written using a pointer-to-shared affinity test.

Every thread uses the **upc_all_reduce** collective to perform the local sum reduction of each shared array. These sums are then collected by thread 0 and combined to form a single value for each shared array. The result of this sum is then compared to the expected value to verify the results.

This benchmark is designed to measure the sustainable memory bandwidth and therefore is written such that all accesses to the shared arrays are local to the executing thread. Thus, the only optimizations that should benefit STREAM are the forall optimization and privatization. In addition, since all accesses are local to the executing thread, the performance in a hybrid environment should be similar to the performance in a distributed environment (for the same number of UPC threads).

8.1.1 Shared-Memory Environment



(a) STREAM Copy Results

(b) STREAM Scale Results

(c) STREAM Add Results

(d) STREAM Triad Results

Figure 8.1: IntegerStream benchmark results in a shared-memory environment

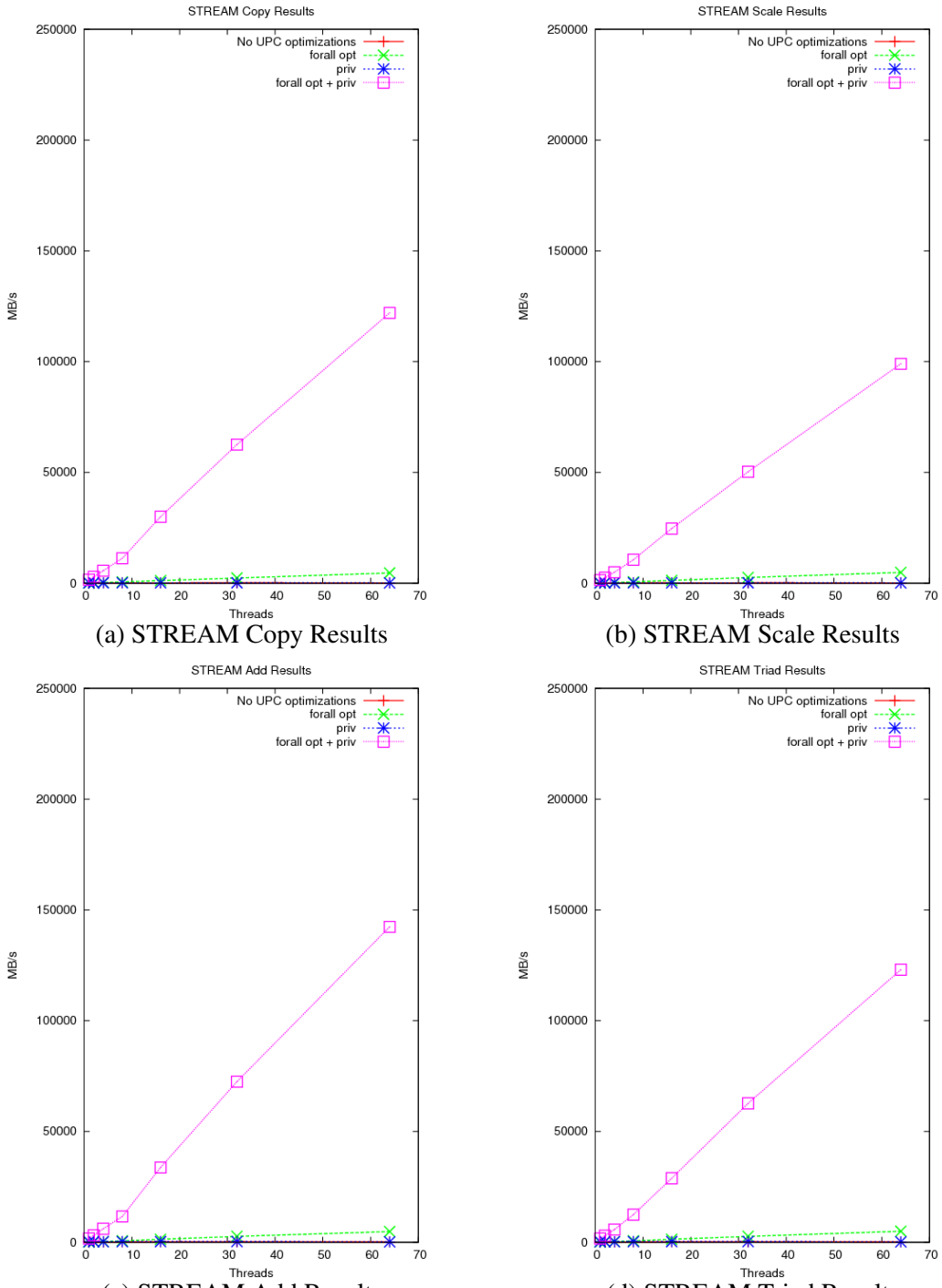
Figure 8.1 shows the results for the four kernels from the IntegerStream benchmark run

in a shared-memory environment. The graphs report the measured bandwidth for each of the four kernels, reported in MB/s (higher numbers are better). These memory bandwidth measurements demonstrate that both the forall optimization and the shared-object privatization optimizations provide a performance improvement over the non-optimized version. For example, for 64 threads, optimizing the **upc_forall** loop results in an 83% improvement and privatizing the shared references results in a 1.62X improvement. However, when both of these optimizations are applied together, the performance improvement increases to more than 104X.

Figure 8.2 shows the memory bandwidth measured by the 4 kernels from the PTSStream benchmark. For 64 threads, optimizing the **upc_forall** loop results in a 25X improvement over the non-optimized version while privatizing the shared references only results in a 10% improvement. This demonstrates that there is more advantage to optimizing the **upc_forall** loops than to privatizing shared references when a pointer-to-shared affinity test is used. This is because the **upc_threadof** function call that is used when the **upc_forall** loop is not optimized becomes the dominating factor in the execution of the **upc_forall** loop. As the number of threads increase, the body of the loop is divided among each of the threads while the **upc_threadof** call is executed by every thread in every iteration of the loop. The effects of the pointer-to-shared affinity test are more pronounced than the integer affinity test because the **upc_threadof** call must query the data layout of the shared array used in the affinity test.

However, optimizing the **upc_forall** loop and privatizing the shared references when using a pointer-to-shared affinity test also yields a significant performance improvement (651X for 64 threads). While this improvement is much larger with a pointer-to-shared affinity test than with the integer affinity test, the absolute performance of the STREAM benchmark using a pointer-to-shared affinity test is actually worse. This is caused by two factors. First, the baseline measurement when using the pointer-to-shared affinity test is much lower than the baseline when using the integer affinity test (188MB/s for the pointer-to-shared affinity test and 2040MB/s for the integer affinity test on 64 threads). This lower baseline explains the high improvement for the pointer-to-shared affinity test. Second, the absolute performance of the integer affinity test is higher because the optimized loop contains fewer instructions that must be executed. The extra instructions required when optimizing a **upc_forall** loop with a pointer-to-shared affinity test yields the difference in absolute performance.

Figure 8.3 shows the speedup of the IntegerStream benchmark compared with the se-



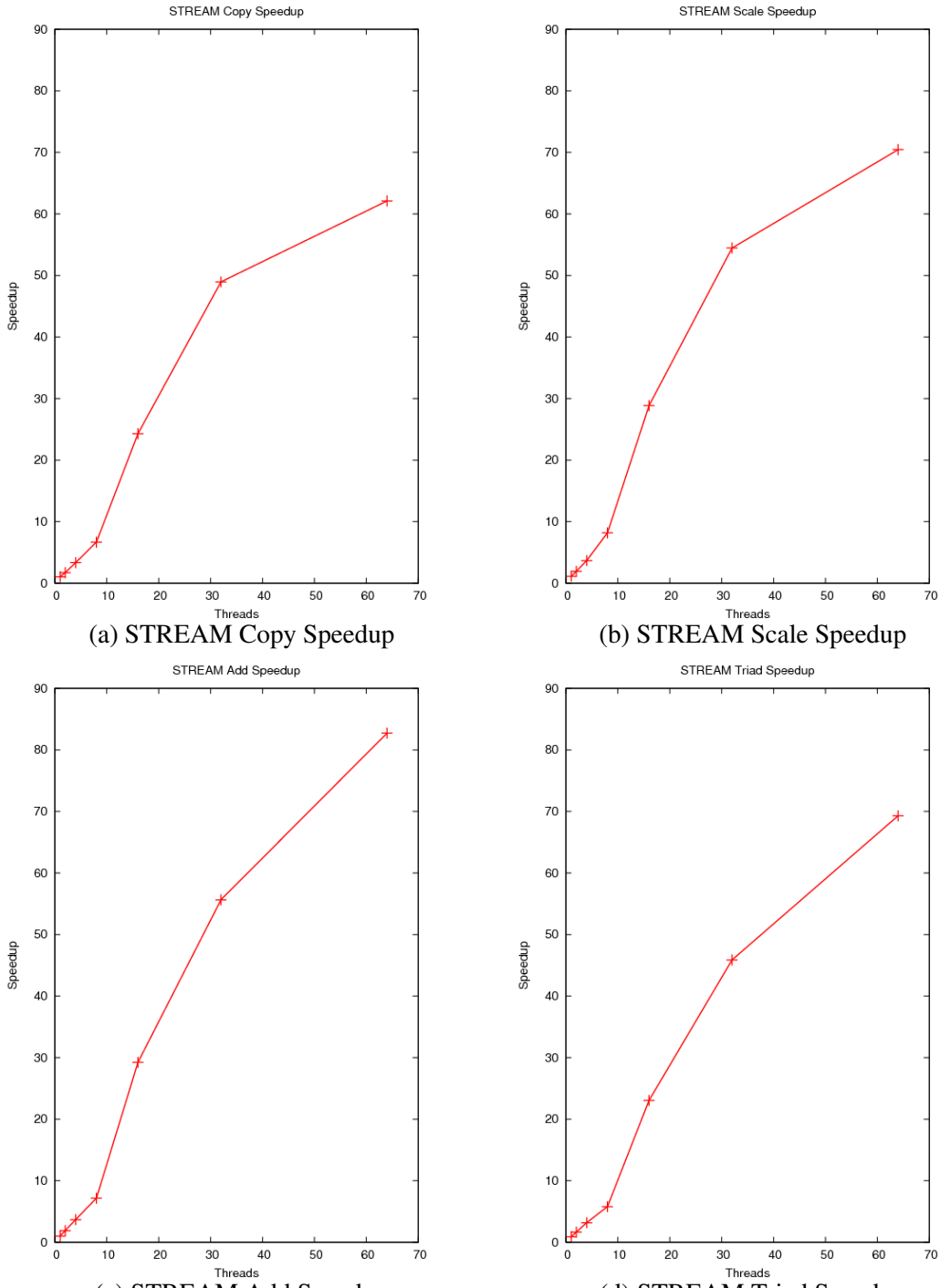
(a) STREAM Copy Results

(b) STREAM Scale Results

(c) STREAM Add Results

(d) STREAM Triad Results

Figure 8.2: PTSStream benchmark results in a shared-memory environment



(a) STREAM Copy Speedup

(b) STREAM Scale Speedup

(c) STREAM Add Speedup

(d) STREAM Triad Speedup

Figure 8.3: IntegerStream speedup over sequential in a shared-memory environment

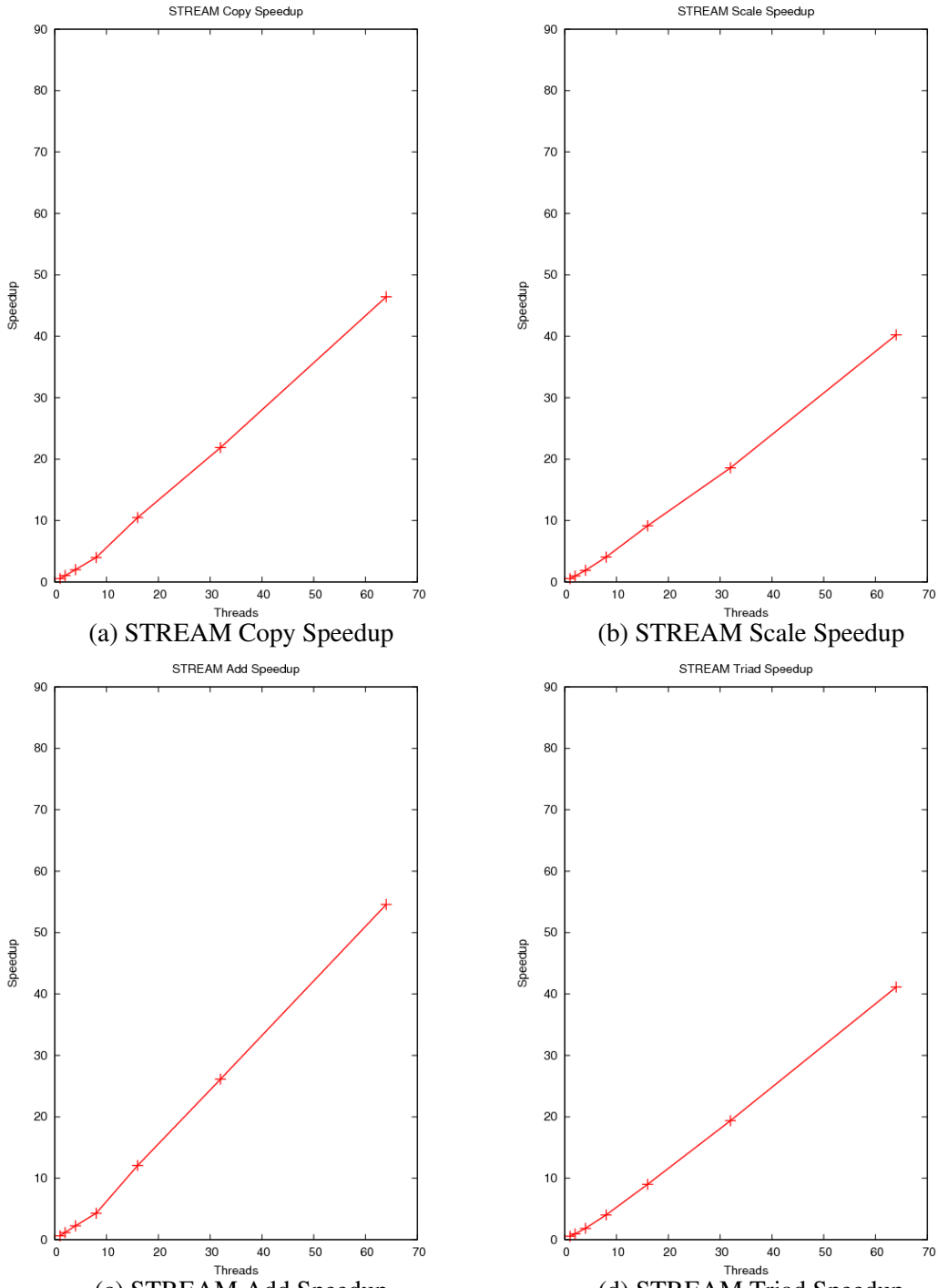
quential C implementation. The results for integer affinity demonstrate an almost perfect speedup for 2, 4, and 8 threads and a super-linear speedup for 16, 32, and 64 threads on all four kernels of the benchmark. The super linear speedup on 16, 32 and 64 threads is attributed to improved cache behaviour as a result of dividing the shared arrays among more threads. For example, the POWER5 processor contains one 36MB L3 cache per chip (2-cores) [53]. When the STREAM Triad kernel is run with 16 UPC threads, each thread will own 1,250,000 elements of each of the three shared arrays for a total memory footprint of 30MB per thread. Furthermore, each thread will be scheduled to run on a separate chip by the operating system. Thus, the three shared arrays will fit into the L3 data cache for each thread, which is not the case when run with 8 UPC threads.

Figure 8.4 shows the speedup of the PTSStream benchmark compared with the sequential C implementation. These results do not exhibit any super linear speedup but do demonstrate reasonable scaling. There should be a performance increase due to improved cache behaviour because, for a given number of threads, the memory footprint of the PTSStream benchmark is the same as the IntegerStream benchmark. However, this performance increase is offset by the additional instructions that are inserted by the forall code transformation for a pointer-to-shared affinity test.

Figure 8.5 compares the optimized IntegerStream and PTSStream benchmarks with the OpenMP implementation. The OpenMP implementation was compiled using a development version of IBM's C compiler, *xlc*, with the same optimizations as the UPC versions (-O3 -qhot). These results demonstrate that both UPC versions of STREAM perform well compared with the OpenMP version. The performance of the OpenMP implementation drops off when run with 64 threads while the IntegerStream benchmark continues to scale nicely. This drop in performance of the OpenMP implementation is attributed to the cost of the OpenMP loop scheduling mechanism and to an insufficient amount of work with 64 threads to cover these costs.

8.1.2 Distributed Environment

The next exprimental evaluation illustrates the effects of the optimizations for the STREAM benchmark in a distributed environment. Only four nodes of the cluster were used because of the long time it takes to run the STREAM benchmark without all optimizations enabled. Figures 8.6 and 8.7 show the IntegerStream and PTSStream benchmarks respectively. Again, for the integer affinity test we see that shared-object access privatization has a much larger benefit than optimizing the forall loop. We also observe that optimizing the



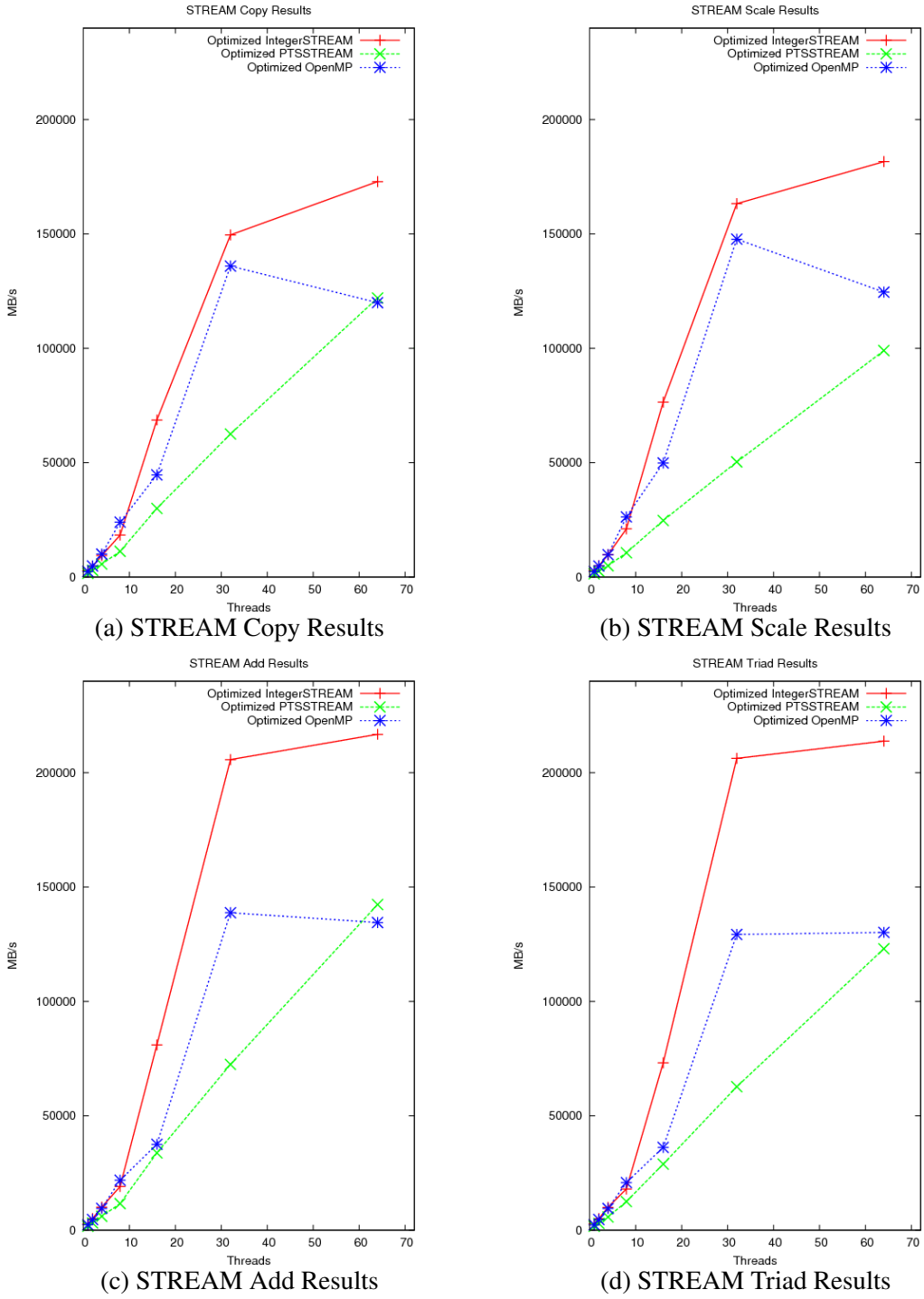
(a) STREAM Copy Speedup

(b) STREAM Scale Speedup

(c) STREAM Add Speedup

(d) STREAM Triad Speedup

Figure 8.4: PTSStream speedup over sequential in a shared-memory environment



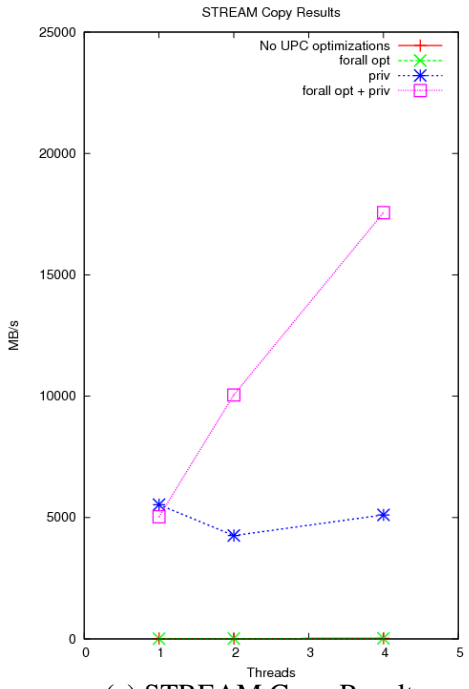
(a) STREAM Copy Results

(b) STREAM Scale Results

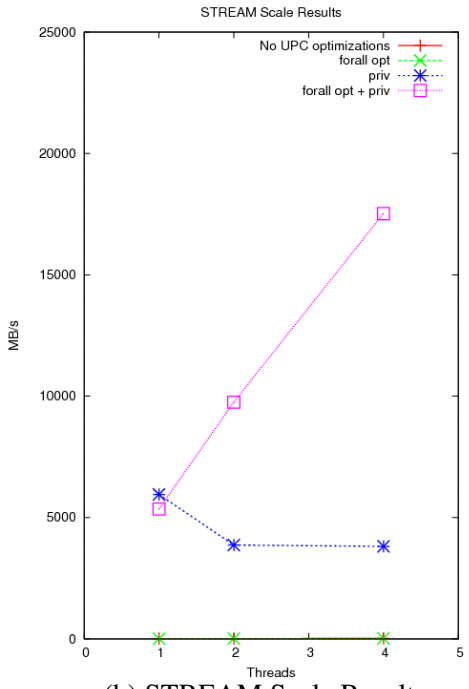
(c) STREAM Add Results

(d) STREAM Triad Results

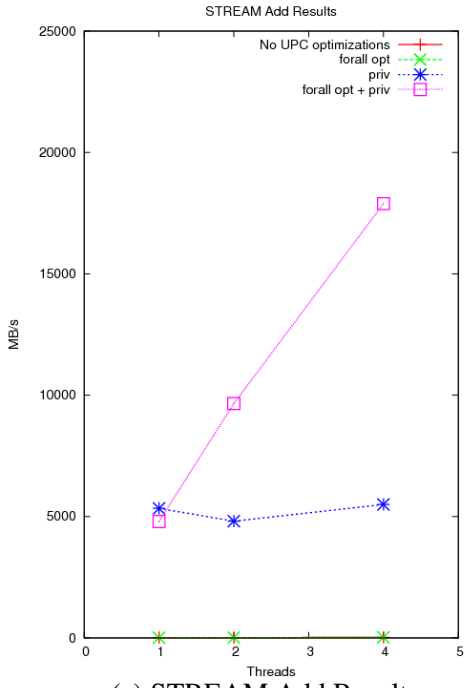
Figure 8.5: Optimized UPC and OpenMP STREAM benchmark results in a shared-memory environment



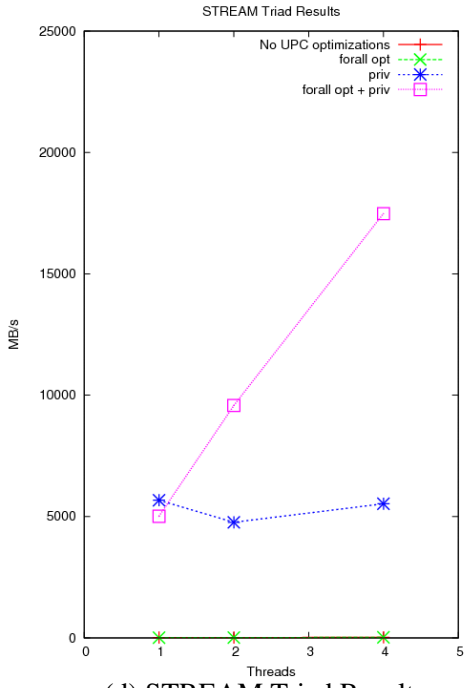
(a) STREAM Copy Results



(b) STREAM Scale Results

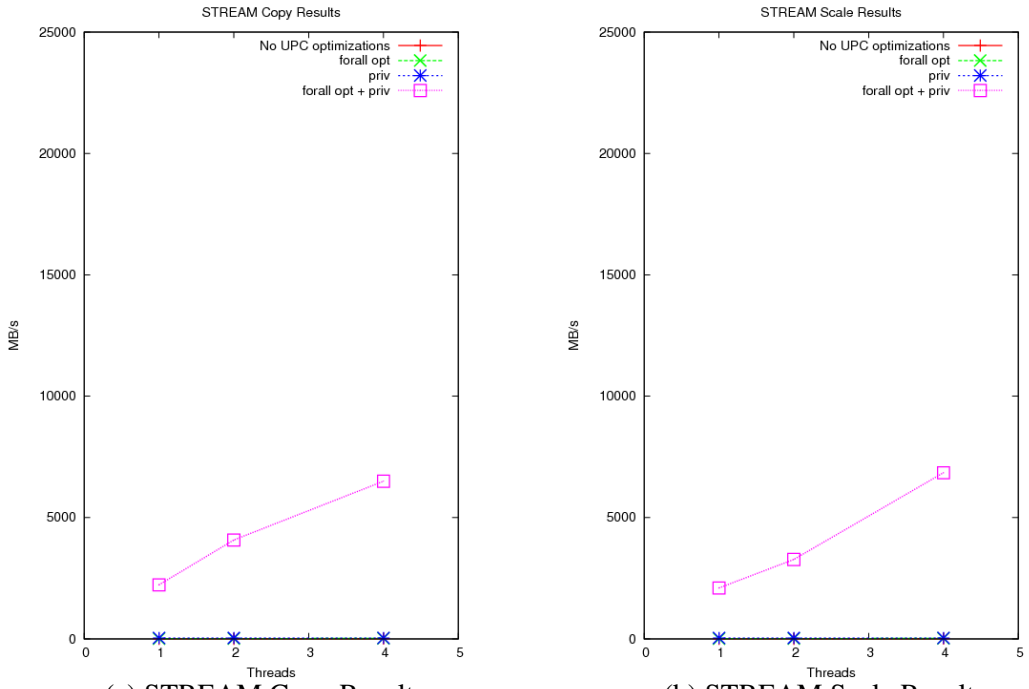


(c) STREAM Add Results



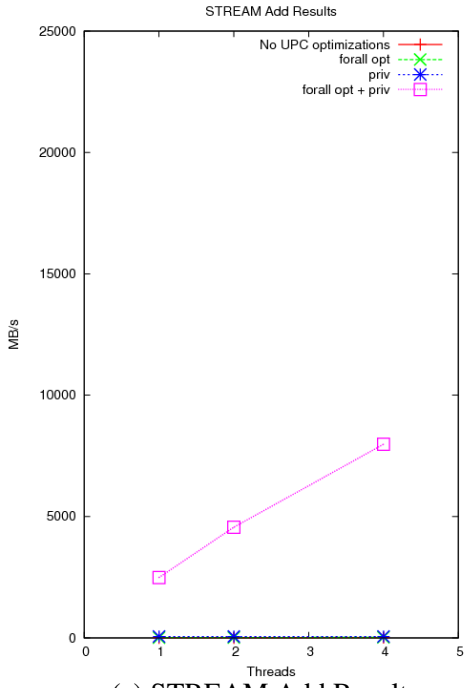
(d) STREAM Triad Results

Figure 8.6: IntegerStream benchmark results on distributed v20

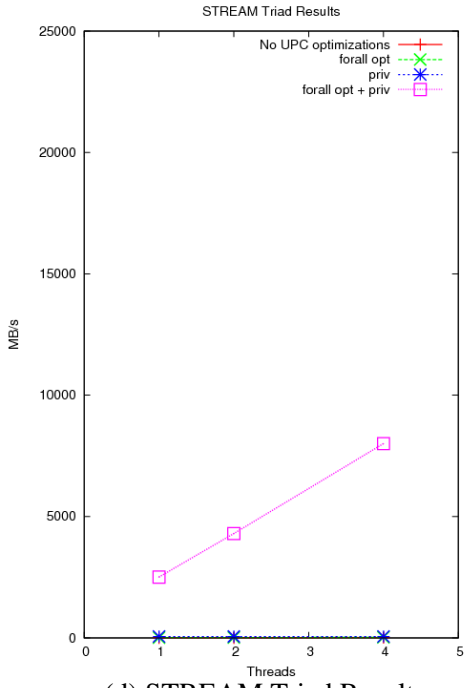


(a) STREAM Copy Results

(b) STREAM Scale Results



(c) STREAM Add Results



(d) STREAM Triad Results

Figure 8.7: PTSStream benchmark results on distributed v20

upc_forall loop and privatizing the shared-object accesses results in a significant performance improvement.

In the distributed environment there is no communication because all computations are performed on local shared data. Therefore no results were collected for the SOAC and SOAS optimizations.

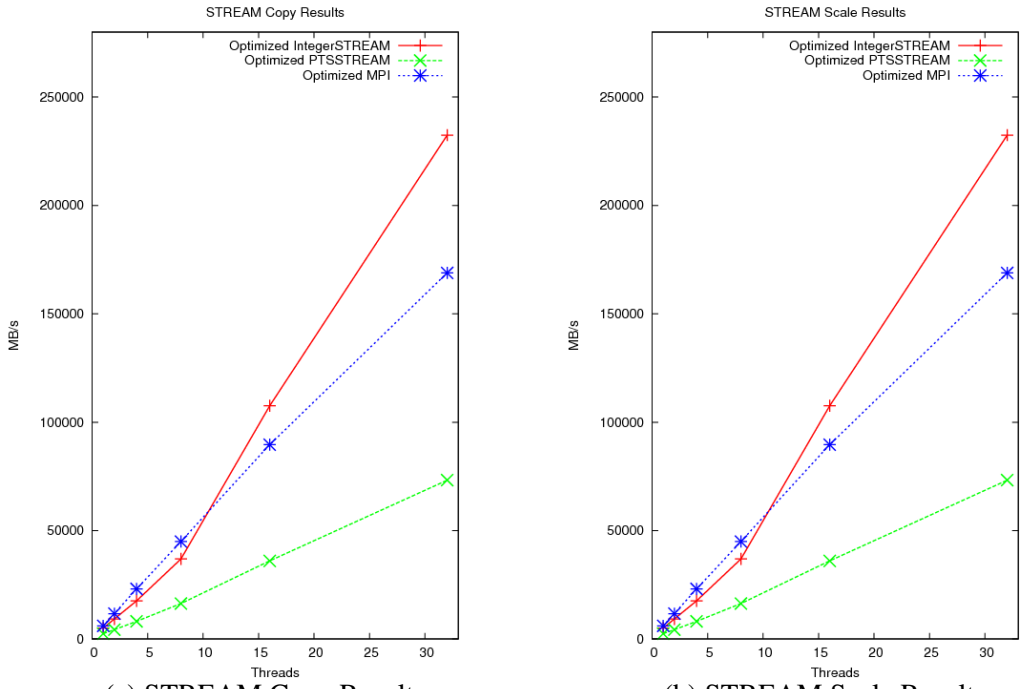
As in the shared-memory environment, the absolute performance of the STREAM benchmark implemented using a pointer-to-shared affinity test is much lower than the performance when using an integer affinity test. For example, the optimized STREAM triad kernel using an integer affinity test delivers a memory bandwidth of 17500MB/s while the same kernel using a pointer-to-shared affinity test delivers 8000MB/s. This difference in performance is due to the additional code required when optimizing parallel loops using a pointer-to-shared affinity test. While the compiler is able to simplify the conditional expressions that are inserted before the loop, to compute the lower and upper bounds of the strip-mined loop, it cannot completely remove them and the performance is reduced as a result.

Figure 8.8 compares the performance of the optimized STREAM benchmark with a reference MPI implementation obtained from [36]. The MPI implementation was compiled using a development version of *xlc*with the same optimizations as the UPC benchmarks (-O3 -qhot). These results show that the IntegerStream benchmark outperforms the PTSStream benchmark. In addition, the performance of the IntegerStream benchmark is comparable to the performance of the MPI implementation.

Threads	Performance	Memory	Efficiency
	(GB/s)	TBytes	(%)
1	0.73	0.000128	100
2	1.46	0.000256	100
4	2.92	0.000512	100
64	46.72	0.008192	100
65536	47827.00	8.000000	100
131072	95660.77	8.000000	100

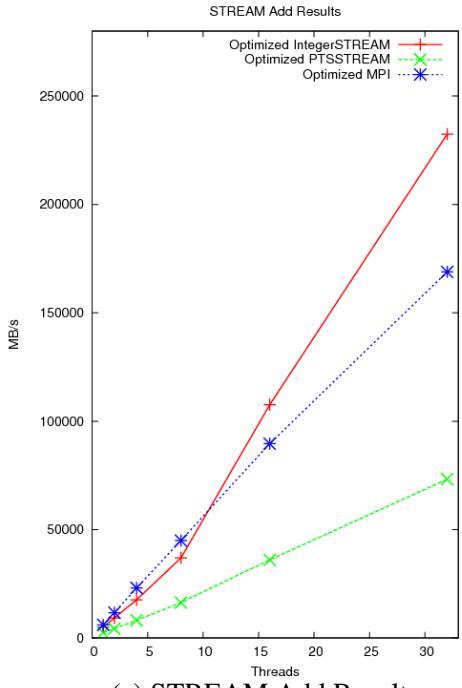
Table 8.1: STREAM triad performance results on bgl, from [10]

Table 8.1 shows the memory bandwidth measured for the Triad kernel from the optimized IntegerStream benchmark run on bgl [10]. These measurements were done to demonstrate the *weak scaling* of the UPC STREAM benchmark. Thus, the problem size and the number of UPC threads is varied to keep the execution time the same. The array sizes were

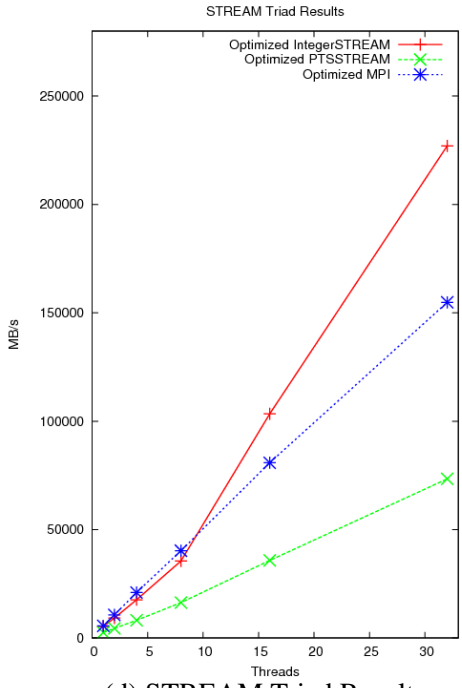


(a) STREAM Copy Results

(b) STREAM Scale Results



(c) STREAM Add Results



(d) STREAM Triad Results

Figure 8.8: Optimized UPC and MPI STREAM benchmark on distributed v20

selected to use half of the available memory based on the number of UPC threads. These results show that the IntegerStream benchmark is able to scale weakly up to a very large number of threads with the help of the forall and SOAP optimizations.

8.1.3 Hybrid Environment

To test the effect of each optimization in a hybrid environment, the IntegerStream and PTSStream benchmarks were run on 4 nodes of the cluster, with each node running 16 threads. Measurements were collected for 16, 32, 48 and 64 threads. Figures 8.9 and 8.10 present the memory bandwidth measured by IntegerStream and PTSStream respectively. For the hybrid architecture we observe small benefits when running STREAM with each individual optimization but a significant performance improvement when all optimizations are combined.

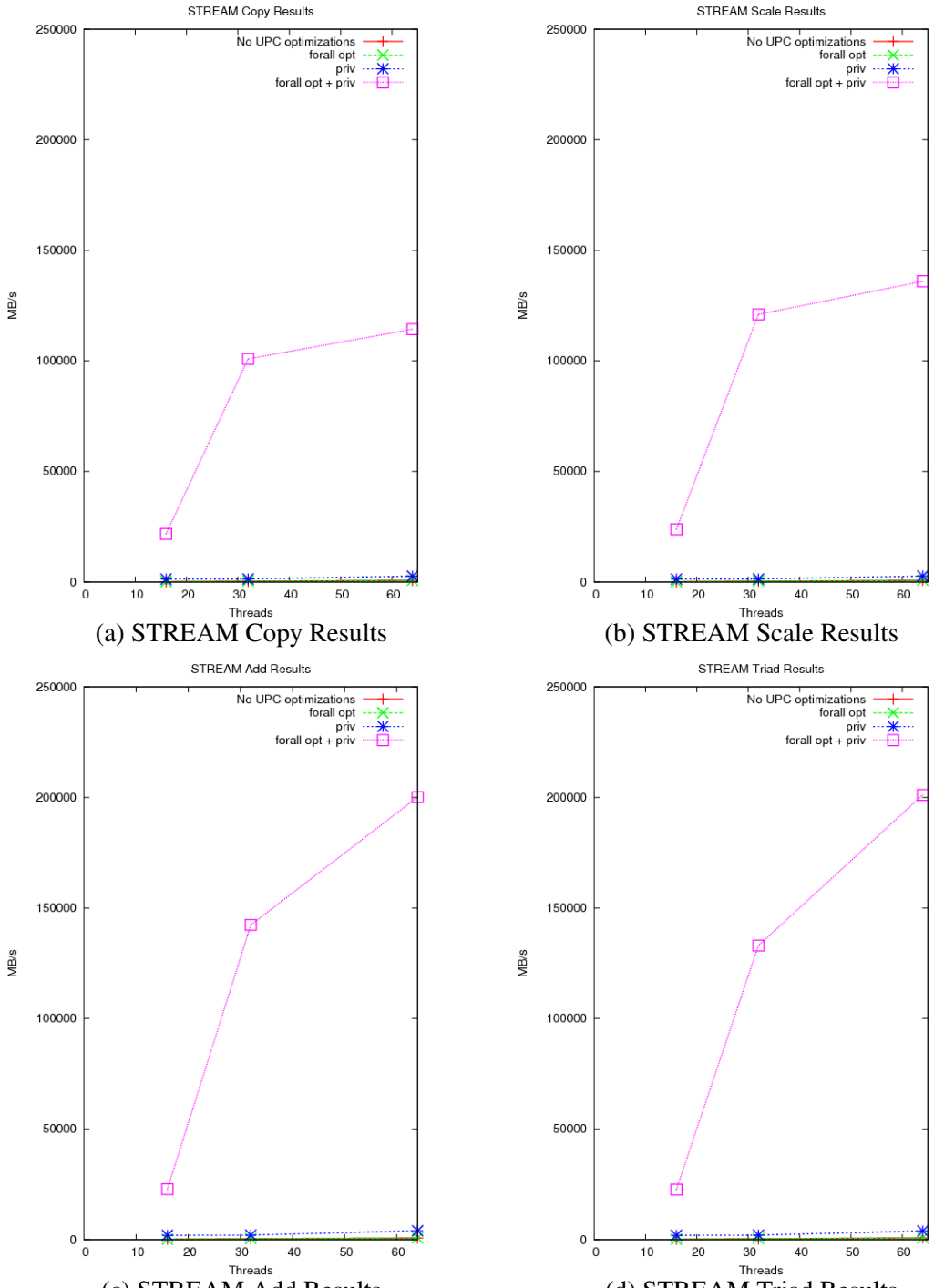
Figure 8.11 shows the performance of the optimized IntegerStream and PTSStream benchmarks in a hybrid environment. As with the distributed version, the absolute performance of IntegerStream is much better than the performance of PTSStream because of the additional instructions generated when optimizing a forall loop with a pointer-to-shared affinity test.

8.1.4 Discussion

The memory bandwidth measured by the STREAM benchmark demonstrates the benefit of the locality analysis and locality optimizations when operating on local shared-objects. The compiler is able to identify all accesses as local in all environments and is able to correctly privatize all accesses. These code transformations lead to significant performance improvements over the non-optimized versions of the benchmark. Furthermore, the optimized UPC version of STREAM outperforms the optimized OpenMP version in a shared-memory environment.

In a distributed memory environment, the performance of the optimized UPC version is comparable to the performance of the optimized MPI implementation when the number of threads is a power of two. When the number of threads is not a power of two, the address computations required to locate the shared-objects in memory cannot be optimized as efficiently, resulting in significantly lower performance than the MPI implementation. These results demonstrate the importance of optimizing the address computations used to determine the location of the shared-objects in memory.

We expected to see benefit from running the benchmark, with the same number of



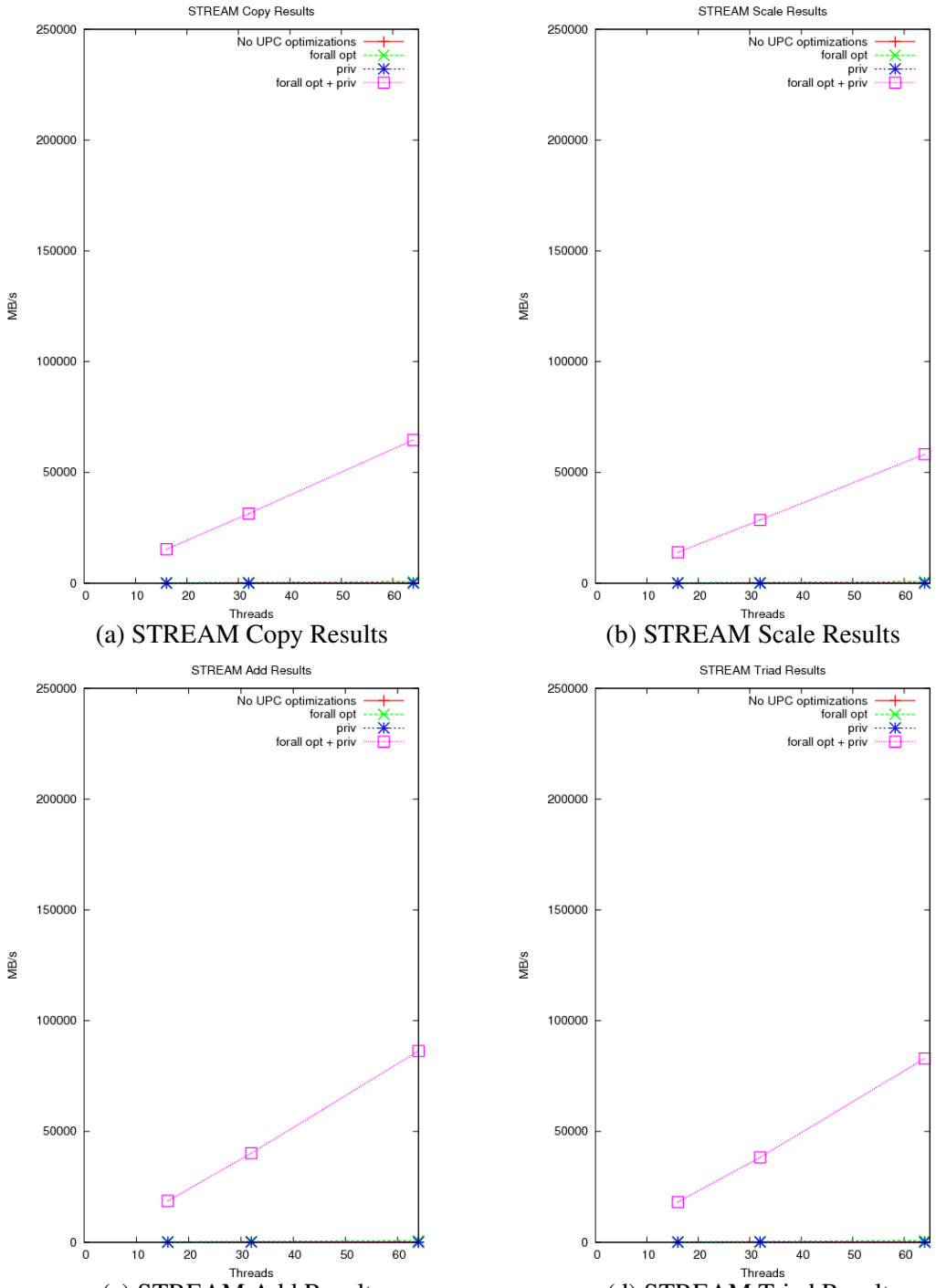
(a) STREAM Copy Results

(b) STREAM Scale Results

(c) STREAM Add Results

(d) STREAM Triad Results

Figure 8.9: IntegerStream benchmark results on hybrid v20



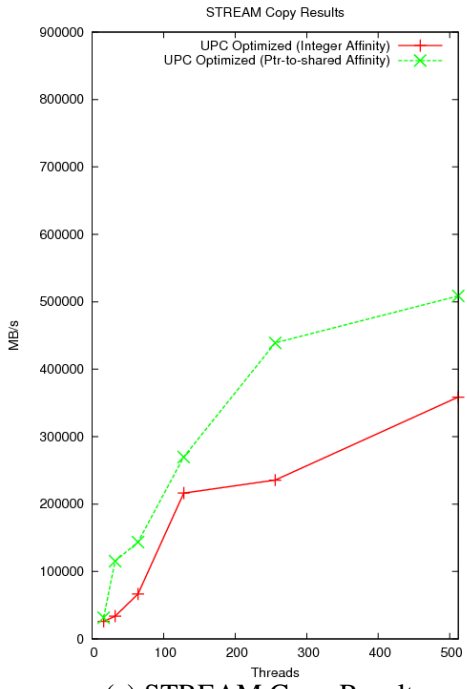
(a) STREAM Copy Results

(b) STREAM Scale Results

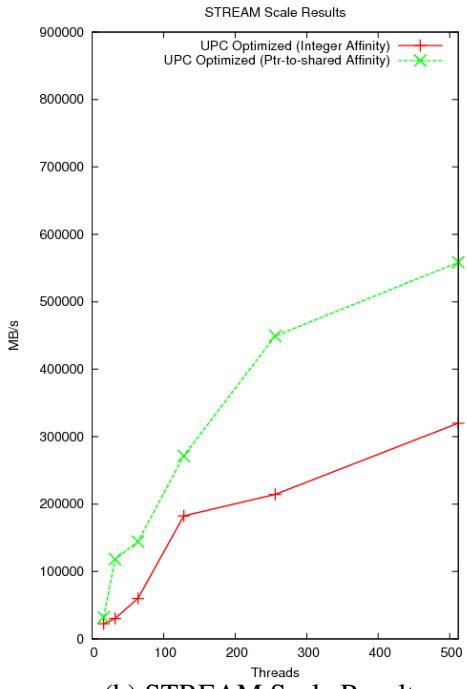
(c) STREAM Add Results

(d) STREAM Triad Results

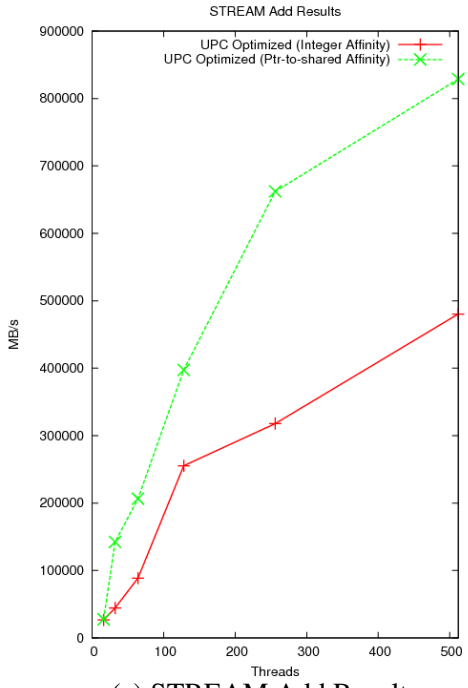
Figure 8.10: PTSStream benchmark results on hybrid v20



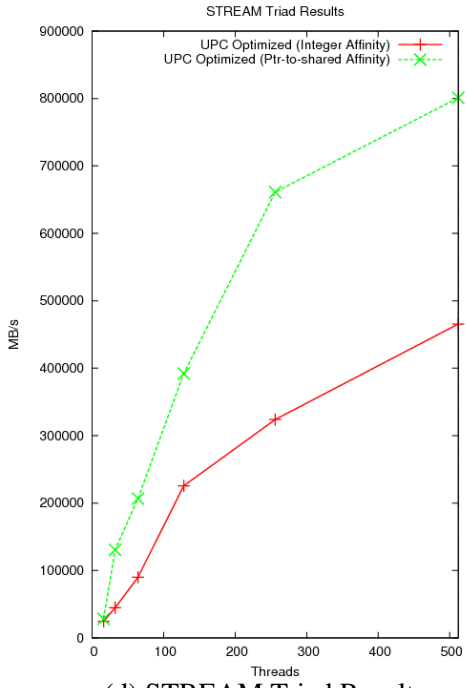
(a) STREAM Copy Results



(b) STREAM Scale Results



(c) STREAM Add Results



(d) STREAM Triad Results

Figure 8.11: Optimized STREAM benchmark results on hybrid v20

threads, in a hybrid environment as opposed to a distributed environment because all computations are performed on local shared objects. In fact, we observe the opposite: the performance for both IntegerStream and PTSStream is better in a distributed environment than in a hybrid environment. For example, the memory bandwidth measured by running the Triad kernel of the IntegerStream benchmark in a distributed environment with 16 threads is 103356 MB/s while running the same kernel in a hybrid environment with 1 node (16 threads) is 28209 MB/s. Thus, the performance of the IntegerStream triad kernel run in a hybrid environment is almost 27% of the performance when run in a distributed environment. These differences are attributed to the cache performance of the machine. In a distributed environment, each node is running a separate UPC thread and thus each thread does not need to share the cache resources with other UPC threads. On the other hand, in a hybrid environment multiple UPC threads must share the cache resources thereby decreasing the performance of the cache. The same effect is observed for the PTSStream benchmark, although the performance difference is not as great.

The performance of the STREAM benchmark is dictated by the code generated during the SOAP optimization because all of the computations performed by this benchmark are local. As discussed in Chapters 4.3.3 and 6.1 the code generated by the SOAP algorithm uses integer division and modulo instructions in the index-to-offset computations for each privatized access. The backend component of the compiler (TOBEY) performs a strength reduction optimization on these computations and replaces the division and modulo instructions by bit-shift instructions when divisors are a power of two. Because `THREADS` is used as the divisor in many of these equations, TOBEY is able to generate more efficient code when the number of threads is a power of two.

	Distributed			Hybrid		
	16	31	32	256	496	512
IntegerStream	103356	47300	227052	661171	253050	801140
PTSStream	35741	25578	73394	323884	210791	465602

Table 8.2: Results of the optimized STREAM triad kernel (MB/s)

When the number of threads is not a power of two, the performance of the STREAM benchmark suffers. For example, in a distributed environment when run on 31 threads, the performance of the optimized (*forall opt + SOAP*) IntegerStream Triad kernel is 47300 MB/s compared to 103356 MB/s when run on 16 threads and 227052 MB/s on 32 threads. Similarly, in a hybrid environment, the optimized IntegerStream Triad kernel achieves

253050 MB/s on 496 threads compared with 661171 MB/s on 256 threads and 801140 MB/s on 512 threads. Table 8.2 shows the memory bandwidth measured by the optimized IntegerStream and PTSStream Triad kernels using power-of-two and non-power-of-two threads. These results demonstrate that running the IntegerStream and PTSStream benchmarks with a higher number of threads does not always produce better results because the performance is highly sensitive to the quality of the code generated by the compiler. These results underscore the significance of the strength reduction optimization and illustrate that it is crucial that the compiler be able to optimize the index-to-offset computations generated by the SOAP optimization in order to achieve performance comparable with other implementations.

8.2 Sobel Edge Detection

The Sobel operator is a discrete differentiation operator used in image processing. It computes an approximation of the gradient of the image intensity function. At each point in an image, the result of the Sobel operator is either the corresponding gradient vector or the norm of the vector [2]. The Sobel operator performs a 9-point *stencil* operation where the 9 adjacent elements to a given array location are used in the computation. Thus, the parallel implementation of the Sobel operator can contain both local and remote operations.

In the UPC version the image is represented as a two-dimensional shared array. The shared array is blocked by rows unless stated otherwise (each UPC thread is assigned a row of the shared array). A parallel loop nest iterates over the shared array applying the Sobel operator to each element in the array. The innermost loop is a **upc_forall** loop using a pointer-to-shared affinity test. The results presented here use an image size of 4000x4000 bytes for a total memory requirement of 16MB.

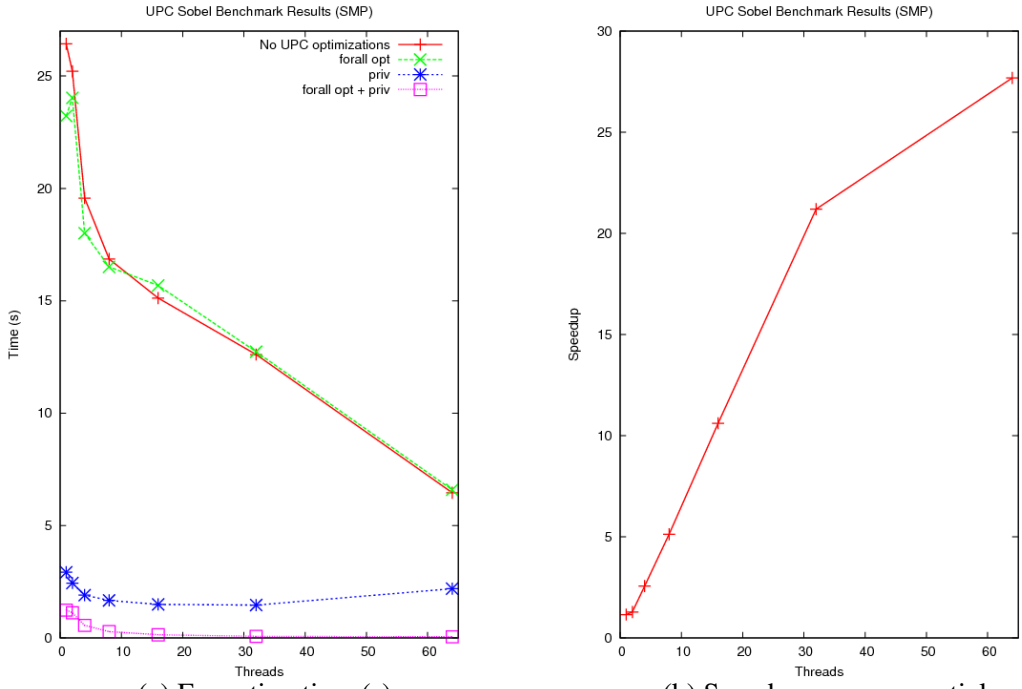
For small problems, test images are read from a file and used to test both the sequential and the parallel implementations. The resulting image is written to a file and a file difference on the outputs from the sequential and parallel implementations is performed to verify the results. For large problems a random image is created using the rand function. In the parallel implementation, this image is created by a single thread to ensure that the initial image is identical to the image created in the sequential implementation. The timing of the benchmark is restricted to the function that performs the Sobel operator and does not include the time to initialize the image or to write the result to file.

Because the Sobel benchmark contains both local and remote accesses and because

the remote accesses are to neighbouring threads, we expect to see a benefit from running the benchmark in a hybrid environment compared to a distributed environment. This is because some of the remote accesses for a given thread will be shared objects owned by another thread on the same node. These cases should be identified by the SOAP optimization resulting in more shared accesses being privatized and fewer remote shared-object accesses.

8.2.1 Shared-Memory Environment

Figure 8.12(a) shows results for the Sobel benchmark run in a shared-memory environment. The graph reports runtime in seconds (lower is better). The execution times indicate that there is no benefit from performing only the forall optimization while privatizing the shared-object accesses results in a reasonable performance improvement. This is because the loop nest contains 12 accesses to a shared array in every iteration and thus the computation is dominated by the shared accesses. As seen with the STREAM benchmark, the best performance is achieved by performing both the parallel loop optimization and the SOAP optimization.



(a) Execution time (s) (b) Speedup over sequential

Figure 8.12: The Sobel benchmark in a shared-memory environment

Figure 8.12(b) shows the speedup of the optimized UPC implementation against the sequential C implementation. With 64 threads the optimized UPC benchmark is able to

achieve a 27X speedup over the sequential C version. As mentioned previously, the loop nest contains 12 accesses to a shared array in every iteration. While the SOAP optimization is able to significantly reduce the overhead of accessing local shared data, the accesses still have a higher cost than a memory access in the sequential version. This higher cost is due to the additional addressing computations required to convert a shared array index into an offset value. These additional computations are the limiting factor in the speedup.

8.2.2 Distributed Environment

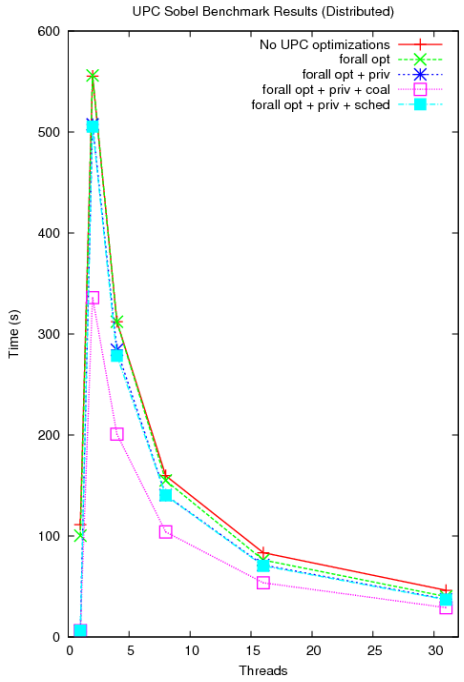


Figure 8.13: The Sobel benchmark on distributed v20

Figure 8.13 shows the execution time of the UPC Sobel benchmark run in a distributed environment. These results were run on 31 nodes of the cluster (32 nodes were not available). These execution times demonstrate that both the parallel loop optimization and the SOAP optimization have a limited impact on the performance of Sobel in a distributed system. This benchmark performs a stencil computation, accessing the 9 immediate neighbours of a given shared array element. The shared array is distributed by rows, causing 6 of these 9 accesses to be remote. Thus, this benchmark has a high number of remote accesses. The cost of these accesses contributes significantly to the execution time of the program and thus optimizing the three local accesses and the overhead of the parallel loop makes little

difference in the overall execution.

On the other hand, coalescing the remote shared accesses yields a significant performance improvement – roughly 37% on all threads. This improvement is attributed to the reduction in the number of accesses to retrieve remote shared data and hence a reduction in the number of messages.

Because all remote shared-object accesses are coalesced by SOAC, there is no benefit to performing SOAS in addition to SOAC. Instead, Sobel was compiled with forall opt + privatization + scheduling to compare the benefits of SOAS with SOAC. The results show no benefits from SOAS which indicates that there is not enough work in the loop body to hide the cost of remote accesses.

8.2.3 Hybrid Environment

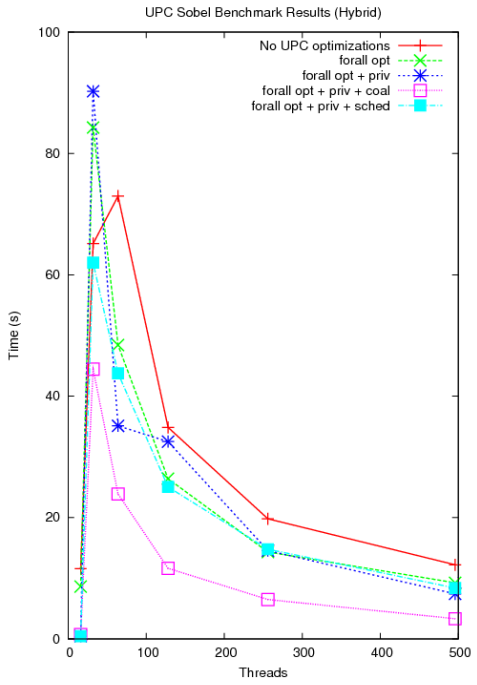


Figure 8.14: The Sobel benchmark on hybrid v20

Figure 8.14 shows the execution time of the UPC Sobel benchmark run in a hybrid environment. These results were also run on 31 nodes of the cluster, resulting in a total of 496 UPC threads. These execution times are a reasonable performance gain, between 25% and 30%, from the forall optimization on all threads except 32. For 32 threads, a performance degradation of 30% was observed. We do not have an explanation for this performance

degradation. Aside from 32 and 64 threads, there was no appreciable difference in the performance when the SOAP optimization was applied in addition to the forall optimization. We have no reasonable explanation for these performance fluctuations on 32 and 64 threads. A plausible hypothesis is that these results could be due to *LoadLeveler* scheduling the jobs on two of the shared nodes in the cluster, which may have had other workloads running on them.

There is a significant performance improvement when the SOAC optimization is performed in addition to the forall optimization and SOAP (94%, 32%, 67%, 66%, 67%, and 75% on 16, 32, 64, 128, 256 and 496 threads respectively). These results indicate that SOAC is able to reduce the number of remote shared-object accesses, thereby improving performance.

As with the distributed memory environment, there is no benefit from the SOAS optimization.

As predicted, we observe that running the Sobel benchmark in a hybrid environment yields better performance results than running the same number of threads in a distributed environment. This is because a larger percentage of the accesses will be to the same node, thereby reducing the number of messages required to access remote shared-objects.

8.3 RandomAccess

The RandomAccess benchmark is designed to measure the time to access random address in memory. The benchmark issues a number of Read-Modify-Write (RMW) operations to random locations in a shared array. A **upc_forall** loop with integer affinity is used to distribute the update operations among all threads. The updates are *exclusive-or* operations, which allows for easy verification of the result. By running the benchmark twice, the original contents of the shared array are restored by the second execution. Verification is part of the benchmark implementation. To keep the source code simple, the implementation uses the simplest possible algorithm. The resulting UPC code has 111 lines.

8.3.1 Shared-Memory Environment

Figure 8.15 shows the results of performing the forall optimization and the SOAP optimization on the RandomAccess benchmark. The benchmark uses a shared array containing 2^{27} unsigned long long integers, for a total memory requirement of 1073MB. The graph reports Millions of Updates Per Second (MUPS) as a function of the number of UPC threads (higher number is better).

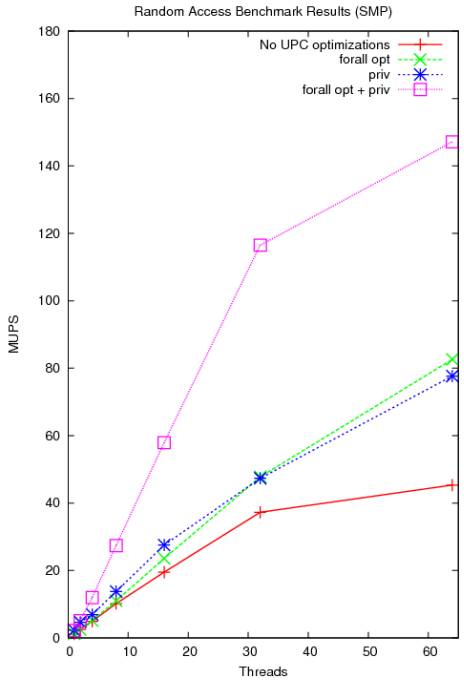


Figure 8.15: RandomAccess results in a shared-memory environment

On shared memory, since all threads map to the same node the locality analysis is able to determine that all accesses are local. Thus, the SOAP algorithm is able to optimize all of the shared references. As a result, the performance increases as seen in Figure 8.15. In addition, since the **upc_forall** loop used in the benchmark uses an integer affinity test, the forall optimization has an equal impact on the performance improvement. As seen with previous benchmarks, performing both optimizations together results in a significant performance improvement and represents more than the additive effects of performing each optimization independently. For example, on 16 threads the forall optimization resulted in a 21% improvement and the SOAP optimization resulted in a 41% improvement. However, performing both optimizations together results in a 1.97X improvement.

Figure 8.16 shows the relative performance improvement over the non-optimized performance that is obtained by applying the forall and SOAP optimizations. These results show the importance of optimizing the **upc_forall** loops and of removing the affinity test as the number of threads increase. In fact, with 64 threads we observe that there is more improvement from the forall optimization than from SOAP. Optimizing the forall loop removes the execution of a branch statement from every iteration of the loop in every thread. With 64 threads, the presence of this branch statement becomes more influential (in terms

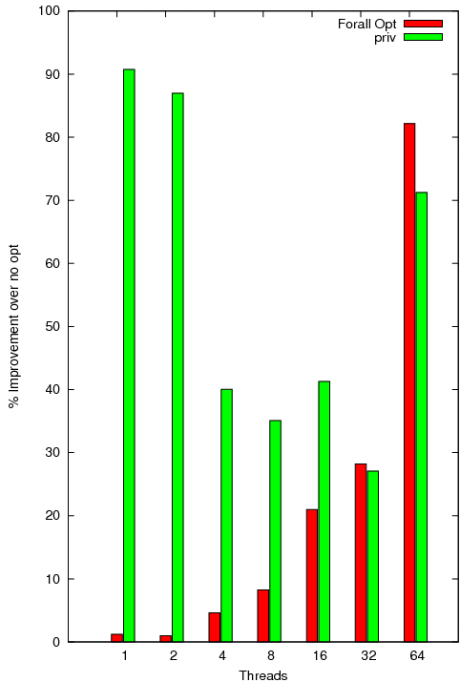


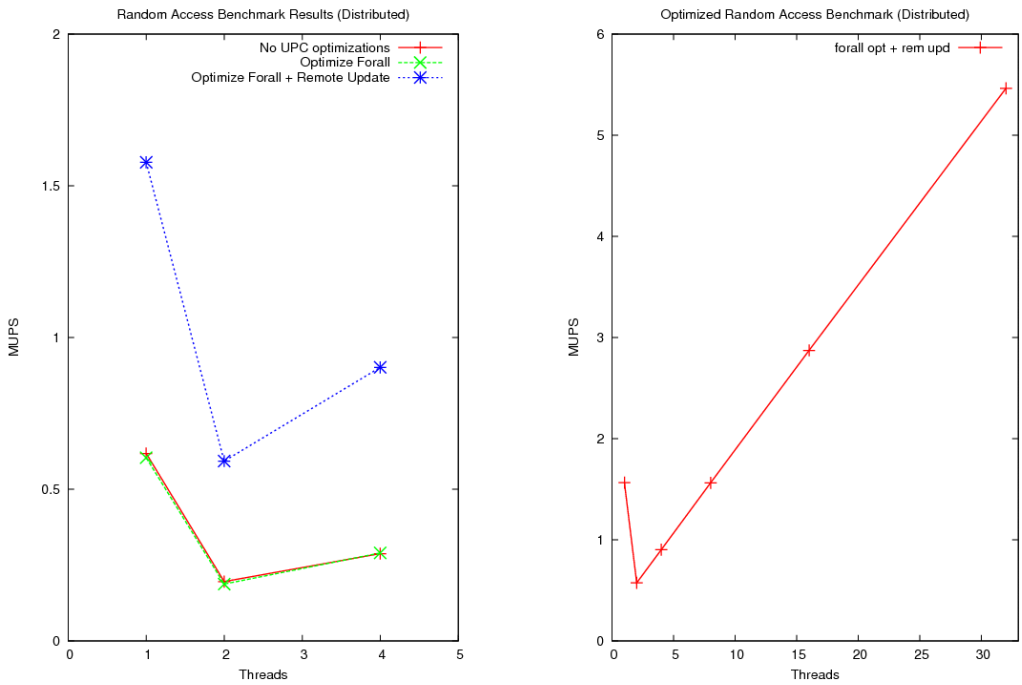
Figure 8.16: Relative performance improvement of optimizations

of execution time) than the call to the RTS to access the shared data.

8.3.2 Distributed Environment

Because of the random nature of the accesses in RandomAccess, the compiler is not able to determine the locality of the shared references when the benchmark is run in a distributed environment. Thus, all shared references are placed in the *UNKNOWN* entry in the shared reference map. As a result, the SOAP, SOAC and SOAS optimizations are not applicable. However, the compiler is able to identify the shared references as candidates for the Remote Update optimization. In addition, the forall optimization is still able to optimize the **upc_forall** loop used to distribute the accesses across threads.

Figure 8.17(a) shows the impact of the forall and remote update optimizations on the RandomAccess benchmark. Again, due to the long execution times of the non-optimized version of RandomAccess, these results are collected using the 4 shared nodes of the v20 cluster. These results demonstrate that optimizing the forall loop provides no benefit to the execution time of the program, while the remote update optimization yields approximately a 2X improvement. This improvement correlates to the reduction in the number of messages sent when the remote update optimization is used.



(a) Optimizations applied to RandomAccess (4-nodes)

(b) Optimized RandomAccess (32-nodes)

Figure 8.17: RandomAccess results on distributed v20

Figure 8.17(b) shows the performance, measured in MUPS, for the optimized RandomAccess benchmark run on the full 32 nodes of the v20 cluster. The degradation is a result of the messages required to update the remote shared-objects. When run with 8 threads, this degradation is offset and the performance is similar to the single-thread performance. The benchmark continues to scale nicely up to 32 nodes, reaching a maximum performance of 5.46 MUPS.

To determine how the RandomAccess benchmark scales to a large number of processors, it was run on BlueGene/L [10]. We arranged for 50% of the memory to be used in order to test weak scaling. With perfect scaling, a RandomAccess run should take about 300 seconds regardless of the number of processors in which it is running. Since performance does not scale linearly (see the efficiency column in Table 8.3), the total runtime increases on larger runs.

Table 8.3 show the absolute and scaling performance of RandomAccess measured on up to 64 racks of BlueGene/L. The following definition of efficiency for N processors is used to measure scaling performance:

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

Threads	Performance	Memory TBytes		efficiency
	(GUPS)	used	total	(%)
1	5.4E-4	0.000128	0.000512	100
2	7.8E-4	0.000256	0.000512	72
4	1.3E-3	0.000512	0.001	61
64	0.02	0.008192	0.016	61
2048	0.56	0.250000	0.500	51
4096	1.11	0.500000	1.000	50
8192	1.70	1.000000	2.000	38
16384	3.36	2.000000	4.000	38
32768	6.10	4.000000	8.000	34
65536	11.54	8.000000	16.000	33
131072	16.72	8.000000	16.000	23

Table 8.3: RandomAccess performance (in GUPS) and memory usage on bgl, from [10]

Where T_{single} is the time for the UPC version run with one thread and $T_{parallel}$ is the time for the UPC version run with multiple threads. The benchmark is affected by two performance-limiting effects. At low numbers of processors the gating factor is communication latency. For large numbers of processors the gating factor becomes the torus network's cross-section bandwidth. The cross-section bandwidth of a booted BlueGene/L partition is determined by its longest torus dimension; cubic partitions have the highest cross-section bandwidth relative to the number of nodes that they contain.

The largest machine configuration on which we ran RandomAccess (128K processors), has an effective cross section of $32 \times 32 \times 2 \times 2 = 4096$ network links. This results from the 32×32 geometry of the cross-section and two doubling factors: each link is bi-directional and the machine is a 3D torus, not a mesh.

Thus, cross-section bandwidth for the 128K processor machine configuration can be determined as the product of the wire speed, 175 MBytes/s, and the number of links in the cross-section, yielding approximately 716.6 GB/s.

Given that RandomAccess update packets end up as 75 bytes each on the wire, and that only half of all RandomAccess updates have to travel through the cross section, the maximum theoretical Giga Updates Per Second (GUPS) number for the benchmark on this configuration can be calculated as:

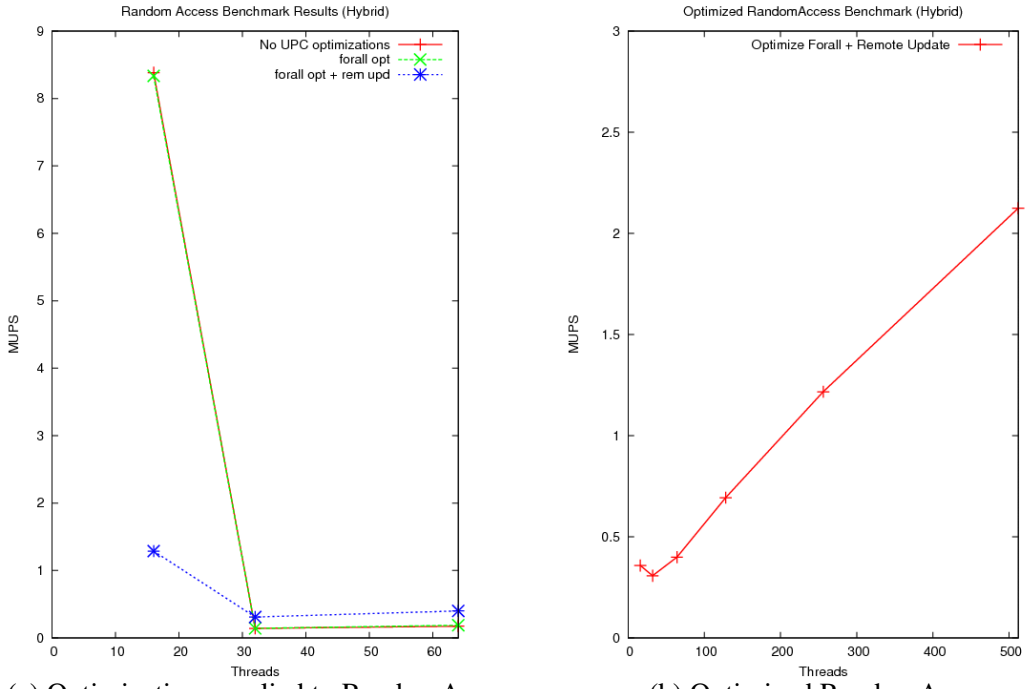
$$\frac{2 \times 716.6}{75} = 19.1 \text{ GUPS}$$

As Table 8.3 shows, the actual measured benchmark performance is very close to this theoretical peak. Furthermore, the measured performance of 16.72 GUPS on 131072 pro-

cessors is approximately 50% of the performance of the best known implementation written for the same machine (35.47 GUPS) [37].

8.3.3 Hybrid Environment

As with the distributed environment, the compiler is not able to determine the owner of the shared references when RandomAccess is compiled for a hybrid environment. As a result, no shared references are candidates for SOAP, SOAC, or SOAS.



(a) Optimizations applied to RandomAccess (4-nodes)

(b) Optimized RandomAccess (32-nodes)

Figure 8.18: RandomAccess results on hybrid v20

Figure 8.18(a) shows the performance, measured in MUPS, for the forall optimization and for the remote update optimization using the 4 shared nodes of the v20 cluster. The most striking observation here is the significant reduction in performance on 16 threads when the remote update optimization is applied (85%). This is because the implementation of remote update for a local shared-object access uses locks to ensure exclusive access during the update operation. For a benchmark such as RandomAccess with a high number of remote updates, these locks significantly impact performance.

As with the distributed environment, the remote update optimization results in approximately a 2X performance improvement when run on 2 and 4 nodes (32 and 64 threads).

Again, this is due to the reduction in messages that results from the remote update optimization.

Figure 8.18(b) shows the MUPS measurement for the optimized RandomAccess benchmark run on 32-nodes of the v20 cluster. The drop in performance between 16 and 32 threads (1 node to 2 nodes) is due to the communication necessary to access remote shared-objects. This difference is made up with 4 nodes (64 threads). As with the distributed environment, the results continue to scale up to 512 threads.

There is a significant difference in the absolute performance between RandomAccess run in a distributed environment and a hybrid environment (Figures 8.17(b) and 8.18(b)). In a distributed environment with 32 threads the maximum performance of RandomAccess is 5.46 MUPS while in the hybrid environment with 512 threads the maximum performance is 2.12 MUPS. This difference is attributed to the overhead of the locks used to perform the remote update on a local shared-object.

8.3.4 Discussion

The remote update optimization turned out to be an effective way to reduce the number of messages required in the Random Access benchmark. However, since locks are used in the implementation of a *local* remote update, the overhead when performing a remote update on a local shared object can be quite high, as seen in the Random Access benchmark in a hybrid environment. The RTS has to respect the remote update calls, even if the shared object maps to the same node as the accessing thread. In other words, the RTS cannot simply choose to change a remote update call into the corresponding *deref*, *update*, *assign* sequence that would normally be generated. The compiler currently is not able to account for this additional overhead and to be more conservative when generating remote update calls for shared objects that *may* be local.

8.4 UPCFish

The UPCFish benchmark, developed at Michigan Tech, is a time step based predator/prey simulation using fish and sharks in an ocean [54]. At each timestep fish can move around the ocean. Sharks also move around, chasing and eating the fish. At each timestep fish and sharks are born or die, based on their age and preset birth and death rates (*i.e.* after a fish reaches a certain age it will die with probability `FishDeathRate`). The simulation terminates when no more fish or sharks exist or after a specified number of time steps.

Fish are represented by a data structure that contains their position, velocity, and age. The data structure representing sharks also includes their hunger level. All fish are stored together in a doubly-linked list which is iterated over in each timestep of the simulation. Similarly, all sharks are stored together in a separate double-linked list. Each UPC thread contains a private list of fish and sharks. In each timestep, each thread updates their list of fish and sharks based on births, deaths and movement.

The ocean is represented as a 2D shared array of structures, blocked by row. Each structure represents a position in the ocean and contains information including: the position in the ocean (X and Y coordinates), the number of fish and sharks at that position, the fish smell and shark smell at that position and the forces acting on the fish and sharks.

8.4.1 Shared-Memory Environment

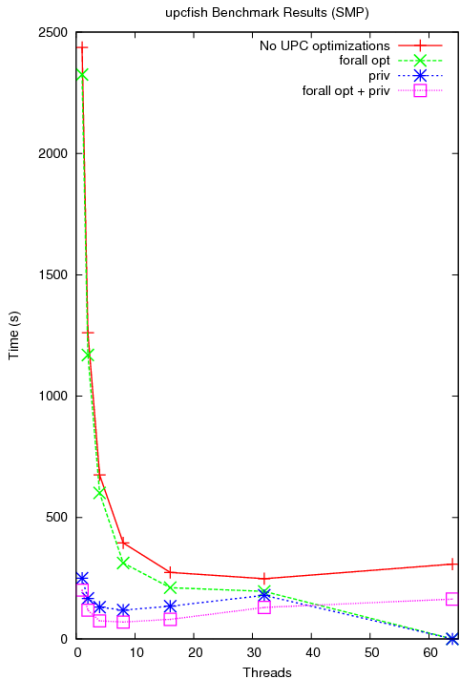


Figure 8.19: UPCFish results in a shared-memory environment

Figure 8.19 shows the results of running the UPCFish benchmark in a shared-memory environment. The graph reports time (in seconds) as a function of the number of threads (lower is better). The execution times show a relatively small improvement from the forall optimization but a fairly substantial improvement from the SOAP optimization. As with other benchmarks, combining both the forall optimization and the SOAP optimization re-

sults in the best performance.

The execution times in Figure 8.19 also demonstrate that the UPCFish benchmark does not scale well. This observation is most obvious for the optimized version, where the best performance is obtained with 8 threads (70s); adding more than 8 threads actually decreases the performance (80s, 130s and 164s for 16, 32 and 64 threads respectively). This behaviour is also seen in the non-optimized case. The failure for this benchmark to scale is due to a load-balancing problem caused by the distribution of fish and sharks among the threads [54]. Each thread contains a private list of fish and sharks, which the thread updates on each time step. When updating the fish and sharks in each timestep, if the lists become large, the amount of work the thread has to do during the update step increases; similarly, if the list becomes empty (all the fish and sharks die), the thread has no work to do during the update step. Thus, in order for this benchmark to scale as more threads are added, the lists of fish and sharks must be redistributed among all of the processors. Suenaga and Merkey discuss the consequences of making these lists shared and the effect it will have on the affinity of the fish and sharks data structure [54]. If the fish and sharks are made shared, then the natural way to distribute the work is using the affinity of the ocean grid where the fish/shark is located. For example, at timestep t if a fish f is located at position $OceanGrid[x, y]$ then the thread that owns $OceanGrid[x, y]$ should update f (decide whether the fish dies, gives birth, where it moves, *etc*). However, this approach could lead to a high number of remote accesses since the fish and sharks move around at random.

8.4.2 Distributed Environment

Figure 8.20 shows the execution times for the UPCFish benchmark run in a distributed environment. In this environment the compiler is not able to determine the locality of the shared-objects and thus the SOAP optimization provides little benefit. Similarly, the forall optimization also provides little benefit because of the communication that is required. On the other hand, the remote update is able to optimize several remote operations and thus leads to a substantial performance improvement.

8.4.3 Hybrid Environment

Figure 8.21 shows the execution times for the UPCFish benchmark run in a hybrid environment. As with the distributed environment, the SOAP optimization provided only a small benefit because the compiler is not able to determine the locality for most of the shared references. However, in the hybrid environment the Remote Update optimization also pro-

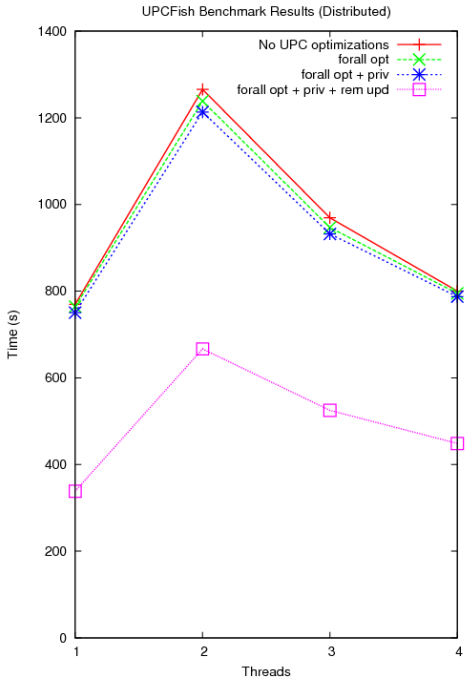


Figure 8.20: UPCFish results on distributed v20

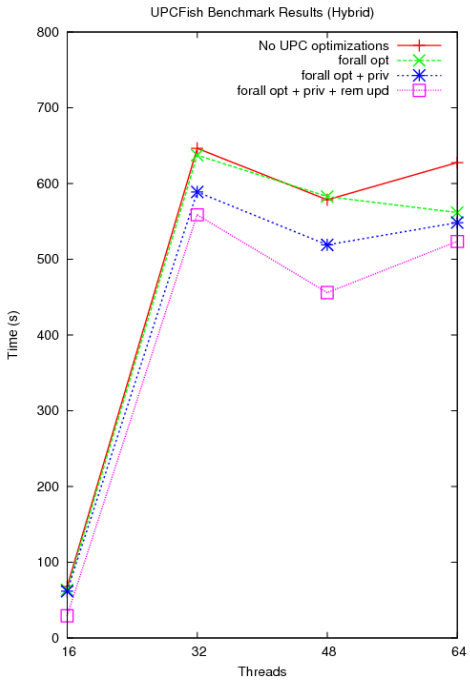


Figure 8.21: UPCFish results on hybrid v20

vides only a small benefit above the SOAP optimization. As seen in the Random Access benchmark, this is due to the use of locks in the Remote Update optimization to guarantee atomicity when performing an update on a shared object located on the same node as the accessing thread.

8.4.4 Discussion

The first UPC benchmark we have seen that has problems with load balancing!

Again, we see that the use of locks in the runtime when performing RemoteUpdates on shared-objects located on the same node as the accessing thread affects the performance.

Due to the relatively high running times of UPCFish, and the load-balancing issues as the number of threads is increased, it was not run on the full v20 cluster in either the distributed or the hybrid environment.

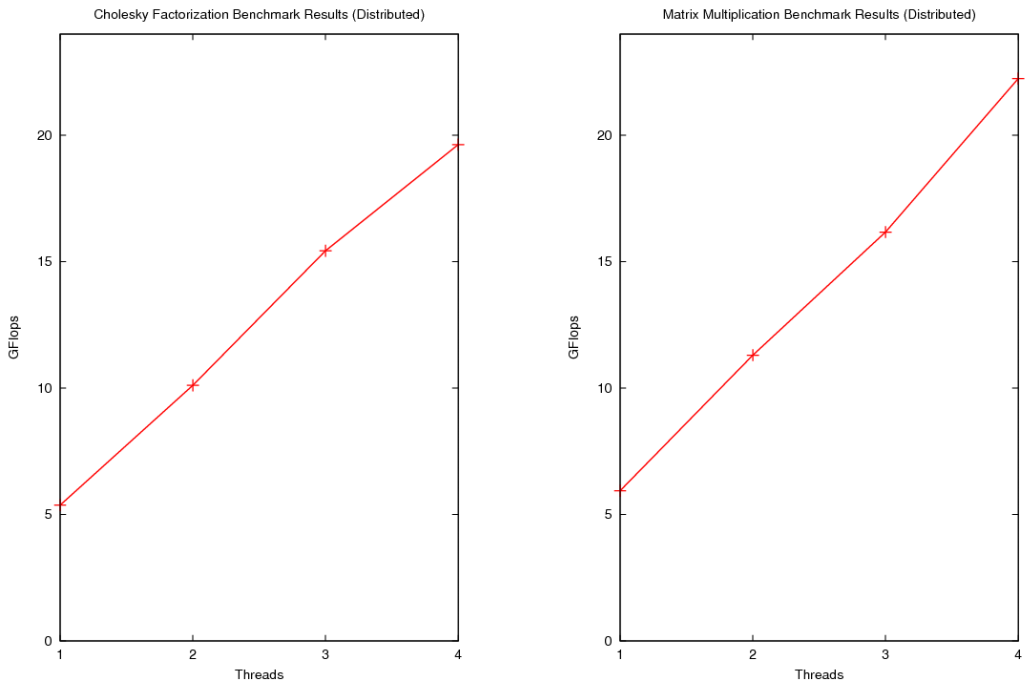
8.5 Cholesky Factorization and Matrix Multiplication

The Cholesky factorization and matrix multiplication benchmarks were written to showcase multiblocked arrays [9]. The shared arrays are distributed using multiblocks so they can be directly used with the Engineering Scientific Subroutine Library (ESSL) library [29]. The code is patterned after the Linear Algebra PACKage (LAPACK) `dpotrf` implementation [26]. To illustrate the compactness of the code, one of the two subroutines from Cholesky factorization, distributed symmetric rank-k update, is presented in Figure 8.22.

```
void update_mb (shared double [B][B] A[N][N], int col0, int col1) {
  double a_local[B*B], b_local[B*B];
  upc_forall (int ii=col1; ii<N; ii+=B; continue)
    upc_forall (int jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {
      upc_memget (a_local, &A[ii][col0], sizeof(double)*B*B);
      upc_memget (b_local, &A[jj][col0], sizeof(double)*B*B);
      dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B, a_local,
             &B, &beta, (void *)&A[ii][jj], &B);
    }
}
```

Figure 8.22: Cholesky distributed symmetric rank-k update kernel

The matrix multiplication benchmark is written in a similar fashion. There is a (serial) `k` loop around the `update` function in Figure 8.22 with slightly different loop bounds and three shared array arguments: `A`, `B`, `C`, instead of only one.



(a) Cholesky factorization (b) Matrix multiplication

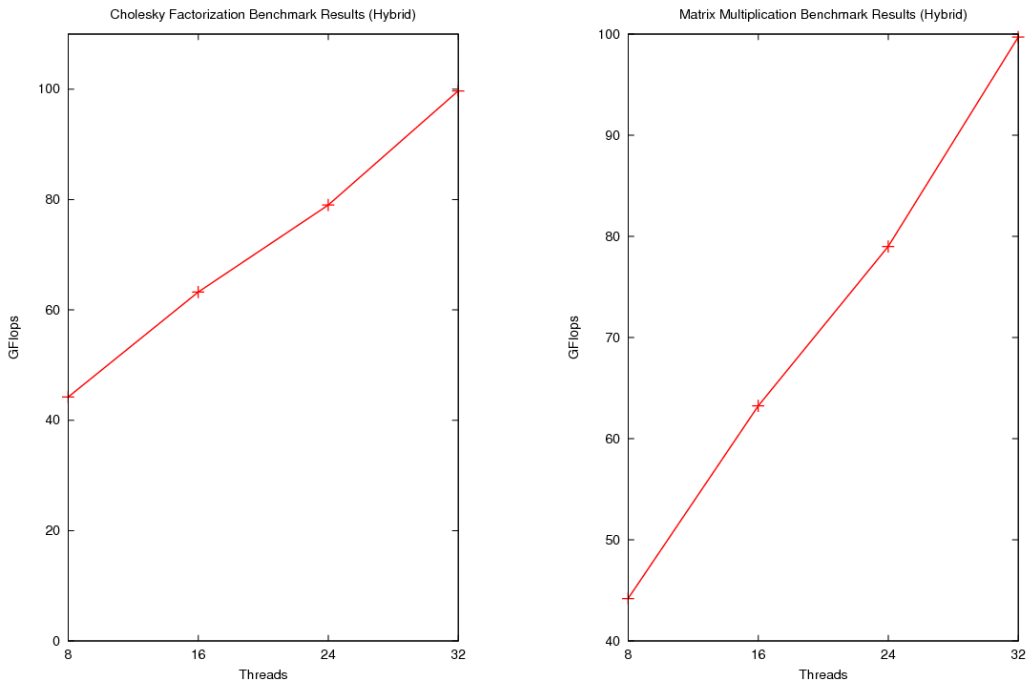
Figure 8.23: Cholesky factorization and matrix multiplication on distributed c132, from [9]

8.5.1 Distributed Environment

The results for the optimized Cholesky factorization and for the matrix multiplication benchmarks, run in a distributed environment, are presented in Figure 8.23. The graph shows the number of Giga Floating-Point Operations Per Second (GFLOPS) as a function of the number of threads (higher is better). Since the benchmarks were written to showcase multi-blocked arrays in UPC and how they can be used to interface with existing high performance libraries, the benchmarks contain explicit calls to the dgemm library function to perform the multiplication of the matrices. As a result there are no opportunities for compiler optimizations. Nevertheless, the benchmarks exhibit significant performance improvement when increasing from 1 to 4 UPC threads.

8.5.2 Hybrid Environment

The GFLOPS for the Cholesky factorization and matrix multiplication benchmarks run on c132 in a hybrid environment are presented in Figure 8.24. Even though there are no opportunities for the compiler to optimize shared-object accesses, the performance scales adequately when the number of UPC threads is increased from 8 (1 node) to 32 (4 nodes) threads.



(a) Cholesky factorization (b) Matrix multiplication

Figure 8.24: Cholesky factorization and matrix multiplication on hybrid c132, from [9]

8.5.3 Discussion

The purpose of the Cholesky factorization and matrix multiplication benchmarks was to illustrate the benefits of multiblocked shared arrays in UPC. The benchmarks illustrate the ability to interface a multiblocked shared array with existing highly tuned library functions (in this case dgemm) in order to obtain good performance and scaling.

8.6 Summary

The experimental evaluation in this chapter studies the benefit of the locality optimizations and forall optimizations on several benchmarks. The results of this evaluation demonstrate that the optimizations are able to handle all of the benchmarks in a shared-memory environment quite well. In a shared-memory environment, the compiler is able to determine the locality of every shared reference and the SOAP optimization is able to privatize all shared references. The results are comparable to OpenMP (STREAM) and most benchmarks scale nicely, with the exception of UPCFish which has a load balancing problem caused by the implementation. Integer affinity tests have a lower overhead than pointer-to-shared affinity tests and thus result in better performance, as seen with the STREAM benchmark.

In the distributed and hybrid environments, the SOAP optimization is also important when the compiler can prove locality (*e.g.,* STREAM), but has no benefit when it cannot (*e.g.,* Random Access). In benchmarks that contain a mixture of local and remote accesses, the SOAP optimization can have a relatively small impact (*e.g.,* Sobel). On the other hand, the SOAC optimization can substantially improve performance when opportunities exist, as seen with the Sobel benchmark. There were no situations where SOAS provided any benefit. This is because the loops (as currently written) do not contain enough work to hide the cost of the communication. This situation may also be remedied by a combination of unrolling (to increase loop body size and therefore the amount of work performed in each iteration) as well as combining SOAC and SOAS algorithms so that the compiler can generate non-blocking coalesced calls. Finally, the remote update optimization was shown to have a significant impact on two benchmarks, Random Access and UPCFish, when run in a distributed environment. These benefits, however, were diminished in the hybrid environment because of the locks used in the RTS to ensure that the updates are atomic.

Results from the STREAM benchmark demonstrate that even through the optimizations described here may dramatically improve performance, existing compiler optimizations (*e.g.,* strength reduction) also play a vital role in achieving performance competitive with alternate implementations.

All of the work presented here was implemented in IBM's XL UPC compiler, based on the XL family of commercial compilers. As such, all optimizations and transformations presented here are subject to rigourous testing using several test suites. Thus, the implementations are not only tested on the benchmarks presented here, but also on a wide-range of benchmarks.

Chapter 9

Related Work

There has been extensive research in the area of parallel programming languages. Work related to each of the objectives presented in Section 1.1 is detailed in Sections 9.1, 9.3, and 9.2 below. Section 9.4 presents a discussion of other parallel programming environments.

9.1 Shared Data Locality Analysis

The new shared-data locality analysis presented in Chapter 5 enables the compiler to distinguish between local and remote shared-data accesses. Previous work on data locality analysis used a combination of compile-time and run-time analysis.

Zhu and Hendren present a technique to distinguish between local and remote shared-data accesses on dynamically allocated data structures in parallel C programs [62]. Their algorithm predicts when an indirect reference via a pointer can be safely assumed to be local using information about the context of function calls and memory allocation statements. Their analysis was run on 5 parallel benchmarks from the Olden benchmark suite [52] that use dynamic data structures. Their technique resulted in up to a 99.99% reduction in remote accesses and up to 93% speedup on 16 processors. They use type inferences and intra- and inter-procedural analysis to identify shared-data accesses that are local. This type of analysis is not required in UPC because the language allows the programmer to specify the distribution of shared data. Thus, the compiler can determine if an access is local or remote based on the user-specified data distribution. This distribution is not present in the environment used by Zhu and Hendren and thus the compiler had to infer locality based on the allocation and assignment of shared pointers.

Koelbel and Mehrota present Kali, a programming environment providing a software layer that supports a global address space on distributed-memory machines [42]. The programmer identifies shared data and specifies how the shared data is distributed across the

processors using language-specific notations. Shared data is used to perform computations in parallel loops. The compiler analyzes the parallel loops and creates two communication sets: *send_set* and *recv_set*, and two iteration sets: *local_iter* and *nonlocal_iter* for each processor. The compiler determines what communication is necessary by performing set intersections using these sets. When the compiler cannot determine the communication sets, the sets need to be computed at runtime. The compiler creates an *inspector* loop that runs before the parallel loop and adds all non-local shared-data accesses to the communication set. Once the inspector loop is complete, the *executor* loop is run. The executor loop performs the actual execution of the parallel loop using the sets computed from the inspector. Their results show that when the compiler is able to determine the communication sets they can achieve performance close to what is achieved through hand-written code. When the compiler cannot compute the communication sets, the inspector loop can introduce very large overheads (up to 100%) for some programs.

Koelbel and Mehrota's approach differs from our approach because the Kali compiler has the ability to identify non-local shared-data accesses and to insert the actual communication calls to the owner of the shared data. In our approach, the RTS calls are responsible for determining the owner of the shared data and for performing the communication; the compiler only distinguishes between local and remote data. Thus, the additional overhead required for remote shared-data accesses is performed in the RTS and is not exposed to the compiler. However, Koelbel and Mehrota's approach does have the benefit of computing the communication sets once for each parallel loop. In our approach, shared-object access coalescing attempts to reduce the number of remote shared-object accesses in every iteration of the loop. If this is unsuccessful, every remote shared-object access will incur the overhead of the RTS in every iteration of the loop. This cost could be reduced by performing loop unrolling, thereby decreasing the number of loop iterations and increasing the number of remote shared-object accesses available for coalescing in each iteration.

The TreadMarks parallel programming environment provides a shared-address-space programming model but is designed to run on distributed-memory architectures [5]. TreadMarks contains a runtime system that detects accesses to shared data and manages communication between processors. Shared data is created using a special memory allocation function `Tmk_malloc` and replicated on all processors. The consistency of shared data is managed at the page-level. By using a Lazy-Release-Consistency model, TreadMarks is able to delay consistency-related communication until the end of a synchronization region. When multiple processors change different shared data that are located in the same page, a

multiple-writer protocol is used to merge all of the changes. If two processors change the same shared data in a page, a data race has occurred and it is reported to the user.

In TreadMarks, shared arrays are replicated on every node in the system. Thus, TreadMarks makes no distinction between local and remote shared data. On the other hand, in UPC shared arrays are distributed among all of the nodes in the system. This distribution of shared objects enables the distinction between local and remote shared data, where accessing local shared data is generally faster then accessing remote shared data because no communication is necessary. The distribution of shared objects allows UPC programs to use very large arrays because the array size is based on the aggregated memory of all nodes and is not limited to the address space of a single node. However, because shared data in UPC is distributed among all processors, a management system must be used to determine the location of each shared object. In our implementation, this management is handled by the SVD. While this management system allows UPC programs to scale to a large number of threads, it also incurs significant performance overhead.

Dwarkadas *et al.* modified the TreadMarks compiler and runtime system to allow the compiler to communicate information about shared-data accesses to the runtime system [27]. Their compiler uses regular section analysis to obtain information about array accesses in loop nests. This information is then passed to the runtime system. When the analysis is successful, runtime detection of accesses to shared data is not necessary. Instead, the runtime system can prepare for the shared-data accesses ahead of time. With this information, the runtime system is able to push shared data to the accessing processor, making it available when needed.

Co-Array Fortran (CAF) and Titanium are two Partitioned Global Address Space languages that can benefit from shared-data locality analysis. Both CAF and Titanium require the programmer to identify local accesses. This requirement increases performance because accessing local shared objects is faster than accessing remote shared objects. However, it also decreases programmer's productivity because the identification of local shared data is the responsibility of the programmer. Using the analysis presented here the compiler can automatically detect local accesses, eliminating the need for programmer annotation.

9.2 Coalescing Shared-Object Accesses

The coalescing of several remote shared-object accesses into a single access is designed to reduce the number of messages exchanged between processors in a distributed-memory ar-

chitecture. The advantage is that the start-up cost associated with sending messages and the network latency associated with messages is only incurred once for several data exchanges. This technique is also known as *communication coalescing*, *message coalescing* or *bulk communication*.

Coarfa *et al.* demonstrated that both UPC and CAF versions of the NAS Parallel Benchmarks obtain scalable performance on a cluster platform when using bulk communication [23]. Currently, it is the responsibility of the programmer to implement the bulk communication explicitly in the program. In CAF the programmer can access remote co-array data using Fortran 90 array triplet notation to implement bulk communication. In UPC, the programmer must explicitly use bulk communication routines such as `upc_memput`. This work has demonstrated several performance improvements that previously required the programmer to explicitly use bulk communication: the compiler detects these opportunities and replaces several remote shared-object accesses with a single coalesced access, resulting in a performance improvement.

By analyzing shared-data accesses, Dwarkadas *et al.* were able to pass information about shared-data accesses to the TreadMarks runtime system [27]. This information allowed the runtime to coalesce several remote data fetches into a single message. Communication coalescing resulted in speedups of between 2X and almost 8X for several benchmarks run on an 8-processor system. These results also indicate that coalescing shared-object accesses result in significant performance improvements. Currently there is no mechanism in the RTS to coalesce shared data accesses – all coalescing must be performed by the compiler. Thus, the techniques used by Dwarkadas *et al.* are not directly applicable to our system. Since there is no concept of local or remote shared data in TreadMarks, their technique would also coalesce local shared accesses, which we do not do.

Following split-phase communication analysis, the Berkeley UPC compiler performs message coalescing to combine small `GET` and `PUT` operations into larger messages [21]. Consider two shared objects O_{s_1} and O_{s_2} and their corresponding addresses $addr_1$ and $addr_2$. If the communication points to retrieve O_{s_1} and O_{s_2} are adjacent in the statement list, then O_{s_1} and O_{s_2} are considered for message coalescing. Their message coalescing uses a *bounding box* method where all data from $addr_1$ to $addr_2$ (inclusive) is contained in the message. As a result, before performing message coalescing a profitability analysis is performed. If $addr1$ and $addr2$ are adjacent then the two messages are combined into a single message. If $addr1$ and $addr2$ are not adjacent, the distance between them is computed and, if the distance is less than a predetermined threshold, message coalescing is performed.

When the accesses to O_{s_1} and O_{s_2} are writes, $addr1$ and $addr2$ must be adjacent to prevent data corruption. Chen *et al.* did not present any results specific to message coalescing to indicate the number of messages that were coalesced or the benefit of performing only message coalescing. However, when combined with redundancy elimination for shared-pointer arithmetic and split-phase communication, only one benchmark (Barnes) benefited from message coalescing. This result can be attributed to message coalescing failing to coalesce messages in the other benchmarks and does not necessarily indicate that successful message coalescing has no benefit.

If the compiler is unable to determine the locations of the shared objects, the decision of whether to coalesce is delayed until runtime. A new function has been added to their runtime system which allows the compiler to specify an array of the source and destination addresses of the `GET` operations. The runtime uses the same profitability analysis as the compiler to determine whether messages should be coalesced. If messages are coalesced, the runtime is responsible for temporary-buffer management and for inserting the corresponding `SYNC` calls to ensure that the communication completes before the data is used.

The message-coalescing technique described by Chen *et al.* has two limitations that our shared-object access coalescing optimizations address. First, in order to be considered for coalescing, the communication points must be adjacent in the statement list. In our technique, the only restrictions are that the accesses must occur in the same loop (to preserve the original program semantics) and there cannot be conflicting accesses (*i.e.* a use and a definition of the same shared object) within the loop. Of these two restrictions, the latter can be relaxed in the future if we find evidence that doing so will be beneficial. The second limitation is the use of a bounding box method for collecting the shared data. This limitation has two implications: (i) the success of message coalescing is partially determined by the shared-object layout in memory; and (ii) definitions of non-contiguous shared-data items cannot be coalesced because data corruption may occur. Our approach does not suffer from these limitations because the information provided in the request allows the RTS to determine the specific shared data items and only the specific shared data items are included in the messages.

Later Chen *et al.* expanded the communication coalescing in the Berkeley compiler to use strided functions provided by the GASNet communication layer and to allow the coalescing of non-contiguous reads and writes [20, 39]. In this implementation of coalescing a translator inserts special markers in the code to signal the begin and the end of a coalescing

region. The compiler also generates a representation of the structure of communication operations based on descriptors proposed by Paek *et al.* [49]. Accesses within this region are managed by the runtime system through queues. The runtime is also responsible for checking for conflicts. This approach has significant runtime overhead when compared with our compile-time only implementation of strided coalescing.

Unlike the Berkeley UPC runtime system, our RTS does not have the ability to decide if messages should be coalesced. Communication is performed for each remote shared-object accesses. Thus, situations when the compiler is unable to identify remote shared-object accesses may still be candidates for message coalescing using the Berkeley UPC runtime system, but will not be candidates for remote shared-object access coalescing using our current RTS.

Coalescing in the Rice dHPF compiler is done after communication vectorization and is restricted to overlapping vectors. Chavarría-Miranda and Mellor-Crummey report up to 55% reduction in communication volume for NAS benchmarks after this vectorized coalescing [18]. Kandemir *et al.* uses a combination of data-flow analysis and linear algebra to create a communication optimization technique that supports both message vectorization and message coalescing [41].

9.3 Shared-Object Access Scheduling

The basic principle behind remote shared-object access scheduling is to overlap the communication required to move remote shared data with unrelated computation. This technique is also known as *split-phase transactions* or *split-phase communication*.

Zhu and Hendren present a technique to overlap communication and computation in parallel C programs [61]. Their technique uses a *possible-placement* analysis to find all possible points to place communication. This analysis collects sets of remote-communication expressions and separates the expressions into remote read operations and remote write operations. Remote read operations are propagated upwards while remote write operations are propagated downwards. Their *communication selection* transformation selects where the communication is inserted and determines if blocking or pipelining of communication should be performed. Their work was implemented in the EARTH-McCAT optimizing/parallelizing C compiler. Their results demonstrated that the communication optimization can result in performance improvements of up to 16%.

Chen *et al.* have implemented split-phase communication in the Berkeley UPC com-

piler to overlap the communication of shared accesses with other computation in UPC programs [21]. The split-phase communication is handled differently for definitions and uses of shared variables. For a given shared variable v_s, every use of v_s is identified. A *communication point*, representing the necessary `GET` operation, is inserted in every basic block that contains a use of v_s. If the basic block contains a definition of v_s, the communication point is placed immediately after the definition, otherwise the communication point is placed at the beginning of the basic block. To reduce the number of communication points to get a given shared variable cp_{v_s}, a breadth-first postorder traversal of the control flow graph is used to propagate communication points upwards. If all of the successors of a basic block b contain cp_{v_s} (*i.e.* all successors contain a use of v_s), the communication points are merged into a single communication point and moved into b. Once the breadth-first search is complete, the communication points represent the locations where a `GET` operation is necessary. The corresponding sync operations are inserted immediately before every use of v_s.

For definitions of shared variables, Chen *et al.*'s analysis attempts to minimize the number of syncs inserted. A `PUT` call is placed after every definition of a shared variable. A path-sensitive analysis then examines all paths from the `PUT` call to the function exit. If a statement that may reference or modify the shared location is discovered, a sync call is placed before the statement and the analysis terminates for the current path. Thus, the algorithm will attempt to push the corresponding sync instructions as far away from the `PUT` instruction as possible. If a loop is encountered during the path traversal, the analysis terminates to avoid placing the `SYNC` instruction inside the loop. If the definition is located inside a loop body, the corresponding `SYNC` is placed at the end of the loop body (*i.e.* the path traversal does not traverse a loop back edge). This placement ensures that the `SYNC` operation will be executed after its corresponding put operation.

Chen et al.'s implementation of split-phase communication has two limitations that are addressed in this work. First, Chen *et al.*'s analysis does not distinguish between local shared accesses and remote shared accesses. As a result, all shared accesses are implemented as split-phase transactions. In programs that contain a lot of shared data, scheduling all accesses using split-phase communication can create many unnecessary communication calls and `SYNC` operations. Not only are these communication calls and `SYNC` operations not necessary for local data, their presence could negatively affect the performance of other optimizations. Our work, however, is able to distinguish between local shared accesses and remote shared accesses and uses split-phase transactions only for accesses to remote shared objects.

The second limitation in Chen *et al.*'s approach is when the definition of a shared variable is found inside a loop body, the `SYNC` instruction is placed at the end of the loop body. Consider the situation where each iteration of a loop performs many computations and at the end of the iteration writes the result to a remote shared variable. In this situation, their analysis will fail to overlap the computation in the loop with the communication at the end of the loop. A more general approach would pipeline the two phases of the split-phase transaction. Pipelining can be achieved by peeling the first iteration of the loop upwards to create a pre-computation block. In the pre-computation block, the remote shared-object access is scheduled immediately after the definition of the shared data. A `SYNC` instruction is then placed inside the loop body before the write to the shared object. This `SYNC` instruction will ensure that the write from the pre-computation block (or the write from the previous iteration) has completed. A final `SYNC` call would be placed after the loop to ensure that the write on the final iteration completes.

Iancu *et al.* implemented *HUNT*, a runtime system to exploit some of the available overlap between communication and computation in UPC programs. *HUNT* ignores the explicit calls to retire communication (*i.e.* the wait calls) and uses virtual-memory support to identify the point at which the shared data is actually used. This way, the communication is retired at the point in the program where the shared data is used. Their findings demonstrate that *HUNT* is able to find and exploit overlap in the FP, IS, and MG NASA Advanced Supercomputing (NAS) Parallel Benchmarks. This approach compliments the work proposed here since it only applies to UPC programs at runtime. Cases where our transformations fail to identify opportunities for remote shared-object access scheduling such as indirect accesses (*e.g.,* `a[b[i]]`, where a is a shared array) may be detected and successfully optimized with this system. However, opportunities that can be identified by the compiler can be transformed without incurring the additional overhead of the runtime support.

Koelbel and Mehrotra also overlap communication with computation to improve performance [42]. Once the inspector loop has finished executing, the communication sets are known. At the beginning of the executor loop, the communication to retrieve the remote shared data is initiated. While waiting for the data to arrive, any local computations (that do not rely on the shared data) are performed. Unfortunately, Koelbel and Mehrotra were unable to measure the overlap between communication and computation due to the program timing methodology they used [42].

9.4 Alternative Parallel-Programming Environments

There are many parallel programming paradigms that are currently available for programmers. These can be separated into two basic categories: (i) parallel programming environments (compiler and runtime system) and (ii) parallel extensions to existing languages.

Cilk is a task-parallel programming environment where the programmer specifies procedures that are run in parallel. Shared data in Cilk is not explicitly marked by the programmer. Instead, data is shared using global variables and references (passing pointers). OpenMP is similar to Cilk in that the program is run sequentially but *regions* are executed in parallel [48]. The programmer marks parallel regions using pragmas recognized by compilers that support OpenMP. OpenMP will work with C, C++ and Fortran programs on shared memory systems. Like Cilk, data is shared in OpenMP using global variables and references and the programmer is required to introduce synchronization to prevent data races and deadlock situations.

High Performance Fortran (HPF) is a version of Fortran that has been extended with data decomposition specifications. The programmer is responsible for specifying the data distribution for each array and the compiler must: (i) determine how to distribute the loop iterations across processors; (ii) ensure that remote data is available on the executing processor.

The Fortran D compiler project from Rice University is an example of a Fortran compiler that supports HPF features [4]. The compiler uses the *owner computes* rule to identify a *partitioning reference* such that the loop iterations will be executed by the processor that owns the data item on the left-hand side of the reference. For loops that contain accesses to multiple arrays, heuristics are used to determine the best partitioning reference. The iteration space of loops is distributed by converting the global iteration space of the entire loop into *local iteration sets* that are run on each processor [34, 34]. Once the partitioning reference has been identified, communication analysis is used to build a *communication set* by determining which references cause remote accesses. The communication analysis examines all *right-hand-side* references and for each reference the compiler constructs the index set accessed by each processor by mapping the global index into the local index set. The difference between the local index set and the index set owned by a given processor represents the remote accesses [32, 33].

Chatterjee *et al.* provide an alternative method of computing local index sets and communication sets using a finite state machine to traverse the local index space of each proces-

sor [17]. This approach is limited to arrays that are distributed using the same block-cyclic distribution and accesses that use the same stride as the partitioning reference.

Gupta *et al.* present techniques for computing local index sets and communication sets for arrays with different distributions and strides than the partitioning reference [31]. The communication sets for a given loop are based on closed form regular sections of the arrays. The closed form regular sections are also used to generate indexing code for the distributed arrays. This approach allows the compiler to analyze arrays with different distributions and determine the local and remote accesses and thus is more general than the approaches presented here which require the same blocking factor (distribution) to be used in the affinity test and the shared-object accesses in the **upc_forall** loop. However, since the indexing is computed at runtime, this approach adds a significant amount of overhead to accessing the local portions of the array.

The Message Passing Interface is a library that provides functions to efficiently move data between processors in a distributed-memory environment. Shared data is explicitly managed by the programmer — including allocation and data movement between processors. Since the programmer is required to explicitly manage the shared data, well-written MPI programs exhibit very good performance across a wide range of applications and distributed-memory architectures and are frequently used as a comparison when testing alternative parallel-programming environments. However, requiring the programmer to explicitly manage shared data decreases programmer productivity. One of the goals of this research is to use the compiler to manage shared-data movement between processors and obtain performance comparable to MPI programs.

Parallel languages that follow the Multiple-Program, Multiple-Data programming model include Chapel, Fortress and X10 [14, 3, 16]. These programs allow each thread to execute a different sequence of instructions. This task-parallel programming model is much more complicated to work with and to optimize, however it also allows much more flexibility in the types of parallel programs that can be implemented. In general, any MPMD programming language that employs a PGAS memory model, where the programmer indicates how the data is distributed, should be able to use the shared-data locality analysis presented here. The techniques to reduce the overhead of accessing local and remote shared-memory may also be applicable, depending on the underlying implementation.

X10 is an object-oriented programming language designed for high-performance and high-productivity programming. X10 is a member of the PGAS family of languages and employs the Multiple Instruction Multiple Data (MIMD) programming paradigm using

lightweight *activities* to execute instructions. X10 introduces the concept of *places* that represent a collection of mutable data objects and *activities* that run in a place [16]. Arrays are represented using a multidimensional set of index points called a region. Since it is a member of the PGAS family of languages, the programmer can create regions using a specified *distribution*. This distribution distributes elements of the array across all places, similar (in concept) to the way shared objects are created in UPC (scalars are owned by one place while array elements can be distributed across multiple places). To manipulate a part of a region located in a place p_1, the programmer creates an activity in place p_1 and specifies the instructions to be executed. If the activity in place p_1 requires data located in another place, p_2, it will create an activity at p_2 to manipulate the remote data (either read or write). X10 also contains the work-sharing constructs *foreach* and *ateach*. The foreach construct performs a parallel iteration over all points in a region. The *ateach* construct performs a parallel iteration over all points in a data distribution. The ateach construct is similar to the **upc_forall** loop using a pointer-to-shared affinity test to indicate that the owner of the shared data will execute the loop iteration.

In principle, an X10 compiler should be able to use techniques similar to the locality analysis presented here to determine whether each activity is going to access local or remote shared data. For local shared data, the activity can directly access the data. In a distributed-memory environment, for remote shared data, similar optimizations can be applied to coalesce activities on the same place (similar to SOAC) and to schedule activities that access remote date in order to overlap the necessary communication with other computation. In practice, some of this will prove to be more challenging because X10 has a very robust mechanism to distribute data among places compared with the relatively simple method of blocking data available in UPC.

Chapter 10

Conclusions

One approach to create compiler optimizations is to identify *hotspots* — places in the code where the majority of the execution time is spent — and to develop code transformations to reduce this time. Section 4.3.5 shows that in UPC programs, calls to the RTS to access shared objects have an extremely high overhead compared with traditional memory accesses (*i.e.* a C-pointer access). Thus, one focus of this work is to reduce this time as much as possible.

In order to optimize shared-object accesses, the compiler must first determine the locality of the shared object being accessed. Chapter 5 presents a compiler analysis that determines the locality of shared references located in a **upc_forall** loop based on the affinity test used by the **upc_forall** loop. Using this analysis, the compiler builds a shared reference map that tracks the relative thread ID of the owner of each shared reference.

Four optimizations that use the locality information in the shared reference map are presented in Chapter 6. These optimizations can be broken into two categories based on the locality of the accesses: (i) accessing local shared objects and (ii) accessing remote shared objects. Accesses to local shared objects are optimized using the PRIVATIZESOA algorithm presented in Chapter 6.1. This optimization works by transforming accesses to local shared-objects into traditional memory accesses by converting the UPC *global index* used to access the shared-object into a thread-specific *local offset* used to access the underlying memory.

Accesses to remote shared-objects are optimized using the COALESCESOA, SCHEDULESOA and and Remote Update algorithms. The COALESCESOA algorithm coalesces multiple accesses to the same shared-object into a single access, thereby reducing the number of messages required to manipulate (read or write) the remote shared data. The SCHEDULESOA algorithm overlaps the communication necessary to access a remote shared-object with other computation. The Remote Update optimization sends a message to the thread

that owns the shared object to be updated, specifying the type of operation to perform and the data to use. This removes the intermediate communication originally required because the update is performed on the remote thread.

A second goal of this work was to reduce the cost of executing parallel loops in UPC. Chapter 7 demonstrates the overhead of the *integer* and *pointer-to-shared* affinity tests that are used in **upc_forall** loops. It also describes optimizations that can reduce the overhead of the two different affinity tests.

We identified three different environments of interest when running UPC programs: *shared-memory*, *distributed*, and *hybrid*. In a shared-memory environment, all threads share the same global memory space. In a distributed environment, each thread has its own address space and the only way for one thread to access the address space of another thread is to use explicit communication. Finally, the hybrid environment is a combination of shared-memory and distributed environments where multiple threads share the same address space. Thus, a thread can access the address space of some other threads, but not of all threads.

The locality optimizations presented in Chapter 6 have different impacts in the different environments tested. For example, the SOAP optimization proves to be extremely beneficial to all UPC benchmarks in a shared-memory environment because the compiler is able to convert all shared-object accesses into direct memory accesses. In the distributed and hybrid environments, the benefits of SOAP varies from extremely high in cases where the locality analysis is able to identify most or all shared-object accesses as local (*e.g.,* STREAM), to none at all when the locality analysis is not able to classify the shared references (*e.g.,* Random Access). For benchmarks that have a mixture of local and remote shared-object accesses, the SOAP optimization provides a small benefit because the resulting remote accesses dominate the execution time (*e.g.,* Sobel). In cases where SOAP provides little benefit, SOAC or Remote Update may improve the performance by reducing the number of messages (*e.g.,* Sobel and UPCFish).

Unfortunately, the only benefit from the SOAS optimization is observed in the synthetic benchmarks used for testing. None of the benchmarks tested in Chapter 8 shows any improvement as a result of the SOAS optimization. The kernels in these benchmarks are relatively small and thus the computations run much faster than the time required for communication, providing only a small overlap between the computation and communication.

Based on these results, we can draw the following conclusions

1. Individual optimizations can improve performance moderately but all optimizations together can improve performance significantly. The elimination of a single overhead

may improve performance slightly, but the execution time is still dominated by other overheads. This observation was most evident in the STREAM benchmark when using pointer-to-shared affinity.

The combination of all optimizations deliver dramatic performance improvements. In many cases, the resulting performance is competative with OpenMP performance. Thus, it is important for the compiler to be able to identify opportunities for all of these optimizations in order to achieve good performance.

2. To achieve scalable performance that is competitive with existing programming models, it is essential to combine compiler optimizations and runtime optimizations. For instance, when the compiler can prove the locality of objects, the RTS can be bypassed and the shared data can be accessed directly. However, when the compiler cannot prove the locality of a shared-object it can perform additional analysis and generate calls to optimized RTS functions. This synergy between the compiler and the RTS was demonstrated by the SOAC, SOAS and Remote-Update optimizations, all of which used optimized routines in the RTS to reduce the amount of communication between UPC threads, resulting in improved performance.

Chapter 11

Future Work

The current locality analysis requires the number of threads, the number of nodes, and the blocking factor to be known at compile time. This information allows the compiler to compute the relative locality of each shared reference. Strictly speaking, this is not necessary. The compiler can compute the locality information symbolically by generating expressions to compute the information at runtime. When some (or all) of the values are known at compile time, these equations can be simplified and possibly removed (*i.e.* computed at compile time by the compiler). Thus, the locality analysis can be performed even when all of the information is not known at compile time at the expense of additional runtime computations.

Another limitation of the current locality analysis is that it only analyzes shared references that have the same layout (blocking factor) as the shared reference used in the affinity test. In order to compute the locality for shared references that have different layouts, the compiler would need to compute the *ranges* where the affinity of the two shared references overlap. The use of ranges may cause more conditions to be generated in the loop and further divide the iteration space, which can have negative side effects (*e.g.,* excessive code growth can reduce instruction-cache performance). At this point we have not seen situations where shared arrays with different layouts have been used in the same benchmark. However, if such situations are identified, this extension may enable the compiler to improve the locality analysis, thereby exposing more opportunities for the locality optimizations.

As discussed in Chapter 6.2.3, loop unrolling is a well-known compiler optimization that replicates the body of a loop several times in order to increase the number of instructions in the loop body. When performed in conjunction with the SOAC optimization, loop unrolling can be used to create additional opportunities for the COALESCESOA algorithm. For example, when the inner loop in the Sobel benchmark is unrolled by 2 (the loop body

is replicated once), the compiler is able to coalesce the accesses from the current iteration and the next iteration, resulting in another 50% reduction in the number of messages sent.

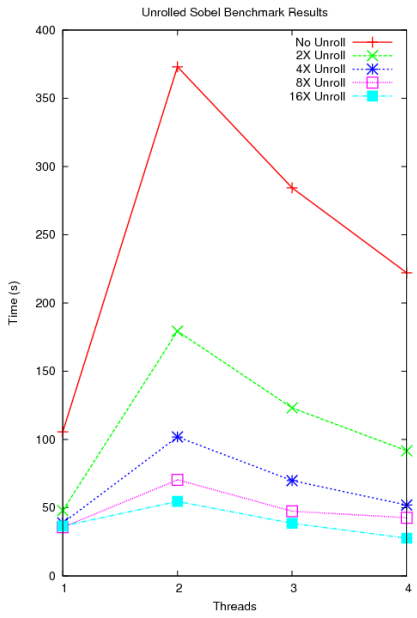


Figure 11.1: Manually unrolled Sobel kernel

Figure 11.1 shows preliminary results of combining loop unrolling with the COALESCESOAalgorithm (from [8]). The results were obtained by manually unrolling the inner loop in the Sobel kernel and then applying the SOAC algorithm. The results in Figure 11.1 demonstrate that unrolling can provide a significant increase in performance when used in conjunction with the SOAC optimization. There are two main tradeoffs to consider when performing loop unrolling for coalescing: (i) increase in message size; (ii) decrease in the number of messages. An increase in the size of individual messages could lead to performance degradation in the network depending on the available network bandwidth and the number of messages that are being sent. On the other hand, decreasing the number of messages can improve program performance (as seen in Figure 11.1) because less time is spent waiting for messages to complete. We are currently investigating techniques for the compiler to identify and manage these tradeoffs.

The UPC runtime currently uses a naive mapping of threads to nodes where given N threads-per-node, threads $0 \ldots N-1$ are located on node 0, threads $N \ldots 2N-1$ are located on node 1, *etc*. It is possible that an alternate mapping of threads to nodes could reduce the number of remote shared-object accesses, thereby increasing the program's performance. The shared reference map can be used by the compiler to create an alternate mapping of

threads to nodes. This remapping will not impact the memory layout of the shared data within a node, but will change how threads are mapped to nodes (which is currently not specified in the UPC Language Specification).

```
shared double A[N];
shared double B[N];

void foo() {
  int i;
  upc_forall (i=0; i < N; i++; &A[i]) {
    A[i] = B[i+2];
  }
}
```

(a) Sample **upc_forall** loop

Thread		Shared References
0	$\rightarrow$	A_i
1	$\rightarrow$	ϕ
2	$\rightarrow$	B_{i+2}
3	$\rightarrow$	ϕ

(b) Shared Reference Map

Figure 11.2: Example **upc_forall** loop and corresponding shared reference map

Consider the example **upc_forall** loop in Figure 11.2(a). Figure 11.2(b) shows the shared reference map created by the compiler when compiling for 4 threads. If this program is run in a hybrid environment with 2 threads-per-node, the current mapping of threads to nodes used by the RTS will place threads 0 and 1 on the first node and threads 2 and 3 together on the second node. Thus, every iteration of the **upc_forall** loop will contain one local access (A[i]) and one remote access (B[i]). However, using the shared reference map the compiler can determine that placing threads 0 and 2 together on the same node will result in two local accesses and no remote accesses in every iteration of the **upc_forall** loop.

When a UPC program contains simple, non-conflicting, cases such as the one shown in Figure 11.2 no additional analysis is required to determine the ideal thread-to-node mapping. However, it is possible that benchmarks will produce different shared reference maps in different loop nests, depending on how the shared arrays are accessed in each nest. Thus, a profitability analysis will also be necessary in order to choose which maps to use when creating the thread-to-node mapping. For example, the estimated execution frequency of different loop nests can be used to prioritize the shared reference maps that are used to determine the thread-to-node mapping.

Bibliography

[1] Fortran - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Fortran#cite_note-0.

[2] Sobel - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Sobel.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification V1.0*. Sun Microsystems, Inc, March 2008.

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

[5] Christina Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[6] Andrew Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.

[7] Roch Archambault, Anthony Bolmarcich, Călin Caşcaval, Siddhartha Chatterjee, Maria Eleftheriou, and Raymond Mak. Scalable runtime system for global address space languages on shared and distributed memory machines, 2008. Patent number 7380086.

[8] Christopher Barton, George Almási, Montse Farreras, and José Nelson Amaral. A Unified Parallel C compiler that implements automatic communication coalescing. In *14th Workshop on Compilers for Parallel Computing (CPC 2009), (To Appear)*, Zurich, Switzerland, January 2009.

[9] Christopher Barton, Călin Caşcaval, George Almási, Rahul Garg, José Nelson Amaral, and Montse Farreras. Multidimensional blocking in UPC. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 47 – 62, Urbana, Illinois, USA, October 2007.

[10] Christopher Barton, Călin Caşcaval, George Almási, Yili Zheng, Montse Farreras, and José Nelson Amaral. Shared memory programming for large scale machines. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 108 – 117, Ottawa, Canada, June 2006. ACM.

[11] Christopher Barton, Călin Caşcaval, and José Nelson Amaral. A characterization of shared data access patterns in UPC programs. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 111–125, New Orleans, Louisiana, USA, November 2006.

[12] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almási, Basili B. Fraguela, María Jesús Garzarán, David A. Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 48–57, New York, New York, USA, March 2006. ACM.

[13] Dan Bonachea. GASNet specification v1.1. Technical Report CSD-02-1207, University of California, Berkeley, Berkeley, CA, October 2002.

[14] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade high productivity language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60, Santa Fe, New Mexico, USA, April 2004.

[15] Francois Cantonnet, Tarek A. El-Ghazawi, Pascal Lorenz, and Jaafer Gaber. Fast address translation techniques for distributed shared memory compilers. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 52.2, Denver, Colorado, USA, 2005. IEEE.

[16] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 519–538, New York, NY, USA, 2005. ACM.

[17] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 149–158, San Diego, CA, May 1993.

[18] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 14–25, Chicago, IL, USA, 2005. ACM.

[19] Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the Berkeley UPC compiler. In *International Conference on Supercomputing (ICS)*, pages 63–73, San Francisco, CA, USA, June 2003. ACM.

[20] Wei-Yu Chen, Dan Bonachea, Costin Iancu, and Katherine Yelick. Automatic nonblocking communication for partitioned global address space programs. In *International Conference on Supercomputing (ICS)*, pages 158–167, New York, NY, USA, 2007. ACM.

[21] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *International Conference on Parallel Architectures and Compiler Techniques (PACT)*, pages 267–278, Washington, DC, USA, 2005. IEEE.

[22] Philippe Clauss and Benoit Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. In *4th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-4)*, pages 11 – 19, Toulouse, France, January 2000.

[23] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, Francois Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanty, YiYi Yao, and Daniel Chavarria-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 36–47, Chicago, Illinois, June 2005. ACM.

[24] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Steve Luna, Thorsten von Eicken, and Katherine Yelick. *Introduction to Split-C*. University of California, Berkeley, Berkeley, CA, March 1994.

[25] David E. Culler, Andrea C. Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Conference on High Performance Networking and Computing (Supercomputing)*, pages 262–273, Portland, Oregon, November 1993. ACM/IEEE.

[26] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[27] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 186–197, Cambridge, Massachusetts, October 1996. ACM.

[28] Tarek El-Ghazawi and Francois Cantonnet. UPC performance and potential: A NPB experimental study. In *Conference on High Performance Networking and Computing (Supercomputing)*, pages 1 – 26, Baltimore, Maryland, 2002. ACM/IEEE.

[29] Engineering scientific subroutine library ESSL User Guide. http://www-03.ibm.com/systems/p/software/essl.html.

[30] Montse Farreras, George Almási, Călin Caşcaval, and Toni Cortes. Scalable RDMA performance in PGAS languages. Unpublished manuscript, 2008.

[31] Sandeep K. S. Gupta, Shivnandan Durgesh Kaushik, and Chua-Huang Huang. On compiling array expressions for efficient execution on distributed-memory machines. In *International Conference on Parallel Processing (ICPP)*, pages 301–305, Syracuse, NY, August 1993.

[32] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Advanced compilation techniques for Fortran D. Technical Report CRPC-TR93338, Rice University, Houston, TX, October 1993.

[33] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation techniques for block-cyclic distributions. In *International Conference on Supercomputing (ICS)*, pages 392–403, Manchester, England, July 1994.

[34] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[35] Seema Hiranandani, Ken Kennedy, and Chau wen Tseng. Preliminary experiences with the Fortran D compiler. In *Conference on High Performance Networking and Computing (Supercomputing)*, pages 338–350, Portland, Oregon, November 1993. ACM/IEEE.

[36] HPC challenge benchmarks. http://icl.cs.utk.edu/hpcc/.

[37] HPC challenge benchmark results. http://icl.cs.utk.edu/hpcc/hpcc_results_all.cgi?display=opt#.

[38] Y. Charlie Hu, Alan Cox, and Willy Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Conference on High Performance Networking and Computing (Supercomputing)*, pages 63 – 76, Dallas, Texas, USA, 2000. ACM/IEEE.

[39] Costin Iancu, Wei Chen, and Katherine Yelick. Performance portable optimizations for loops containing communication operations. In *International Conference on Supercomputing (ICS)*, pages 266–276, New York, NY, USA, 2008. ACM.

[40] Costin Iancu, Parry Husbands, and Paul Hargrove. HUNTing the overlap. In *International Conference on Parallel Architectures and Compiler Techniques (PACT)*, pages 279–290, Saint Louis, MO, USA, September 2005. IEEE.

[41] Mahmut Kandemir, Prithviraj Banerjee, Alok Choudhary, Jagannathan Ramanujam, and Nagraj Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1251–1297, 1999.

[42] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440 – 451, October 1991.

[43] William Kuchera and Charles Wallace. The UPC memory model: Problems and prospects. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 16–24, Santa Fe, New Mexico, USA, April 2004. IEEE.

[44] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/.

[45] Ashish Narayan. Optimizing data locality and parallelism for scalable multiprocessors. Master's thesis, Louisiana State University and Agricultural and Mechanical College, Louisiana, May 1994.

[46] Rajesh Nishtala, George Almási, and Călin Caşcaval. Performance without pain = productivity: Data layout and collective communication in UPC. In *Symposium on Principles and Practice of Parallel Programming(PPoPP)*, pages 99–110, Salt Lake City, Utah, USA, February 2008. ACM.

[47] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[48] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[49] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):65–109, 2002.

[50] William Pugh and Evan Rosser. Iteration space slicing for locality. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 164 – 184, La Jolla, California, USA, August 1999.

[51] Jagannathan Ramanujam and Ponnuswamy Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[52] Anne Rogers, Martin C. Carlisle, John Reppy, and Laurie Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, March 1995.

[53] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, July/September 2005.

[54] Hiromi Suenaga and Phil Merkey. Fish and sharks. Technical report, Michigan Tech University, September 2003.

[55] UPC Consortium. *UPC Collective Operations Specifications V1.0*, December 2003.

[56] UPC Consortium. *UPC Language Specification, V1.2*, May 2005.

[57] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357 – 361, 1987.

[58] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[59] Zhang Zhang and Steven R. Seidel. A performance model for fine-grain accesses in UPC. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006. IEEE.

[60] Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 199 – 211, Montreal, Quebec, Canada, 1998. ACM.

[61] Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel C programs. *Journal of Parallel and Distributed Computing*, 58(2):301–332, 1999.

[62] Yingchun Zhu and Laurie J. Hendren. Locality analysis for parallel C programs. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):99–114, 1999.

Appendix A

General Compiler Terminology

This chapter provides definitions for general compiler terminology that is used throughout the document. Additional descriptions of these terms can be found in [4], [58].

Basic Block A basic block is a sequence of executable statements that has the following attributes:

- There is a single entry point at the first statement in the block
- There is a single exit point at the last statement in the block

Thus, if one statement in a basic block is executed, all statements in the block must be executed.

Common-subexpression elimination Common subexpression elimination identifies expressions that evaluate to the same value and replaces the expression with a variable holding the value.

```
x = a*b + c;
y = a*b*d;
```

Two expressions

```
tmp = a*b;
x = tmp + c;
y = tmp * d;
```

After common subexpression elimination

Figure A.1: Common subexpression elimination example

Control Flow Graph A Control Flow Graph (CFG) is a directed multigraph. Nodes in the CFG correspond to basic blocks, and edges in the CFG correspond to transfer of control between basic blocks.

Figure A.2 shows a simple control flow graph. The CFG contains four basic blocks ($B1$, $B2$, $B3$, and $B4$), a distinguished *Entry Node* (S) and a distinguished *Exit Node*

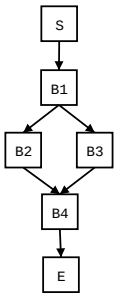


Figure A.2: Simple control flow graph

(E). Every CFG must have exactly one entry node and exactly one exit node. If the CFG has multiple entry or multiple exit nodes, a simple graph transformation inserts a new entry and/or exit node along with the necessary new edges.

Block $B1$ has two successors. For a given execution of this graph, either $B2$ or $B3$ is executed, but not both. The last statement in $B1$ is a test (*e.g.,* an *if* statement). Based on the outcome of the test, one of the two paths is selected and the subsequent code executed. This test is called a *decision point*.

Block $B4$ has two predecessors, meaning that the entry point into $B4$ can be from either $B2$ or $B3$. $B4$ is called a *join node* because it joins two (or possibly more) different paths.

Countable loop A loop is countable when the upper bound and increment of the loop do not depend on values that are computed in the loop body.

Data Dependencies Two statements S_i and S_j that access the same variable (or memory location) have a *data dependence* between them.

There are four kinds of data dependencies that can occur, based on the type of access that the statements are performing. Figure A.3 illustrates the four types of data dependencies between two statements.

1. Input dependence occurs when S_i and S_j both read from the same location (Figure A.3(a)).
2. Output dependence occurs when two statements both write to the same location (Figure A.3(b)). An output dependence from statement S_i to S_j is written as $S_i \delta^o S_j$.

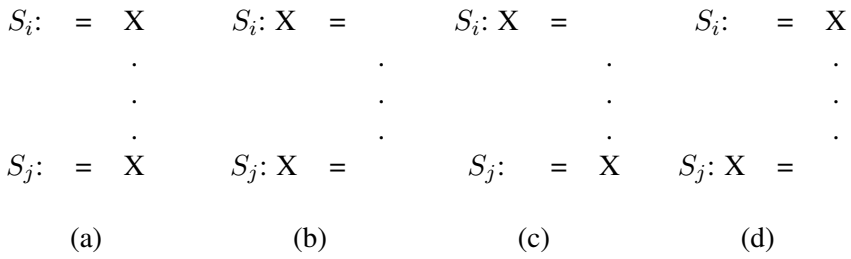


Figure A.3: Types of data dependencies

3. Flow dependence occurs when S_i writes to a location and a subsequent statement S_j reads from the same location (Figure A.3(c)). A flow dependence from statement S_i to S_j is written as $S_i \delta^f S_j$.

4. Anti-dependence occurs when S_i reads from a location and a subsequent statement S_j writes to the same location (Figure A.3(d)). An anti-dependence from statement S_i to S_j is written as $S_i \delta^a S_j$.

Data Dependence Graph A Data Dependence Graph (DDG) is a directed multigraph that represents data dependencies between statements. A node in a DDG corresponds to either a single statement or a group of statements. Edges in the DDG represent data dependencies between nodes. A directed edge from node N_i to N_j represents a data dependence from N_i to N_j. Edges in the DDG can have attributes that represent the type of data dependence the edge represents and a distance/direction vector.

Data Dependence in a loop When data dependencies occur between two statements in a loop, they are generally discussed in terms of the distance of the iteration space containing the statements. That is, when we refer to the distance of a data dependence in a loop, we are referring to the number of loop iterations that the dependence crosses.

Dominance A CFG node N_i *dominates* a node N_j if every path from the entry node to N_j goes through N_i. When N_i dominates N_j, the relationship is described using the notation $N_i \prec_d N_j$.

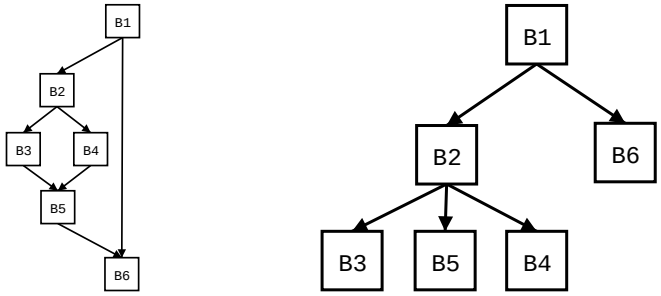
In Figure A.4(a), $B1$ dominates $B2$, $B3$, $B4$, $B5$, and $B6$ because the only way to get to each of those nodes is through $B1$. On the other hand, $B3$ does not dominate $B5$, because there is a path from $B2$ to $B5$ that does not contain $B3$. Similarly, $B4$ does not dominate $B5$.

A node N_i strictly dominates a node N_j if N_i dominates N_j and $N_i \neq N_j$. A node

N_i immediately dominates a node N_j if N_i strictly dominates N_j and there is no other node N_k such that N_i dominates N_k and N_k dominates N_j.

Dominator Tree A dominator tree represents dominance relationships. The root of the dominator tree is the entry node. There is an edge from a node N_i to a node N_j in the dominator tree if and only if N_i immediately dominates N_j. Every node in the dominator tree only dominates its decedents in the tree.

Figure A.4(b) presents the dominator tree for the Control Flow Graph of Figure A.4(a).



(a) Control flow graph (b) Associated dominator tree

Figure A.4: Example control flow graph and dominator tree

We refer to the sub-tree rooted at a node N_i as the dominator tree of node N_i. The dominator tree for each CFG node can be efficiently represented using bit vectors. Every node in the CFG has an associated bit in the bit vector. Consider the bit vector that represents the dominator tree for a node N_i. The number of the CFG node provides the index for the associated bit in the bit vector. A 0 bit means that the corresponding node in the vector is not dominated by N_i. A 1 bit means that the corresponding node is dominated by N_i.

For example, the dominator tree for $B2$ in Figure A.4(a) is 011110. The dominator tree for $B6$ is 000001 because $B6$ only dominates itself.

Induction Variable An induction variable is any scalar variable for which the value of the variable on a given iteration of the loop is a function of loop invariants and the iteration number $(0, \ldots, n)$ [58].

During loop normalization in TPO, two types of induction variables are created: *controlling* induction variable and *derived* induction variable. The controlling induction variable is the variable that controls the execution of the loop. The derived induc-

tion variable is any variable whose value is a function of the controlling induction variable.

Iteration count The iteration count of a loop is the number of times that the loop body executes. For instance, a countable loop in the C language, with an upper bound U, a lower bound L and an induction variable that is incremented by 1 every iteration has an iteration count of $U - L + 1$.

Loop Invariant Expression A loop invariant expression is an expression whose result is not based on the iteration of a loop. That is, the value of the expression does not change throughout the execution of the loop.

Loop Invariant Code Motion Any expression that is loop invariant can move from inside the loop body to outside the loop body. The benefit of this is that the expression will only be evaluated once as opposed to being evaluated in every iteration of the loop.

Lower bound The lower bound of a normalized loop is the minimum value that the controlling induction variable can be assigned to during loop execution. If the controlling induction variable is less than the lower bound, the loop body does not execute.

Nesting level The nesting level of a loop is equal to the number of loops that enclose it. Loops are numbered from the outermost to the innermost nesting level, starting at 0.

Given two loops, L_i and L_j, if L_i dominates L_j, or if L_j postdominates L_i, then L_i and L_j must be at the same nesting level. That is, dominance relationships are only reported between loops that are at the same nesting level.

Normalized Loop A normalized loop is a countable loop that has been modified to start at a specific lower bound and iterate, by increments of 1, to a specific upper bound.

Postdominance A CFG node N_j *postdominates* a node N_i if and only if every path from N_i to the exit node goes through N_j. When N_j postdominates N_i, the relationship is described using the notation $N_j \prec_{pd} N_i$.

In Figure A.4(a), $B6$ postdominates $B1$, $B2$, $B3$, $B4$ and $B5$ because the only way to get from each of those nodes to the exit node is through $B6$. On the other hand, $B3$ does not postdominate $B2$ because there is a path from $B2$ to $B5$ that does not contain $B3$. Similarly, $B4$ does not postdominate $B2$.

The postdominance relationship can also be represented in a tree structure. The root of the tree is the exit node. Every node in the tree only post dominates its decedents

in the tree. Postdominator trees for specific CFG nodes can also be represented using bit vectors. The representation is exactly the same as for dominator trees, except that the postdominance relationship is represented instead.

The postdominator tree for $B6$ in Figure A.4(a) is 111111. The postdominator tree for $B5$ is 011110. Likewise, the postdominator tree for $B1$ is 100000 because $B1$ only postdominates itself.

Regions A region is a collection of basic blocks B, in which a header node dominates all the other nodes in the region. A consequence of this definition is that any edge from a node not in the region to a node in the region must be incident in the header.

A *Single Entry, Single Exit*(SESE) region (also known as a *Contiguous* region) is a region in which control can only enter through a single entry node and leave through a single exit node. Thus, when traversing the nodes in a SESE region, the following two conditions must be met: (i) from the entry node, all nodes in the region can be reached, (ii) during the traversal, no node not in the region is reached before the exit node.

Strength reduction Strength reduction attempts to replace a computationally expensive instruction with a less-expensive instruction. For example, an integer division operation where the divisor is a power of two is replaced with a bit-shift operation.

Strip-Mining Strip-mining is an optimization that divides the iteration space of a loop into *strips*. This is done by replacing the loop with a two-level loop nest. The outer loop iterates over the strips while the inner loop iterates within the strips. Figure A.5 shows an example loop before and after strip-mining.

```
for (i=0; i < N; i++) {
  A[i] = i/N;
}
```

(a) Original loop

```
for (j=0; j < N; j+= STRIP) {
  for (k=0; k < STRIP; k++) {
    A[j+k] = (j+k)/N;
  }
}
```

Loop after strip-mining

Figure A.5: Strip-mining example

Upper bound The upper bound of a normalized loop is the maximum value that the controlling induction variable can be assigned to during the execution of the loop body.

Whenever the controlling induction variable is greater than the upper bound, the loop body does not execute.