Illuminating the UPC Memory Model

By
William Kuchera

A THESIS

Submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY
2003
This thesis, "Illuminating the UPC Memory Model", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT: Computer Science

Signatures:

Thesis Advisor: ________________________________  Dr. Charles Wallace

Department Chair: ______________________________  Dr. Linda Ott

Date: ______________________________
Abstract

Shared memory multiprocessors offer superior computational power to single processor machines. The interaction between processors and the memory system on such a platform must be well understood by programmers. Memory (consistency) models reduce the level of nondeterminism in shared-memory systems, by guaranteeing what can be read from memory based on the previous behavior of a system.

Unified Parallel C (UPC) an extension of ANSI C with support for parallel programming, includes a memory model in its specification. Programmers must have a clear understanding of the UPC memory model to write efficient and correct programs. Language implementors must understand the memory model to know when UPC programs can be optimized.

This work formalizes the memory consistency model of UPC using ASMs. In doing so, we point out some problems with the current UPC memory model definition. Abstract State Machines (ASMs) are an operational semantics methodology which have been applied to several problem domains, including a variety of memory consistency models. Our formalization has helped design tests which show invalid behavior under the memory model and unexpected valid behavior. We also examine the performance potential of the memory model with sample UPC programs.
# Contents

1 Introduction ........................................... 2

2 Background ............................................ 3
   2.1 Memory Consistency Models .......................... 3
   2.2 Unified Parallel C .................................. 7
      2.2.1 Discussion .................................. 9
   2.3 Abstract State Machines ............................ 12

3 Formalization .......................................... 14
   3.1 Introduction ...................................... 14
   3.2 An operational semantics approach ................. 14
   3.3 Discussion ...................................... 18

4 Compliance Tests ....................................... 19
   4.1 Introduction ...................................... 19
   4.2 Compliance test cases .............................. 20
   4.3 “Dark corners” test cases .......................... 25

5 Performance Potential ................................. 31
   5.1 Experimental Results .............................. 38

6 Future Work ........................................... 49

A UPC Specification .................................... 52

B Notify/Wait Semantics ................................. 53

C Test Code ............................................. 55
Part 1

Introduction

Shared memory multiprocessors offer superior computational power to single processor machines. Unfortunately the speed and power of a multiprocessor is offset by an increase in programming complexity. Simple issues in a uniprocessor environment become surprisingly complex when multiple processors become available. In particular, multiple processors working with shared memory means reading a location in memory does not always produce the last value written by a given processor.

Memory consistency models (memory models) have been developed to guarantee what can be read from memory based on the previous behavior of a system. This gives programmers the ability to reason about more complex interactions with memory and use this information while writing programs. Competing philosophies about the desirable qualities of a memory model have led to the development of many different models for different multiprocessors. Instead of alleviating the burden of a parallel programmer, this forces her to deal with a new model for every platform for which she needs to program.

More recently, language designers have included memory models as part of language specifications. This means a programmer using the same language on two different platforms will have the same guarantees about memory behavior. Unified Parallel C (UPC) [10], an extension of ANSI C [26] with support for parallel programming, includes a memory model in its specification. To make effective use of UPC, programmers and language implementers need a clear understanding of the memory model. To achieve this goal, the language designers must express memory semantics clearly and precisely. Not doing so leads to problems on two fronts. At one end, a programmer may interpret the memory semantics too loosely and overcompensate by programming too conservatively. At the other end, the semantics may be interpreted more strictly than intended, leading to incorrect expectations of the memory model.

In its current form the UPC specification is deficient in its description of the UPC memory model. To correct this situation there needs to be a formal analysis of the current UPC memory model. The Abstract State Machine (ASM) [1] methodology is an operational semantics formalism which can be applied to many different problem domains. The precision and understandability of ASMs make them an attractive choice to describe the UPC memory model. The work presented in this thesis consists of the following:

1. **Formalizing the memory semantics of UPC using ASMs.** This work will benefit UPC both novice and experienced UPC programmers as well as language implementers who need to reason about the UPC memory model.

2. **Illustrative test cases.** Two types of test cases are developed. Compliance tests indicate behavior prohibited by the memory model. “Dark corner” tests show legal behavior which may be unexpected by programmers.

3. **Potential benefits of the UPC memory model.** Sample programs are developed which can profit in terms of performance from optimizations allowed by the UPC memory model.
Part 2

Background

This section explores the three essential elements of this work. First, several existing and well documented memory models are presented. Then language features of UPC are examined, in particular the language's memory model. Finally, the details of ASMs and the benefits of using them are discussed.

2.1 Memory Consistency Models

Two alternative approaches have emerged to facilitate interaction between processors in a multiprocessor system, message passing and shared memory. With message passing each processor has exclusive access to a piece of memory. If a processor $p_1$ needs to exchange data with another processor $p_2$, $p_2$ must explicitly send a message containing the data to $p_1$. The receiving processor $p_1$ must be prepared to receive the message and have sufficient memory available to store the data. Although this is easy to understand in principle, it becomes cumbersome in practice when coordinating sends and receives among processors or exchanging complex data structures.

The shared memory paradigm introduces a global address space which all participating processors can access. In a shared memory system explicit message passing is not necessary. Different processors exchange data by reading from, and writing to, shared memory. A significant drawback of shared memory is the difficulty in maintaining a consistent view of memory across all processors. Multiple processors may be attempting to read or write to the same memory location at the same time. In a system where processors can cache shared data, the problem is exacerbated because data in a processor’s cache may not be the same data in the global shared memory. To give programmers the ability to make reasonable assumptions about memory behavior, memory consistency models have been developed.

All memory models have two characteristics: they define the ordering of shared accesses and restrict values which can be read from shared memory. A “shared access” is simply an access which operates on shared memory. Typically a shared access is either a read or write, but more exotic access types are possible as we describe later. The ordering constraints of a memory model can be categorized in terms of strictness. A stricter memory model places more restrictions on the ordering, while relaxed models impose fewer restrictions.

Perhaps the best way to get an understanding of memory models is to look at examples of existing models. The models presented here have been well documented [28] and serve as the basis for several other memory models. The term “program order” which is used here and elsewhere is the order which a single processor executes its instructions.

Uniprocessor Consistency

The memory model of a uniprocessor system is so simple and intuitive that the average programmer has probably not given it much thought. We can define the consistency model of a uniprocessor

3
system as follows: a read operation from a location gets the most recent value written to that location. This notion of consistency allows a great deal of flexibility for common compiler optimizations. Consider a program with two consecutive write operations to different locations, the instructions may be executed out of order without losing data consistency. This model is undesirable for parallel computation since it forces all processors to agree on the most recent value.

Here the term “uniprocessor” is used in the system-level sense. A program running on a single processor may be composed of many threads, where the order in which threads execute is decided non-deterministically by the system. However, at any given time only a single thread is executing and possibly writing to memory.

**Sequential Consistency**

Sequential consistency (SC) [21] is one of the earliest attempts at defining a memory model for shared memory systems. Its frequently cited definition is noteworthy for its clarity and ease of understanding:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

What this says is all instructions from a single processor are executed in program order. Furthermore, when all processors have completed there exists a linear order of executed instructions. This does not say that there is only one possible ordering every time the program executes. Rather, for any given execution there is a single ordering imposed on the instructions, from several possible orderings.

Figure 2.1 presents a program which we use to describe possible instruction ordering under sequential consistency. Note that in this example x and y are initially 0 and all processors have synchronized. This program fragment executing under SC can result in one of the following orderings (not all possible orderings are listed):

- First $p_2$ performs both reads first, getting 0 for both variables. Next $p_0$ writes and then $p_1$ writes. Finally, $p_3$ reads 2 for $y$ and 1 for $x$.
- First, $p_2$ reads 0 for $x$. Next $p_0$ writes and then $p_1$ writes. Finally, both $p_2$ and $p_3$ read 2 for $y$, then $p_3$ reads 1 for $x$.
- First $p_2$ reads 1 for $x$ and $p_3$ reads 0 for $y$. Next $p_0$ writes and then $p_1$ writes. Finally, $p_2$ reads 2 for $y$ and $p_3$ reads 1 for $x$.

Although SC is relatively simple, many researchers have found it to be too restrictive. It prohibits standard compiler and architecture optimizations such as instruction reordering and out-of-order execution. This rigidity tends to make SC a less desirable choice of system designers than other memory models.

A property of SC present in several other memory models is coherence (alternatively cache coherence or cache consistency). The notion of coherence we use is defined by Adve and Gharachorloo [2]: the writes to a single location are linearly ordered with respect to all processors. Coherence is typically not considered a memory model because it defines ordering only in terms of a single location; the ordering of accesses to different locations is not a consideration.
<table>
<thead>
<tr>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(x, 1)</td>
<td>write(x, 2)</td>
<td>read(x)</td>
<td>read(x)</td>
</tr>
<tr>
<td>read(x)</td>
<td>read(x)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: Sample Program

Processor Consistency
Processor consistency (PC) [14] is a less restrictive consistency model than SC but still preserves coherence. The restriction of SC that forces a linear ordering of all access is no longer present. This means two processors may “see” instructions from other processors executed in different orders. We can define processor consistency as follows: accesses from a processor are seen by all other processors in program order, the order any two processor see accesses from other processors may be different as long as coherence is maintained. To demonstrate the relaxation offered by processor consistency we give an execution which is possible in Figure 2.1 that is not possible under SC.

- From the perspective of $p_2$, $p_0$ writes first. Then $p_2$ performs both reads and gets 1 for $x$ and 0 for $y$. From the perspective of $p_3$, $p_1$ writes first. Then $p_3$ performs both reads and gets 2 for $y$ and 0 for $x$.

This execution is not allowed under SC because all threads must obey the same ordering, either $p_0$ rights before $p_1$ or $p_1$ rights before $p_2$. The restrictions of PC are not as strong as SC allowing processors to see instructions in different orders. In this example coherence is trivially maintained since there is a single write to each location.

PRAM Consistency
Despite the relaxation from SC to PC the coherence restriction remains. A system must perform additional work to maintain coherence, work which limits system performance. Pipelined RAM (PRAM) consistency was developed by Lipton and Sandberg [22], and is almost identical to processor consistency except it does not require coherence. We can define PRAM consistency as follows: accesses from a processor are seen by all other processors in program order, the order a processor sees accesses from different processors may differ from any other processor.

By removing the coherence restriction the result of a program execution becomes more interesting. Figure 2.2 presents an execution similar to Figure 2.1, except that all accesses occur on the same location. Under PRAM consistency $p_2$ may read 1 and then 2; in the same execution $p_3$ may read 2 and then 1. Clearly, such an execution is neither coherent nor sequentially consistent. The further relaxation of PRAM consistency gives a system greater freedom to execute instructions out of order.

Weak Consistency
Until this point the memory models we have looked at operate either on shared reads and writes. Other models take into account additional types of memory accesses. Weak consistency (WC) [9] introduces the notion of a synchronization variable, which affects the ordering of instructions. Under weak consistency a system is allowed to postpone committing a value to memory until it is necessary to do so. It also allows a synchronizing processor to bring itself up to date with all other processors that have synchronized. Deciding when it is necessary to synchronize is left to either the programmer or compiler.

There are three restrictions that must be obeyed in the execution of a weakly consistent program:

1. *Accesses to synchronization variables are sequentially consistent.*

2. *Before a processor can access a synchronization variable all previous non-synchronization (ordinary) accesses must have completed.*
3. Before an ordinary access can be performed all previous accesses to synchronization variables must be performed.

In some sense weak consistency is very flexible since it relies on the user providing synchronization at appropriate execution points. The trade-off for this flexibility is additional complexity for the programmer or compiler.

Release Consistency

Despite the freedom WC allows in a program, it is still more restrictive than it needs to be. If we look at the three properties of weak consistency we notice when a process synchronizes it does two things. First, the processor must complete all previous accesses, and it prevents any succeeding accesses from being performed until the synchronization access is complete. The designers of release consistency [13] decided this was unnecessarily restrictive and derived two separate synchronization operations, acquire and release.

Performing an acquire corresponds to exclusive access to a critical section. By acquiring a process needs to bring itself up to date with other processors but does not need to complete its own pending accesses. This allows an operation that precedes an acquire in program order to finish after the acquire has been performed. A release corresponds to leaving a critical section. By releasing a processor indicates all previous accesses have completed, but it may have also completed an instruction that follows the release in program order.

A release consistent execution adheres to the three following properties:

1. Acquire and release accesses are processor consistent.

2. Before a processor releases all previous accesses must be performed.

3. Before an ordinary access can be performed all previous acquire operations must be performed.

The leniency available in release consistency allows more instruction reordering and potentially increased performance over weak ordering. As before any performance gains rely on the ability of programs to take advantage of features of the model.

Other Models

With the exception of PRAM consistency all the memory models discussed thus far rely on coherence, whether for all shared accesses or synchronization accesses. Location consistency [12] seeks to relax the coherence restriction to allow many compiler and hardware optimizations. Instead of serializing the order of writes to a location, writes to a location are treated as a partial ordered multiset.

The Commit-Reconcile & Fences (CRF) [27, 23] memory model is a low level, mechanism-oriented model targeted toward computer architectures. Under the CRF model all write operations are atomic and globally performed. The authors of CRF advocate coherence as an intuitive restriction expected by most programmers. They note that non-coherent executions rarely, if at all, occur in practice.
2.2 Unified Parallel C

Unified Parallel C (UPC) is a recently developed language that extends ANSI C with support for parallel processing. Several implementations are available and include MuPC [25], Compaq UPC [8], and GCC UPC [11]. UPC views memory as Partitioned Shared Memory. Such a system partitions memory among all participating processes, where memory partitioned to $p_0$ is said to have “affinity” to $p_0$. In UPC a process’s partition is subdivided into a local portion and a shared portion. An object in the local portion may only be accessed by the process to which it has affinity. Data in the shared portion are accessible by all processes.

As a notational aside a processing unit in UPC is called a thread (hereafter “thread” always refers to a UPC thread). For all intents and purposes a thread is synonymous with the standard notion of a UNIX process. A typical UPC program may be composed of several threads, and each thread runs on a separate processor.

Language Basics

UPC retains all the features of standard ANSI C, and many of its extensions stem from the keyword shared. Declaring an object as shared means it will be stored in shared memory and be accessible by all threads. From the programmers perspective getting data from shared memory involves a simple C assignment statement. For example, suppose $x$ references memory with affinity to $T_0$ and $y$ references memory with affinity to $T_1$. To copy data from from $T_1$ to $T_0$ a programmer includes the statement $x=y$ and the system takes care of all the underlying communication.

Particularly noteworthy shared objects are arrays and pointers. In UPC arrays can be stored in shared memory, with elements from the same array distributed among different processors. This means iterating an array, and by extension pointer arithmetic, is much more sophisticated than standard C. More details on the intricacies of UPC pointers can be found in existing UPC documentation [10, 6].

Programmers are given tremendous flexibility when dealing with shared memory. However, a central assumption in the design philosophy of UPC is that data in a given processor’s partition is most efficiently accessed by that processor. For a programmer to write efficient programs she must take into account the data distribution across processors. Accessing shared data is potentially expensive if the data resides in the shared portion of memory of a processor different from the one accessing the data. It depends on the underlying architecture how much more expensive reading remote shared data is than reading local data. Regardless, a programmer would be naive not to consider data locality when designing a UPC program.

Memory Modes

Besides the ability to declare data as local or shared UPC introduces two “memory modes”, strict and relaxed. By performing accesses in different modes a programmer affects the ordering of shared accesses within a program. As the names suggest, the behavior of a strict reference is more conservative than the behavior of a relaxed reference.

Within a program the memory mode can be toggled with different degrees of granularity. A single mode may be used for an entire program. Blocks of several lines of code can be specified to execute in a single mode. Alternatively, instructions in a single line of code may execute in different modes. A programmer has three choices for specifying the memory access mode of a shared access, header files, pragmas, or type qualifiers.

To specify the memory mode of all unqualified shared accesses in a program a programmer can include one of two header files, strict.h or relaxed.h. If neither header file is included the default mode is relaxed. Allowing all accesses in a program to execute in a single mode is reasonable, but mixing access modes within a program is possible. To override the memory mode specified by a header file a programmer can insert pragmas into a block of code. A pragma specifies the memory mode of all unqualified accesses from the pragma to the end of the code block of the pragma. A programmer may want a shared object to always operate in a single memory mode. Including the type qualifier
shared or relaxed in the declaration of a shared object forces all accesses of that object to be either strict or relaxed.

From this we make a key observation, that the memory mode of an UPC program can be specified on a per instruction basis. This gives a programmer extraordinary flexibility for interacting with memory. The interleaving of relaxed accesses and shared accesses allows for interesting behavior while adding complexity.

The memory modes result in four non-synchronizing types of shared accesses, strict-writes, relaxed-writes, strict-reads and relaxed-reads. In fact, these are the only possible types of shared accesses. Since the memory mode of a UPC program is relaxed by default a read or write will be relaxed unless otherwise specified. For example a strict-write to the variable x is due to either x being qualified as strict, or if x is unqualified, by executing a write to x in a strict region of code. To achieve the same effect as performing a strict write without actually writing a value a user can issue a upc fence. It is equivalent to a “null strict reference” [10].

Synchronization

Along with the four previously mentioned shared accesses UPC includes support for global synchronization. The statement upc barrier causes a thread to halt execution until all other threads have executed their corresponding barrier. Once each thread has reached the upc barrier all threads may resume execution. Viewed from an implementation standpoint a barrier is composed of two distinct operations. A thread first notifies all other threads that it has reached the barrier and then waits until all threads have notified. UPC allows programmers to use two split-phase barrier statements upc notify and upc wait for synchronization. Separating a barrier into two separate instructions gives a programmer the opportunity to do local computations after notifying but before having to wait. To understand the UPC memory model we must understand the semantics of the global synch operations. In particular, we need a clear picture of how memory looks after synchronization. The instructions that make up the memory model are summarized in Table 2.1.

### Memory Semantics

The previous two sections were intended to familiarize the reader with the UPC objects that make up the memory model. This section looks at the restrictions imposed on memory by the UPC spec, the two essential sections are available in the appendix. Key points will be highlighted to give the reader a feeling for what is in the specification. First we must explain the notation used by the UPC specification. The term a(T_i, i) denotes a memory access from thread \( T_i \) in memory mode a , where a is either strict or relaxed. The \( i \) term is an integer label corresponding to a program counter which increases as each instruction is executed. For example, \( S(T_i, 10) \) is represents the tenth access performed by thread \( 1 \), and the access mode is strict.

The following list highlights key points in the current UPC specification. In particular the model is discussed by formulating two orderings, an “abstract order” and an “actual order”, and then giving constraints on the two orderings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict-write</td>
<td>strictly ordered write to shared memory</td>
</tr>
<tr>
<td>strict-read</td>
<td>strictly ordered read from shared memory</td>
</tr>
<tr>
<td>relaxed-write</td>
<td>laxly ordered write to shared memory</td>
</tr>
<tr>
<td>relaxed-read</td>
<td>laxly ordered read from shared memory</td>
</tr>
<tr>
<td>fence</td>
<td>strictly ordered null access</td>
</tr>
<tr>
<td>notify</td>
<td>broadcasts arrival to barrier</td>
</tr>
<tr>
<td>wait</td>
<td>stops execution until all threads have notified</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of Shared Accesses in UPC
1. The “abstract order” is a partial ordering of all accesses by all threads such that an access \( x(a,b) \) occurs before \( y(c,d) \) in the ordering if \( a = c \) and \( b < d \).

2. The “actual order(k)” for thread \( k \) is another partial order in which \( x(a,b) \) occurs before \( y(c,d) \) if thread \( k \) observes the \( x \) access before it observes the \( y \) access.

3. \( x(a,b) \) must “occur before” \( y(c,d) \) in actual order(e) if \( a = c \) and \( a = e \) and \( b < d \)

4. \( x(a,b) \) must “occur before” \( y(c,d) \) in actual order(e) if \( a = c \) and \( b < d \) and \( ((x = S) \) or \( y = S) \))

5. UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in [26].

The first point stipulates that for an instruction \( i_0 \) to “occur before” another instruction \( i_1 \) in the abstract order both instructions must come from the same thread \( t_0 \). Additionally, the instruction \( i_0 \) must have been executed before \( i_1 \) on \( t_0 \). For non-synchronizing accesses there is no ordering imposed in the abstract order across different threads. The second point defines an actual order for each thread in terms of “observing”. Nothing else in the specification discusses the semantics of observation.

The third point states that if a thread executes instruction \( i_0 \) before instruction \( i_1 \) then it must observe \( i_0 \) before observing \( i_1 \). In other words, all accesses executed by a thread \( t_0 \) must be observed in the order they were executed. Point four states that a thread must observe an access \( a_0 \) before access \( a_1 \) on a remote thread if either access is strict. This restricts the order observations of remote accesses can occur. As it is, strict accesses only order accesses in a single threads actual order. Nowhere is it specified that observing an access impacts what another thread may observe. The last point simply states that restrictions 3 and 4 can be ignored if they do not change the results of executing the program.

2.2.1 Discussion

The previous section gives an introduction to the description of UPC memory model as given in the UPC specification. We believe in its current form the description is ambiguous, imprecise and difficult to understand. Here we outline four items which we feel need to be clarified.

What does “observe” mean?

In the UPC specification, the actual order is defined in terms of observing the accesses of another thread. However, observation is never defined in either the UPC spec or the ANSI C spec. We believe it needs to be explained, because it is not obvious what the term means. In particular, there is no association at all between what a thread does (in terms of detectable behavior during an execution) and what it “observes”. The notion of observing lies on a plane totally divorced from actual behavior. As a result, any witness — any programmer, tester, or user who executes a given UPC program and sees the results — has no means to determine whether the visible behavior is even legal according to the memory model.

Consider an execution with the following sequences of operations:

\[
T_0: \text{write}(x,1); \text{fence}; \text{write}(x,2) \\
T_1: \text{read}(x,2); \text{read}(x,1)
\]

Here, \( \text{write}(\text{addr}, \text{val}) \) signifies a write operation on address \( \text{addr} \) that stores the value \( \text{val} \); \( \text{read}(\text{addr}, \text{val}) \) signifies a read operation on address \( \text{addr} \) that returns the value \( \text{val} \). We assume that \( T_0 \)'s writes are relaxed and the variable \( x \) is initialized to 0. Furthermore, \( T_1 \) writes the values it reads to an output device, so its reads (and, as a result, the operations it observes) do “affect...its input and output dynamics”, as stated in item 3 of the memory model spec. Discussion on the UPC
mailing list [30] has confirmed our hypothesis that this should be an illegal result. The fence should prevent $T_1$ from reading the value 1 at its second read.

Let us now put ourselves in the position of a witness of this program execution. We wish to check whether the behavior we have witnessed is in keeping with the restrictions of the UPC memory model. The only evidence available to us is what the program actually outputs. In this case, this includes the two values read for $x$. Armed with this evidence, we could reason as follows:

1. Since $T_1$ read the value 2 the first time, it must have observed $T_0$’s second write.
2. If $T_1$ reads the value 1 the second time, it must have observed $T_0$’s first write.
3. This implies that $T_1$ must observe $T_0$’s fence, since it turns out that this fence does indeed “affect...its input and output dynamics”. The reasoning here would go as follows: if $T_1$ does not observe the fence, it can read 1 the second time, but if it does observe the fence, it can not read 1. Since there is a possible difference in what is read (and output), $T_1$ had better observe the fence.
4. Then since the fence is really a strict reference, $T_1$ must observe $T_0$’s first write before it observes the fence, and it must observe $T_0$’s second write after it observes the fence.
5. This would seem to indicate that if $T_1$ reads the value 2 first, it has already observed $T_0$’s first write, but the value of that write has been “overwritten” (in $T_1$’s view) by $T_0$’s second write. Therefore, $T_1$ cannot get 1 for its second read.

Notice that this reasoning makes a couple of important assumptions about what it means to observe an operation. The first assumption comes in steps 1 and 2: from seeing a given value $a$ for address $x$ written to output, we conclude that $T_1$ must have observed a write of $a$ to $x$. The second assumption comes in step 5: once we establish that $T_1$ has observed writes of 1 and 2 to $x$ in order, we conclude that the “older” value 1 has been “overwritten” by the “newer” value 2, and hence 1 is no longer visible to $T_1$.

As it is, nothing is said about what “observing” means – so its meaning is left up to the reader’s imagination. This could easily lead people to invalid conclusions. Consider the following code fragments:

$$
T_0: \quad x = -1; \; x = 1;
T_1: \quad \text{if (x==1)} \; y = 2;
T_2: \quad \text{temp1 = y; temp2 = x;}
$$

Here, $T_0$’s writes are strict, all others are relaxed, and $x$ and $y$ are initialized to 0. Say these fragments are executed concurrently, and $T_2$ gets the value 2 for $y$. It certainly seems natural for a programmer to reason, “Since $T_2$ obviously observed $T_1$’s write, and that write was contingent on $x$ having the value 1, $T_2$ must also have observed $T_0$’s second write.” The programmer would then conclude that $T_2$ cannot get -1 when it reads $x$. This uses a more sophisticated, “causal” notion of observation, which seems perfectly reasonable. However, discussion with UPC experts [30] has resulted in disagreement about the legality of this execution.

**How ordered does each “actual order” need to be?**

For each thread $T$, the memory model spec defines a partial order “actual-order($T$)”, representing the order in which $T$ observes accesses. This order is defined quite loosely — so loosely, in fact, that it permits certain behaviors of doubtful legality. Consider the following execution:

$$
T_0: \quad \text{write(x.1); read(x.2); read(x.1)}
T_1: \quad \text{write(x.2)}
$$

Is this a valid execution? It all depends on actual-order(0). If the two writes are ordered in actual-order(0), then $T_0$’s first read would indicate that $T_0$’s write is ordered before $T_1$’s write.
Following the “overwriting” assumption of the previous section, the value 1 would not be possible for the second read.

But actual-order(0) is simply defined as a partial order, so there does not seem to be any reason to assume that the writes are ordered with respect to each other. If the writes are not ordered in actual-order(0), then the second read could legally return 1. So should this be a legal result? Communication with Bill Carlson [7], the lead designer of UPC, indicates that this is not intended to be legal. If the intent is to disallow this result, something more about actual-order(0) must be stated in the memory model spec – perhaps, writes to a single location are linearly ordered in each thread’s actual order.

To circumvent these problems, we propose adding some material to the official description: in particular, we provide a link between visible behavior and the notion of “observing”, through a number of axioms. These axioms are based on our understanding of the language designers’ intent, based on conversations we have had [7].

The definition of “abstract order” is circular

The memory model spec relies on the notion of an “abstract order” on accesses. In item 3, we are asked to think of the accesses of each thread as labeled with integers according to their order of execution; these integers “monotonically increase as the evaluation of the program proceeds from startup through termination.” In the abstract order, accesses by a single thread are ordered linearly according to their integer labels. However, there is a circularity here: the accesses that a thread performs, and the order in which it performs them, may depend crucially on the memory model. Consider the following code fragment:

\[ T_0: \quad \text{if } (x == 1) \ y = 2; \text{ else } z = 3; \]

Assume that threads other than \( T_0 \) also access \( x \). Let us establish the abstract order for the accesses by \( T_0 \). First of all, what are the accesses that \( T_0 \) performs? Clearly, a read of \( x \)— but then, it performs either a write to \( y \) or a write to \( z \), depending on the value it reads for \( x \). This in turn depends on the definition of the memory model, which is based on the abstract order.

The problem here is that the memory model rests on a well defined notion of “program order” of accesses, but “program order” is not well defined without a memory model. Note that many other memory-model definitions suffer from an \textit{a priori} notion of “program order”. Rudolph, Arvind and Shen comment on this in their original CRF paper [27].
2.3 Abstract State Machines

Abstract State Machines are operational semantics which have been successfully applied to several different domains, including programming languages [18, 31], concurrent algorithms [5] and memory consistency models [19, 32, 4]. This section provides a primer to ASMs and discusses the benefits of using them. A brief introduction to ASMs is provided; further documentation can be found in one of several resources [15, 16, 17, 20].

ASM Basics

An ASM consists of three main elements: functions, universes and rules. The collection of all function names for a given ASM is called a vocabulary. Values (or interpretations) of functions are elements of the superuniverse. The superuniverse is logically divided into separate components called universes. Universes are collections of similar types of elements. A universe may be something general such as the collection of real numbers. Alternatively a universe may be domain specific, like the collection of all instructions in a programming language.

The “state” in Abstract State Machine is the interpretation of all function names at a given step in an execution of the abstract machine. This notion of state as compared to a Turing Machine is much richer. A state in a Turing Machine is just a state with very low level attributes, such as the control state, tape head position, and tape contents. In an ASM a state can be much more descriptive, represented in terms of sets, tuples, or even more sophisticated data structures like queues or graphs.

An execution of an ASM is called a run, where the interpretation of function names are updated with each new state. From state to state a machine will update several function names in parallel. A run is conceptually similar to using a debugger to step through a program in your favorite programming language. At each step a user can look at the state of the program based on the value of all the variables. Unlike a debugger, which executes only a single instruction at a step, an ASM typically updates several functions in parallel.

Moving from one state to another requires transition rules. Rules take into account the current state of a machine and then fire appropriately. Before we look at the types of rules we must define terms. A term in an ASM is defined as either a variable or \( f(t_1, t_2 \ldots t_n) \) where \( f \) is a function name of arity \( n \), where \( n \geq 0 \), and \( t_i \) is a term. Table 2.2 lists various rules along with short descriptions of each.

To successfully describe distributed or parallel algorithms we need additional tools. A distributed ASM uses an additional universe of Agents. An ASM may consist of several agents executing concurrently. All agents execute the same program but at any state different agents may be firing different rules. Since UPC is intended for distributed environments we will need Agents in our formalization of the memory model.

In essence, designing an ASM is simply a matter of defining the universes, functions and the appropriate transition rules. Although this is a non-trivial undertaking, it requires little more than basic programming experience and knowledge of the problem domain.

Why ASMs?

Several different features of ASMs make them appropriate for this work:

- **Precise.** ASMs have been developed as precise mathematical tools. This allows researchers to seamlessly incorporate ASMs in proofs. This is especially useful for work with memory models, where certain properties may need to be verified.

- **Understandable.** The style of writing ASMs is similar to writing properly documented code in any modern programming language. This means a person with moderate programming experience should be able to read an ASM and have little trouble understanding it. For a
<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>( f(t_1, t_2, \ldots, t_n) := t_0 )</td>
<td>Updates the interpretation of the function name ( f ) across terms ( t_1, t_2, \ldots, t_n ) to ( t_0 ).</td>
</tr>
<tr>
<td>Conditional</td>
<td><code>if cond then</code> ( \text{rule}_1 ) <code>else</code> ( \text{rule}_2 ) <code>endif</code></td>
<td>Evaluates the condition ( \text{cond} ) and fires ( \text{rule}_1 ) if it is true, otherwise ( \text{rule}_2 ) is fired.</td>
</tr>
<tr>
<td>Do-In-parallel</td>
<td><code>do-inparallel</code> ( \text{rule}_1 ) <code>\ldots</code> ( \text{rule}_n ) <code>enddo</code></td>
<td>Fires ( \text{rule}_1 ) through ( \text{rule}_n ) in parallel.</td>
</tr>
<tr>
<td>Choose</td>
<td><code>choose \ x: U; P(x)</code> ( \text{rule}_1 ) <code>endchoose</code></td>
<td>Chooses an element in Universe ( U ) consistent with ( P(x) ) and fires ( \text{rule}_1 ).</td>
</tr>
<tr>
<td>Do-Forall</td>
<td><code>do-forall \ x: U; P(x)</code> ( \text{rule}_1 ) <code>enddo</code></td>
<td>Selects all elements in Universe ( U ) consistent with ( P(x) ) and fires ( \text{rule}_1 ) with all the elements in parallel.</td>
</tr>
</tbody>
</table>

Table 2.2: ASM Rules

…

programmer trying to grasp something as complex as a memory model, ASMs can be valuable educational aids.

- **Operational.** Given the description of an ASM the machine can be simulated with little effort. While working through the run of a machine is possible, it is probably tiresome for long runs. Several tools exist to allow an ASM to be simulated automatically.

- **Differing levels of abstraction.** Unlike other models of computation, such as Turing Machines, the level of abstraction of an ASM can be tailored to a specific algorithm. This gives algorithm designers the ability to postpone including unnecessary detail while preserving the ability to simulate the model.
Part 3

Formalization

3.1 Introduction

Providing a precise memory consistency model using ASMs should alleviate at least some of the problems with the current UPC spec. Here the view of a UPC program execution is quite high-level, in keeping with the definition in the memory model spec: each thread produces a stream of accesses (reads and writes), as well as barrier statements (fence, notify, and wait).

We begin by providing a translation from the official memory model spec to our semantics. This requires adding some material to the official spec: in particular, a link is provided between visible behavior and the memory model spec's notion of "observing". These axioms are based on an understanding of the language designers' intent, based on conversations with them.

If a thread performs a read at an address $a$ and gets back a certain value $v$ (as made visible to a witness through a write to a file or output device), what can we say about what it must have observed? It seems reasonable to infer that it observed a write of $v$ to $a$. The first axiom states that threads read values provided by earlier write accesses, rather than "out of thin air". The second axiom ties together the notions of reading and observing: reading is only possible for observed write accesses. Finally, the third axiom asserts the assumption about write accesses “overwriting” one another.

- The “not out of thin air” axiom: If thread $t$ performs a read at address $a$ that returns value $v$, then $t$ must read a write $w(a,v)$.
- The “observe what you read” axiom: If $t$ reads a write $w$, then $t$ must have observed $w$.
- The “overwriting” axiom: Let $w(a)$ and $w'(a)$ be writes observed by $t$. If $w < w'$ in actual-order($t$), then $t$ can no longer read $w$.

3.2 An operational semantics approach

Universes

Now the new and improved operational semantics are provided. The ASM definitions are annotated with comments which we hope will be illustrative. To begin the universes are defined. This ASM consists of thread agents of the universe Thread that execute instructions of the universe Instruction. A universe Action contains the type of actions that an instruction will perform.

In describing the memory model of UPC the focus is the ordering of shared memory accesses or events. A universe Event is defined containing all executions of write, read, and fence instruction in the system history. Read and write instructions operate on addresses, which form the universe Address. An address may take any value from the universe Value.
To facilitate global synchronization, UPC uses two so-called “split-phase” barrier statements, upc.notify and upc.wait. An integer label is associated with each upc.wait or upc.notify statement. The label of any upc.wait must match the label of all upc.notify statements in the same phase. The universe BarrierLabel is the universe of barrier statement labels. A synchronization phase is defined in §6.5.1 of the UPC spec as the collection of statements between upc.wait statements. We introduce the universe Phase of synchronization phases.

Functions

Here the functions used in the ASM are defined to relate elements of the various universes. Each thread agent needs to keep track of the instruction to be executed. The function currInstr maps each thread to its current instruction. Another function nextInstr maps an instruction to the next instruction to be executed. For any instruction we need the ability to extract information about its attributes: the instruction type, the address being read/written (in the case of a read or write), the value being written (in the case of a write), the barrier label (in the case of a notify or wait), the consistency mode (in the case of a read or write). The functions type, value, addr, label, and mode map an instruction to these various attributes. Of course, some of these attributes will be undefined for any given instruction.

When a thread issues a write, certain attributes of the write must be established. The functions thr, addr, and val map events to threads, addresses, and values, respectively. The function type maps events to their respective types: write, read, or fence. A thread performing a read uses these attributes when selecting a valid write.

In UPC, a thread is more restricted when reading a write issued by itself than reading a write from another thread. Only the last write to a location can be read by the thread that issued the write, although there may be many writes from other threads which can be read for that location. To keep track of the latest write function maxLocal is used, which maps a thread and an address to a write.

As mentioned earlier, a memory consistency model can be described in terms of two characteristics: the precedence relation between shared accesses and the values that can be returned from a read. The relation < specifies how shared accesses are ordered across all threads. We use a partially ordered precedence relation to describe the dual-mode memory model of UPC. It is helpful to think of each thread maintaining a history of shared accesses. All accesses created by relaxed instructions are ordered after the most recent strict access. A strict access is ordered after the most recent strict access in the history. The function maxStrict maps a thread to its latest strict access.

If a thread reads a write from another thread, it must obey the restrictions imposed by the memory model spec. To determine which writes it can read, a thread agent must keep track of the maximal write it has read from a remote thread. The function maxRemote maps the reading thread and a remote thread to the maximal event read from the remote thread.

To perform synchronization operations, thread agents must store information regarding phases and barrier labels. The function phase maps a thread to its current synchronization phase. Another function nextPhase maps a phase to the next synchronization phase. Each phase contains at most one notify statement and notify label. Once a thread reaches a upc.wait it must compare the label of the upc.wait to the label of all the upc.notifies issued in the same phase as the upc.wait. This comparison necessitates the function notifyLabel that maps a given thread and phase to a barrier label. If a thread has not yet reached the upc.notify in a certain phase the function is undefined. When a thread has reached a upc.notify, all waiting threads are brought up to date with regard to the thread’s most recent event(s) before the upc.notify. The function phaseMax maps a thread and a phase to the most recent event before a notify.

Rules

Now the rules describing the dynamics of any UPC memory system are given. We view each thread as operating on a stream of instructions. With the exception of a wait instruction, we conceive of each instruction type as taking a single step.
module Thread;
let i = Self.instr
case i.type of
  write: Write i.val to i.addr in mode i.mode
  read: Read to i.addr in mode i.mode
  fence: Fence
  notify: Notify i.label
  wait: Wait i.label
endcase

rule Proceed:
Self.instr := Self.instr.nextInstr

We begin by focusing on the actions associated with a write. When a write event is issued, the
three attributes associated with it are updated. The write is also added to the thread’s shared access
history. How it is ordered with respect to other events in the history depends on the consistency
mode of the write (strict or relaxed).

rule Write v to a in mode m:
extend WriteEvent with w
  w.thr := Self  w.addr := a  w.val := v
  Self.maxLocal(a) := w
  if m = strict then Order new strict event w
  else Order new relaxed event w
endif
endextend
Proceed

When a strict access is created, it must be ordered in a thread’s shared access history. The new
strict access is ordered after the latest strict access and all relaxed accesses that follow the latest
strict access. The new strict access then becomes the latest strict access. In contrast, a relaxed
write only needs to be ordered after the latest strict access.

rule Order new strict event e:
do-forall d: Event: Self.maxLocalStrict ≤ d
d < e
  Self.maxLocalStrict := e
endo

rule Order new relaxed write event w:
Self.maxLocalStrict < w := true

Here the dynamics behind a read access are presented. To read from a shared address, a thread
has two options: read a local write or a remote write. If a thread reads a write from itself, then
it must choose the latest write to the corresponding location. When a thread reads a write from
another thread, there may be multiple legal choices. For any pair of threads $t_1$ and $t_2$, there is a
maximal event that $t_1$ has observed from $t_2$. A write is (remotely) readable if there is no intervening
write between it and the maximal element read by the reading thread. As a result, the semantics of
reading relaxed writes between two strict accesses is very relaxed indeed.

If $t_1$ reads a write from $t_2$ that is ordered after the current maximal element, the write is updated
as the new maximal element. Keeping track of maxRemote allows us to limit which writes can be
read by one thread from another. If a read is strict, it is ordered as a new strict access.

rule Read to a for r locally:
  r.val := Self.maxLocal(a).val

term w.overwritten?:
  (∃w': WriteEvent: w'.addr = w.addr) w ≺⁺ w' ≺⁺ Self.maxRemote(w.thr)

rule Read to a for r remotely:
choose w: WriteEvent: w.thr ≠ Self and w.addr = a and not w.overwritten?
  r.val := w.val
  if Self.maxRemote(w.thr) ≺⁺ w then Self.maxRemote(w.thr) := w
  endif
endchoose

rule Read to a in mode m:
extend ReadEvent with r
  r.thr := Self  r.addr := a
choose among
  Read to a for r locally
  Read to a for r remotely
endchoose
  if m = strict then Order new strict event r
  endif
endextend
Proceed

A fence is defined as a null strict reference, so it only affects the ordering of shared accesses. The fence is ordered as a new strict access.

rule Fence:
extend FenceEvent with f
  Order new strict event f
endextend
Proceed

The dynamics of notify and wait are explained here. A thread that issues a notify in a given phase must keep track of the associated barrier label. A notify instruction also records the maximal strict event issued by the thread. Other threads will be brought up to date at least up to this maximal event when they reach the corresponding wait.

rule Notify ℓ:
  Self.phaseMax(Self.phase) := Self.maxLocalStrict
  Self.notifyLabel(Self.phase) := ℓ
Proceed

Three possible situations arise when a thread reaches a wait instruction. First, some thread may have issued a notify with a label different from the label of the current wait. This is illegal behavior, according to the UPC spec; all threads must share the same label for a given phase. A mismatch in label values is detected by labelMismatch? and, according to the UPC spec, results in a “runtime error”. Both the UPC and ISO C specs are silent on the meaning of “runtime error”, and no answer
has been forthcoming on the UPC mailing list. For the time being, we leave this portion of the ASM undefined.

The second situation arises if another thread has not reached the corresponding notify; in this case, the waiting thread continues to wait. The final situation is when all other threads have executed their respective notify instructions and there is no label mismatch. In this case, a thread proceeds and updates its current phase. It must also update maxRemote for itself across all other threads, bringing itself up to date with respect to accesses issued by the other threads.

term stopWaiting?(\ell):
(\forall t: Thread) \text{Self.phase} \leq t.\text{phase} \text{ and } t.\text{notifyLabel(Self.phase)} = \ell

term labelMismatch?():
(\exists t: Thread) t.\text{notifyLabel(Self.phase)} \neq \ell

rule Wait \ell:
if stopWaiting? then
   do-forall t: Thread; t \neq \text{Self}
      \text{Self.maxRemote}(t) := t.\text{phaseMax(Self.phase)}
   enddo
   \text{Self.phase} := \text{Self.phase.nextPhase}
   \text{Proceed}
else labelMismatch?(\ell) then \text{“Runtime error”}
endif

3.3 Discussion

The UPC memory model places an order on operations on a per-thread basis. In this regard, it is similar to processor consistency. There is no mechanism for ordering operations by different threads. This makes the UPC memory model strictly weaker than sequential consistency. Strict operations by a single thread are linearly ordered; relaxed operations are ordered with respect to strict operations but unordered with respect to one another. The lack of a linear order on all operations by a single thread makes the UPC memory model strictly weaker than processor consistency.

The constraints on local reading (i.e., a thread’s reading a value that it wrote) are much stricter than those on remote reading (i.e., a thread’s reading a value that another thread wrote). A local read always returns the most recent value written by the thread. A thread remotely reading must follow the ordering of operations established by the writing thread.

Notify and wait bring threads to a consensus on the remotely visible operations for each thread. When each \( T_i \) completes its wait, its view of operations by each \( T_j \) is updated to include operations all the way up to \( T_j \)’s notify. Notify and wait do not order operations by different threads, however, so the operations of \( T_i \) and those of \( T_j \) remain unordered.

The UPC memory model does not have the coherence property. This condition fails in the UPC memory model for two reasons. First, writes by different threads, even those to a common location, are never ordered. Second, relaxed writes by a single thread, even those to a single location, are never ordered.
Part 4

Compliance Tests

4.1 Introduction

As mentioned previously it is difficult to tie the description of the memory model in the UPC spec to actual behavior of a UPC program. The previous section gave a formal basis for understanding the UPC memory model. Here program behavior that complies with the UPC specification versus behavior that does not comply is examined. This is especially important for UPC implementors and application developers. Implementors of UPC on a particular platform need to ensure that the optimizations they employ guarantee compliant behavior. Programmers exploiting relaxed consistency must have a grasp of what possible behaviors their programs may induce.

To this end, a set of test cases for the UPC memory model has been devised. These cases fall into two categories:

Compliance tests. These are examples of behavior that falls outside the UPC specification. They illustrate the consistency guarantees that UPC gives the programmer.

“Dark corner” tests. These are examples of acceptable behavior, according to the UPC specification, that may be surprising to the UPC novice. The UPC memory model is designed to be quite lax and allows some cases that programmers may not anticipate. These examples serve to highlight some of these “dark corners”.

Our test cases are at the architecture level rather than the program level; that is, each thread’s execution is expressed in terms of a sequence of read, write, fence, notify, and wait instructions, rather than UPC statements. Any references to “ordering”, “following”, “intervening”, etc. refer to this thread-by-thread instruction ordering. Read and write instructions are assumed to be relaxed unless specified otherwise. For each case, we illustrate why it illustrates compliance or non-compliance, using our operational semantics.
4.2 Compliance test cases

Compliance Test 1  If ordered writes are followed by a local read, only the latest write may be read.

Thread 0: write(x,1); write(x,2); read(x,?) \(\Leftarrow 1\) not a legal value

\[
\begin{array}{c}
T_0 : \quad \boxed{W(x, 1)} \quad \boxed{W(x, 2)} \quad \boxed{R(x, ?)} \\
\downarrow \\
T_0. \text{maxLocal}(x)
\end{array}
\]

Explanation.
At the time of \(T_0\)'s read, \(t_0. \text{maxLocal}(x) = W(x, 2)\).
The only write to \(x\) locally readable by \(T_0\) is \(\text{maxLocal}(x)\).
So \(W(x,1)\) is not legal for this read.

Compliance Test 2  A strict write and a following relaxed write, if read by a remote thread, must be read in order.

Thread 0: strict-write(x,1); write(x,2)
Thread 1: read(x,2); read(x,?) \(\Leftarrow 1\) not a legal value

\[
\begin{array}{c}
T_0 : \quad \boxed{SW(x, 1)} \quad \boxed{W(x, 2)} \\
\downarrow \\
T_1. \text{maxRemote}(T_0)
\end{array}
\]

\[
T_1 : \quad \boxed{R(x, 2)} \quad \boxed{R(x, ?)}
\]

Explanation.
\(T_0\)'s write \(W(x,2)\) is ordered after its strict write \(SW(x,1)\).
\(T_1\)'s first read updates \(T_1. \text{maxRemote}(T_0)\) to \(W(x,2)\).
Thus at the time of \(T_1\)'s second read, we have \(SW(x,1) \prec W(x,2) = T_1. \text{maxRemote}(T_0)\).
So \(SW(x,1)\) is not legal for this read.
**Compliance Test 3** A relaxed write and a following strict write, if read by a remote thread, must be read in order.

Thread 0: write(x,1); strict-write(x,2)
Thread 1: read(x,2); read(x,?) \(\Leftarrow 1\) not a legal value

\[
\begin{align*}
T_0 : & \quad \boxed{W(x,1)} \quad \boxed{SW(x,2)} \\
& \quad \boxed{T_1.\text{maxRemote}(T_0)} \\
T_1 : & \quad \boxed{R(x,2)} \quad \boxed{R(x,?)}
\end{align*}
\]

Explanation.

\(T_0\)'s strict write SW(x,2) is ordered after its write W(x,1).
\(T_1\)'s first read updates \(T_1.\text{maxRemote}(T_0)\) to SW(x,2).
Thus at the time of \(T_1\)'s second read, we have \(W(x,1) \prec SW(x,2) = T_1.\text{maxRemote}(T_0)\).
So W(x,1) is not legal for this read.

**Compliance Test 4** Ordered strict writes, if read by a remote thread, must be read in order.

Thread 0: strict-write(x,1); strict-write(x,2)
Thread 1: read(x,2); read(x,?) \(\Leftarrow 1\) not a legal value

\[
\begin{align*}
T_0 : & \quad \boxed{SW(x,1)} \quad \boxed{SW(x,2)} \\
& \quad \boxed{T_1.\text{maxRemote}(T_0)} \\
T_1 : & \quad \boxed{R(x,2)} \quad \boxed{R(x,?)}
\end{align*}
\]

Explanation.

\(T_0\)'s strict write SW(x,2) is ordered after its write SW(x,1).
\(T_1\)'s first read updates \(T_1.\text{maxRemote}(T_0)\) to SW(x,2).
Thus at the time of \(T_1\)'s second read, we have \(SW(x,1) \prec SW(x,2) = T_1.\text{maxRemote}(T_0)\).
So SW(x,1) is not legal for this read.
Compliance Test 5  Ordered relaxed writes, if read by a remote thread, must be read in order, if a strict write intervenes between the writes.

Thread 0: write(x,1); strict-write(y); write(x,2)  
Thread 1: read(x,2); read(x,1) ⇔ 1 not a legal value

\[ T_0: \begin{array}{c}
W(x,1) \\
SW(y) \\
W(x,2) \\
\end{array} \]

\[ T_1: \begin{array}{c}
\text{maxRemote}(T_0) \\
R(x,2) \\
R(x,?) \\
\end{array} \]

Explanation.  
\( T_0 \)'s strict write \( SW(y) \) is ordered after its write \( W(x,1) \).  
\( T_0 \)'s write \( W(x,2) \) is ordered after \( SW(y) \).  
\( T_1 \)'s first read updates \( T_1.\text{maxRemote}(T_0) \) to \( W(x,2) \).  
Thus at the time of \( T_1 \)'s second read, we have \( W(x,1) \prec SW(y) \prec W(x,2) = T_1.\text{maxRemote}(T_0) \).  
So \( W(x,1) \) is not legal for this read.

Compliance Test 6  Ordered relaxed writes, if read by a remote thread, must be read in order, if a strict read intervenes between the writes.

Thread 0: write(x,1); strict-read(y); write(x,2)  
Thread 1: read(x,2); read(x,?) ⇔ 1 not a legal value

\[ T_0: \begin{array}{c}
W(x,1) \\
SR(y) \\
W(x,2) \\
\end{array} \]

\[ T_1: \begin{array}{c}
\text{maxRemote}(T_0) \\
R(x,2) \\
R(x,?) \\
\end{array} \]

Explanation.  
\( T_0 \)'s strict read \( SR(y) \) is ordered after its write \( W(x,1) \).  
\( T_0 \)'s write \( W(x,2) \) is ordered after \( SR(y) \).  
\( T_1 \)'s first read updates \( T_1.\text{maxRemote}(T_0) \) to \( W(x,2) \).  
Thus at the time of \( T_1 \)'s second read, we have \( W(x,1) \prec SR(y) \prec W(x,2) = T_1.\text{maxRemote}(T_0) \).  
So \( W(x,1) \) is not legal for this read.
Compliance Test 7  Ordered relaxed writes, if read by a remote thread, must be read in order, if a fence intervenes between the writes.

Thread 0: write(x,1); fence; write(x,2)
Thread 1: read(x,2); read(x,:) ∈ 1 not a legal value

\[ T_0 : \begin{array}{c}
\text{W}(x,1) \\
\text{F} \\
\text{W}(x,2) \\
T_1 \text{.maxRemote}(T_0)
\end{array} \]

\[ T_1 : \begin{array}{c}
\text{R}(x,2) \\
\text{R}(x,?)
\end{array} \]

Explanation.
\( T_0 \)'s fence \( F \) is ordered after its write \( \text{W}(x,1) \).
\( T_0 \)'s write \( \text{W}(x,2) \) is ordered after \( F \).
\( T_1 \)'s first read updates \( T_1 \text{.maxRemote}(T_0) \) to \( \text{W}(x,2) \).
Thus at the time of \( T_1 \)'s second read, we have \( \text{W}(x,1) \prec F \prec \text{W}(x,2) = T_1 \text{.maxRemote}(T_0) \).
So \( \text{W}(x,1) \) is not legal for this read.

Compliance Test 8  Ordered relaxed writes, if read by a remote thread, must be read in order, if a notify intervenes between the writes.

Thread 0: write(x,1); notify(\ell); write(x,2)
Thread 1: read(x,2); read(x,:) ∈ 1 not a legal value

\[ T_0 : \begin{array}{c}
\text{W}(x,1) \\
\text{F} \\
\text{NOT}(\ell) \\
\text{W}(x,2) \\
T_1 \text{.maxRemote}(T_0)
\end{array} \]

\[ T_1 : \begin{array}{c}
\text{R}(x,2) \\
\text{R}(x,?)
\end{array} \]

Explanation.
\( T_0 \)'s notify is preceded by an implicit fence \( F \).
\( F \) is ordered after \( T_0 \)'s write \( \text{W}(x,1) \).
\( T_0 \)'s write \( \text{W}(x,2) \) is ordered after \( F \).
\( T_1 \)'s first read updates \( T_1 \text{.maxRemote}(T_0) \) to \( \text{W}(x,2) \).
Thus at the time of \( T_1 \)'s second read, we have \( \text{W}(x,1) \prec F \prec \text{W}(x,2) = T_1 \text{.maxRemote}(T_0) \).
So \( \text{W}(x,1) \) is not legal for this read.
**Compliance Test 9** If a relaxed write and a following strict write precede a notify and corresponding wait, only the strict write may be read after the wait.

Thread 0: write(x,1); strict-write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); read(x,?) ⊑ 1 not a legal value

\[
\begin{align*}
\text{T}_0 : & \quad [W(x,1)] \quad [SW(x,2)] \quad [\text{F}] \quad [\text{NOT}(\ell)] \\
\text{T}_0 . \text{phaseMax} \quad \downarrow \\
\text{T}_0 : & \quad [W(x,1)] \quad [SW(x,2)] \quad [\text{F}] \quad [\text{NOT}(\ell)] \quad [\text{WAIT}(\ell)] \\
\text{T}_1 . \text{maxRemote}(T_0) \quad \downarrow \\
\text{T}_1 : & \quad [\text{NOT}(\ell)] \quad [\text{WAIT}(\ell)]
\end{align*}
\]

*Explanation (part 1).*
\(T_0\)'s notify is preceded by an implicit fence F.
\(W(x,1)\) is ordered after \(T_0\)'s strict write \(SW(x,2)\).
\(SW(x,2)\) is ordered after \(T_0\)'s write \(W(x,1)\).
At the time of \(T_0\)'s notify, \(T_0 . \text{phaseMax}\) is updated to F.

*Explanation (part 2).*
When \(T_1\)'s wait completes, \(T_1 . \text{maxRemote}(T_0)\) is updated to \(T_0 . \text{phaseMax}\), which is F.
Thus at the time of \(T_1\)'s read, we have \(W(x,1) \prec SW(x,2) \prec F = T_1 . \text{maxRemote}(T_0)\).
So \(W(x,1)\) is not legal for this read.
4.3 “Dark corners” test cases

**Dark Corner Test 1** Ordered relaxed writes may be remotely read in different orders by different threads.

Thread 0: write(x,1); write(x,2)
Thread 1: read(x,1); read(x,2)
Thread 2: read(x,2); read(x,?) ⇐ 1 is a legal value

\[
T_1 : \begin{array}{c}
R(x,1) \\
\text{T}_1.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
R(x,2)
\end{array}
\]

\[
T_0 : \begin{array}{c}
W(x,1) \\
\text{T}_2.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
W(x,2)
\end{array}
\]

\[
T_2 : \begin{array}{c}
R(x,2) \\
\text{T}_1.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
R(x,?)
\end{array}
\]

**Explanation.**
W(x,1) and W(x,2) are unordered with respect to each other.
T_1's first read updates T_1.\text{maxRemote}(T_0) to W(x,1).
At T_1's second read, since there is no chain W(x,2) ⪯^+ W(x,*), W(x,2) is readable.
T_2's first read updates T_2.\text{maxRemote}(T_0) to W(x,2).
At T_2's second read, since there is no chain W(x,1) ⪯^+ W(x,*), W(x,1) is legal for this read.

**Dark Corner Test 2** Ordered relaxed writes may be remotely read in different orders by a single thread.

Thread 0: write(x,1); write(x,2)
Thread 1: read(x,1); read(x,2); read(x,?) ⇐ 1 is a legal value

\[
T_0 : \begin{array}{c}
W(x,1) \\
\text{T}_1.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
W(x,2)
\end{array}
\]

\[
T_1 : \begin{array}{c}
R(x,1) \\
\text{T}_1.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
R(x,2) \\
\text{T}_1.\text{maxRemote}(T_0)
\end{array} \quad \begin{array}{c}
R(x,?)
\end{array}
\]

**Explanation.**
W(x,1) and W(x,2) are unordered with respect to each other.
T_1's first read updates T_1.\text{maxRemote}(T_0) to W(x,1).
At T_1's second read, since there is no chain W(x,2) ⪯^+ W(x,*), W(x,2) is readable.
Since W(x,2) ⪯^+ W(x,1), T_1.\text{maxRemote}(T_0) is unchanged by T_1's second read.
At T_1's third read, since there is no chain W(x,1) ⪯^+ W(x,*), W(x,1) is readable.
Dark Corner Test 3  Unordered strict writes may be read in different orders by different threads.

Thread 0: strict-write(x,1)
Thread 1: strict-write(x,2)
Thread 2: read(x,1); read(x,2)
Thread 3: read(x,2); read(x,?) \in 1 is a readable value

\[
\begin{array}{c}
T_2 : \\
\quad \{ R(x,1) \} \\
\quad \{ R(x,2) \} \\
\quad T_2.\text{maxRemote}(T_0) \\
\quad \{ SW(x,1) \} \\
\quad \{ SW(x,2) \} \\
\quad T_3.\text{maxRemote}(T_1) \\
\quad \{ R(x,2) \} \\
\quad \{ R(x,?) \} \\
T_0 : \\
T_1 : \\
T_3 : \\
\end{array}
\]

Explanation.
SW(x,1) and SW(x,2) are unordered with respect to each other.
T_2’s first read updates T_2.\text{maxRemote}(T_0) to SW(x,1).
At T_2’s second read, since there is no chain SW(x,2) \prec^+ W(x,\ast), SW(x,2) is readable.
T_2’s second read updates T_2.\text{maxRemote}(T_1) to SW(x,2).
T_3’s first read updates T_3.\text{maxRemote}(T_1) to SW(x,2).
At T_3’s second read, since there is no chain SW(x,1) \prec^+ W(x,\ast), SW(x,1) is readable.
**Dark Corner Test 4** Unordered strict writes may be read in different orders by a single thread.

Thread 0: strict-write(x,1)
Thread 1: strict-write(x,2)
Thread 2: read(x,1); read(x,2); **read(x,?) \(\leftarrow 1\) is a legal value

\[
\begin{array}{c}
T_2 : \quad [R(x,1)] \quad [R(x,2)] \quad [R(x,?)] \\
\quad T_2.\text{maxRemote}(T_0) \\
\end{array}
\]

\[
\begin{array}{c}
T_0 : \quad [SW(x,1)] \\
\quad T_2.\text{maxRemote}(T_1) \\
\end{array}
\]

\[
\begin{array}{c}
T_1 : \quad [SW(x,2)] \\
\end{array}
\]

**Explanation.**
SW(x,1) and SW(x,2) are unordered with respect to each other.
T_2’s first read updates \(T_2.\text{maxRemote}(T_0)\) to SW(x,1).
At T_2’s second read, since there is no chain SW(x,2) ϒ \(+\) W(x,*), SW(x,2) is readable.
T_2’s second read updates \(T_2.\text{maxRemote}(T_1)\) to SW(x,2).
At T_2’s third read, since there is no chain SW(x,1) ϒ \(+\) W(x,*), SW(x,1) is readable.

**Dark Corner Test 5** Unordered writes, if one is read locally and the other remotely by the same thread, may be read in different orders by that thread.

Thread 0: write(x,1)
Thread 1: write(x,2); read(x,1); read(x,2); **read(x,?) \(\leftarrow 1\) is a legal value

\[
\begin{array}{c}
T_0 : \quad [W(x,1)] \\
\quad T_1.\text{maxRemote}(T_0) \\
\end{array}
\]

\[
\begin{array}{c}
T_1 : \quad [W(x,2)] \quad [R(x,1)] \quad [R(x,2)] \quad [R(x,?)] \\
\quad T_1.\text{maxLocal}(x) \\
\end{array}
\]

**Explanation.**
T_1’s write updates \(T_1.\text{maxLocal}(x)\) to W(x,2).
T_1’s first read updates \(T_1.\text{maxRemote}(T_0)\) to W(x,1), but leaves \(T_1.\text{maxLocal}(x)\) unchanged.
Since \(T_1.\text{maxLocal}(x)\) is always a legal readable value, W(x,2) is legal for this read.
T_1’s second read has no effect on \(T_1.\text{maxRemote}(T_0) = W(x,1)\).
So W(x,1) is still a legal readable value.
**Dark Corner Test 6** If ordered relaxed writes precede a fence, the writes may be read in different orders by a single thread.

Thread 0: write(x,1); write(x,2); fence; write(y,0); Thread 1: read(y,0); read(x,1); read(x,2); \textbf{read}(x,?) \leftarrow 1 \textbf{ is a legal value}

\[ T_0 : \begin{array}{c} W(x,1) \quad W(x,2) \quad F \quad W(y,0) \end{array} \]
\[ T_1 : \begin{array}{c} R(y,0) \quad R(x,1) \quad R(x,2) \end{array} \]

\textit{Explanation.}
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
F is ordered before W(y,0).
T₁'s first read updates $T₁.\text{maxRemote}(T₀)$ to W(y,0).
At T₁’s second read, since there is no chain $W(x,1) \prec^+ W(x,*)$, W(x,1) is readable.
Since $T₁.\text{maxRemote}(T₀) = W(y,0)$ and $W(y,0) \succ W(x,1)$, $T₁.\text{maxRemote}(T₀)$ is unchanged.
At T₁’s third read, since there is no chain $W(x,2) \prec^+ W(x,*)$, W(x,2) is readable.
Since $T₁.\text{maxRemote}(T₀) = W(y,0)$ and $W(y,0) \succ W(x,2)$, $T₁.\text{maxRemote}(T₀)$ is unchanged.
At T₁’s fourth read, since there is no chain $W(x,1) \prec^+ W(x,*)$, W(x,1) is readable.
Dark Corner Test 7 If ordered relaxed writes precede a notify and corresponding wait, the writes may be read in different orders by a single thread.

Thread 0: write(x,1); write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); read(x,1); read(x,2); read(x,?) \( \Leftarrow \) 1 is a legal value

\[
T_0: \quad \begin{array}{c|c|c|c}
W(x,1) & W(x,2) & \ell & \text{NOT(ℓ)}
\end{array}
\]

Explanation (Part 1).
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
\( T_0 \)'s notify updates \( T_0.phaseMax \) to F.

\[
T_0: \quad \begin{array}{c|c|c|c|c}
W(x,1) & W(x,2) & \ell & \text{NOT(ℓ)} & \text{T1.maxRemote(T0)}
\end{array}
\]

\[
T_1: \quad \begin{array}{c|c|c|c|c}
\text{NOT(ℓ)} & \text{WAIT(ℓ)} & R(x,1) & R(x,2)
\end{array}
\]

Explanation (Part 2).
\( T_1 \)'s wait updates \( T_1.maxRemote(T_0) \) to \( T_0.phaseMax \), which is F.
At \( T_1 \)'s first read, since there is no chain \( W(x,1) \prec W(x,*), W(x,1) \) is readable.
Since \( T_1.maxRemote(T_0) = F \) and \( F \succ W(x,1), T_1.maxRemote(T_0) \) is unchanged.
At \( T_1 \)'s second read, since there is no chain \( W(x,2) \prec W(x,*), W(x,2) \) is readable.
Since \( T_1.maxRemote(T_0) = F \) and \( F \succ W(x,2), T_1.maxRemote(T_0) \) is unchanged.
At \( T_1 \)'s third read, since there is still no chain \( W(x,1) \prec W(x,*), W(x,1) \) is readable.
**Dark Corner Test 8** If ordered relaxed writes precede a notify and corresponding wait, the writes may be read in either order by different threads.

Thread 0: write(x,1); write(x,2); notify(ℓ); wait(ℓ)
Thread 1: notify(ℓ); wait(ℓ); read(x,1); read(x,2)
Thread 2: notify(ℓ); wait(ℓ); read(x,2); read(x,1)

\[ T_0 \text{.phaseMax} \]

\[ T_0 : \begin{array}{c|c|c|c|c} \text{W(x,1)} & \text{W(x,2)} & \text{F} & \text{NOT(ℓ)} \\ \hline \end{array} \]

**Explanation (part 1).**
W(x,1) and W(x,2) are unordered with respect to each other.
Both W(x,1) and W(x,2) are ordered before F.
T₀’s notify updates T₀.phaseMax to F.

\[ T_1 : \begin{array}{c|c|c|c|c} \text{NOT(ℓ)} & \text{WAIT(ℓ)} & \text{R(x,1)} & \text{R(x,2)} \\ \hline \end{array} \]

\[ T_0 \text{.maxRemote(T₀)} \]

\[ T_0 : \begin{array}{c|c|c|c|c} \text{W(x,1)} & \text{W(x,2)} & \text{F} & \text{NOT(ℓ)} & \text{WAIT(ℓ)} \\ \hline \end{array} \]

\[ T_2 \text{.maxRemote(T₀)} \]

\[ T_2 : \begin{array}{c|c|c|c|c} \text{NOT(ℓ)} & \text{WAIT(ℓ)} & \text{R(x,2)} & \text{R(x,1)} \\ \hline \end{array} \]

**Explanation (part 2).**
When T₁’s wait completes, T₁.maxRemote(T₀) is updated to T₀.phaseMax, which is F.
At T₁’s first read, since there is no chain W(x,1) \( \rightarrow^+ \) W(x,*), W(x,1) is readable.
Since T₁.maxRemote(T₀) = F and F \( \succ \) W(x,1), T₁.maxRemote(T₀) is unchanged.
At T₁’s second read, since there is no chain W(x,2) \( \rightarrow^+ \) W(x,*), W(x,2) is readable.
When T₂’s wait completes, T₂.maxRemote(T₀) is updated to T₀.phaseMax, which is F.
At T₂’s first read, since there is no chain W(x,2) \( \rightarrow^+ \) W(x,*), W(x,2) is readable.
Since T₂.maxRemote(T₀) = F and F \( \succ \) W(x,2), T₂.maxRemote(T₀) is unchanged.
At T₂’s second read, since there is no chain W(x,1) \( \rightarrow^+ \) W(x,*), W(x,1) is readable.
Part 5

Performance Potential

One of the exciting features of the UPC memory model is the potential for increased performance. As discussed previously, performing a strict instruction forces the execution environment to obey more conservative consistency rules than when performing a relaxed instruction. If a programmer specifies that an instruction will be executed in the relaxed mode then the compiler has greater flexibility when performing optimizations. Before optimizing a piece of UPC code a compiler must determine that such an optimization will not violate the restrictions of the memory model. The formal definition makes it possible to decide when certain optimizations are possible and when they are not. Here several optimization techniques are examined that may potentially be used by a UPC compiler. It will be shown that blocks of code operating under the strict mode allow fewer optimizations than statements operating under the relaxed mode.

The following optimizations are presented as complete, albeit short, UPC programs. The intent is to provide compact and easy to understand examples that are as complete as necessary. Three functions are included which we do not define but should be somewhat self-explanatory, initialize, output and allocate. The first initializes necessary objects by reading from a file or some other means. The second dynamically allocates space for an array. The need for the third stems from the specification of the memory model. It stipulates that if a strict instruction does not affect the output of the program it can be treated as a relaxed instruction. Of course if a program does not produce any output then executing an instruction under the strict mode is never necessary. The presence of output ensures that strict instructions are actually executed under the strict memory mode.

In the examples discussed here only array references are shared. Additionally, all array accesses in the same program obey a single memory mode. More complex interactions between the two memory modes are certainly possible but these programs are probably representative of real world UPC code.

The next three sections are organized as follows: first examine compiler optimizations that can be applied in both modes are examined. These illustrate the fact that the extra constraints of the strict mode do not eliminate all potential optimizations. Next, optimizations that violate the strict mode but may be used under the relaxed mode are looked at. Finally, experimental results are presented regarding the potential benefits the relaxed mode has in terms of compiler optimizations. Many of the following examples have been adapted from Wolfe [33] and other resources are available [24].
shared `mode` int a[100], b[100];  \implies \quad \text{shared `mode` int a[100], b[100];}
int i, n;

int main() {
    initialize(b, &n);
    for (i = 0; i < n; i++) {
        a[i] = b[i] * 2;
    }
    output(a, n);
}

Figure 5.1: Loop Peeling

Legal Optimizations

Despite the additional requirements of the strict mode many compiler optimizations are still possible. The following is a list of legal optimizations under both strict and relaxed modes.

- Loop peeling reduces the number of times a loop is executed by “peeling” multiple statements from the loop. (Figure 5.1)
- Loop set splitting transforms loops with statements that are not executed on every iteration into multiple loops. The new loops correspond to the different iteration sets. (Figure 5.2)
- Unswitching moves a conditional statement from inside a loop to outside that loop thereby reducing the number of times the conditional statement is executed. (Figure 5.3)
- Loop unrolling reduces the total number of times a loop is executed by performing statements from multiple iterations in a single iteration. (Figure 5.4)
- Scalar expansion eliminates anti-dependence relations of scalars inside loops. This allows other transformations that are not possible when the anti-dependence relation exists. (Figure 5.5)
- Strip mining converts a single non-nested loop to a nested loop. It is a simplified version of loop tiling which transforms nested loops into more deeply nested loops. (Figure 5.6)
shared mode int a[100], b[100];        \implies \quad \text{shared mode int a[100], b[100];}
\begin{align*}
\text{int i; } \\
\text{int main()}{ } \\
\quad \text{for (i = 0; i < 100; i++)}{ } \\
\quad \quad \text{a[i] = 1;} \\
\quad \quad \text{if (i >= 25){ } } \\
\quad \quad \quad \text{b[i] = 10; } \\
\quad \text{}} \\
\quad \text{output(a, b); } \\
\text{}}
\end{align*}

int main(){
\quad \text{for (i = 0; i < 25; i++)}{ } \\
\quad \quad \text{a[i] = 1;} \\
\quad \text{}} \\
\quad \text{for (i = 25; i < 100; i++)}{ } \\
\quad \quad \text{a[i] = 1;} \\
\quad \quad \text{b[i] = 10; } \\
\quad \text{}} \\
\quad \text{output(a, b); } \\
\text{}}

\begin{align*}
\text{Figure 5.2: Loop Set Splitting}
\end{align*}

shared mode int a2[100][100];        \implies \quad \text{shared mode int a2[100][100];}
\begin{align*}
\text{int t[100]; } \\
\text{int i, j; } \\
\text{int main()}{ } \\
\quad \text{initialize(t); } \\
\quad \text{for (i = 0; i < 100; i++)}{ } \\
\quad \quad \text{for (j = 0; j < 100; j++)}{ } \\
\quad \quad \quad \text{if (t[i] > 0)}{ } \\
\quad \quad \quad \quad \text{a2[i][j] = t[i]; } \\
\quad \quad \quad \text{else}{ } \\
\quad \quad \quad \quad \text{a2[i][j] = 0; } \\
\quad \quad \text{}} \\
\quad \text{output(a2); } \\
\text{}}
\end{align*}

int main(){
\quad \text{initialize(t); } \\
\quad \text{for (i = 0; i < 100; i++)}{ } \\
\quad \quad \text{for (j = 0; j < 100; j++)}{ } \\
\quad \quad \quad \text{if (t[i] > 0)}{ } \\
\quad \quad \quad \quad \text{a2[i][j] = t[i]; } \\
\quad \quad \quad \text{else}{ } \\
\quad \quad \quad \quad \text{for (j = 0; j < 100; j++)}{ } \\
\quad \quad \quad \quad \quad \text{a2[i][j] = 0; } \\
\quad \quad \quad \text{}} \\
\quad \text{}} \\
\quad \text{output(a2); } \\
\text{}}

\begin{align*}
\text{Figure 5.3: Unswitching}
\end{align*}
Figure 5.4: Loop Unrolling

Figure 5.5: Scalar Expansion

Figure 5.6: Strip Mining
shared \textit{mode} int a[100], b[100]; \implies \textit{shared} \textit{mode} int a[100], b[100];  
shared \textit{mode} int c[100], d[100];  
int i, n;

int main()
\{
    initialize(c, d, \&n);
    a[0] = 0;
    b[0] = 0;
    for (i = 1; i < n; i++)
    \{
        a[i] = c[i-1];
        b[i] = d[i-1] + 1;
    \}
    output(a, b, n);
\}

int main()
\{
    initialize(c, \&n);
    a[0] = 0;
    b[0] = 0;
    for (i = 0; i < n; i++)
    \{
        a[i] = c[i-1];
        for (i = 0; i < n; i++)
        \{
            b[i] = d[i-1] + 1;
        \}
    \}
    output(a, b, n);
\}

\begin{figure}[h]
\begin{center}
\begin{tabular}{|c|c|}
\hline
shared \textit{mode} int a[100], b[100]; & \textit{shared} \textit{mode} int a[100], b[100]; \\
\hline
shared \textit{mode} int c[100], d[100]; & \textit{shared} \textit{mode} int c[100], d[100]; \\
int i, n; & int i, n; \\
\hline
\end{tabular}
\end{center}
\caption{Loop Fission}
\end{figure}

\textbf{Optimizations prohibited by \textit{strict}}

Under the strict memory mode a thread must execute all shared accesses in program order. Certain optimizations reorder code in such a way that program order is no longer maintained. The following optimizations are prohibited by strict but allowed under relaxed.

\begin{itemize}
\item Loop fission breaks a single loop with several statements into multiple loops. If the original loop has multiple references to different shared objects the new loops will make better use of caching. (Figure 5.7)
\item Loop fusion is the opposite of loop fission. It fuses multiple loops with the same trip count into a single loop. It can take advantage of data locality if the fused loops have multiple references to the same shared object. (Figure 5.8)
\item Loop interchange changes the order in which loops are nested. This is useful when a 2-dimension array is traversed in column major order instead of row major order. Because arrays in C and UPC are stored in row major order loop interchange will make better use of data locality and caching. (Figure 5.9)
\item Statement reordering in this example makes the array references to the same array adjacent to one another. (Figure 5.10)
\item Loop reversal transforms a loop to count in the opposite direction specified in the program. (Figure 5.11)
\end{itemize}
<table>
<thead>
<tr>
<th>shared mode int a[100], b[100], c[100];</th>
<th>=&gt;</th>
<th>shared mode int a[100], b[100], c[100];</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i, n;</td>
<td>int i, n;</td>
<td>int i, n;</td>
</tr>
<tr>
<td>int main(){</td>
<td>int main(){</td>
<td>int main(){</td>
</tr>
<tr>
<td>initialize(b, &amp;n);</td>
<td>initialize(b, &amp;n);</td>
<td>initialize(b, &amp;n);</td>
</tr>
<tr>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
</tr>
<tr>
<td>a[i] = b[i];</td>
<td>a[i] = b[i];</td>
<td>a[i] = b[i];</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
</tr>
<tr>
<td>c[i] = a[i] + 1;</td>
<td>c[i] = a[i] + 1;</td>
<td>c[i] = a[i] + 1;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
|   output(c, n); |   output(c, n); | }

Figure 5.8: Loop Fusion

<table>
<thead>
<tr>
<th>shared mode int a2[100][100];</th>
<th>=&gt;</th>
<th>shared mode int a2[100][100];</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared mode int b2[100][100];</td>
<td>shared mode int b2[100][100];</td>
<td>int i, j, n;</td>
</tr>
<tr>
<td>int i, j, n;</td>
<td>int i, j, n;</td>
<td>int i, j, n;</td>
</tr>
<tr>
<td>int main(){</td>
<td>int main(){</td>
<td>int main(){</td>
</tr>
<tr>
<td>initialize(b2, &amp;n);</td>
<td>initialize(b2, &amp;n);</td>
<td>initialize(b2, &amp;n);</td>
</tr>
<tr>
<td>for (j = 0; j &lt; n; j++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (j = 0; j &lt; n; j++) {</td>
</tr>
<tr>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (i = 0; i &lt; n; i++) {</td>
<td>for (j = 0; j &lt; n; j++) {</td>
</tr>
<tr>
<td>a2[i][j] = b2[i][j];</td>
<td>a2[i][j] = b2[i][j];</td>
<td>a2[i][j] = b2[i][j];</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
|   output(a2); |   output(a2); | }
| }
| }

Figure 5.9: Loop Interchange
shared \texttt{mode} int a[100], b[100]; \quad \Rightarrow \quad \text{shared \texttt{mode} int a[100], b[100]};

\begin{verbatim}
int main(){
    initialize(&x);
    a[0] = 0;
    b[0] = 0;
    if (x > 0) {
        a[1] = 1;
        b[1] = -1;
    } // (i = 2; i < 100; i++) {
    a[i] = a[1];
        b[i] = a[1];
        output(a, b);
    }

Figure 5.10: Statement Reordering
\end{verbatim}

\begin{verbatim}
int main(){
    initialize(b, &n);
    for (i = 0; i < n; n++) {
        a[i] = b[i] + 1;
        c[i] = a[i] + 1;
    } // (i = 0; i < n; i++) {
    for (i = 0; i < n; i++) {
        d[i] = c[i]+1 * 2;
    } output(a, c, d, n);
}

Figure 5.11: Loop Reversal
\end{verbatim}
5.1 Experimental Results

Since there are optimizations that are allowed in the relaxed mode but not in the strict mode, the question arises: *does this make a difference?* To help answer this question a series of tests were performed on three examples from the previous section, loop fission, fusion, and interchange. The test platform was a 16-node Beowulf cluster running the MuPC [25] run time system for UPC. MuPC is a publicly available implementation of UPC which runs using a combination of MPI and Pthreads. It is intended to run on a wide variety of platforms and allows users to explore the language features of UPC. Future releases of MuPC will utilize caching to improve the performance of the system.

The tests were run using a development version of MuPC which uses remote data caching.

Two programs were used for each optimization technique. The programs fission1.c, fusion1.c and interchange1.c contained the UPC code before applying the optimizations. The programs fission2.c, fusion2.c and interchange2.c contained the code after applying the optimizations.¹ Our measurements were cache misses, total remote references and execution time.

The tests were run using 1, 2, 4 and 8 threads(1). It is important to note that although multiple threads were used, only a single thread was actually doing any processing. Therefore, the timing results do not reflect the parallel runtime of the algorithm, instead they show the cost overhead of running on multiple processors. This additional overhead accounts for the observation that increasing the number of threads usually caused the execution time to increase.

One of the test parameters was the blocking factor of shared arrays. In UPC a user is able to specify how an array is laid out in shared memory. The default behavior distributes elements of the array to the threads using a round-robin method. For example, if there are 10 threads (t0, t1, \ldots, t9) and an array is declared with 100 elements (0-99), then thread t_i would get every tenth element starting at i. A user may declare an array with a blocking factor of x where the elements are again distributed among the threads, but each thread maintains blocks of x contiguous elements instead of the default of 1. At the other extreme a user may specify an indeterminate block size. In this case the compiler will set the block size equal to the size of the array divided by the number of threads. In practice there is a limit on the maximum block size and a compiler error results if it is exceeded using and indeterminate array declaration.

Two UPC array blocking schemes were used for testing. The first was the default blocking scheme with a block size of 1. The second was the indeterminate blocking scheme with one accommodation. If indeterminate blocking caused an array block size to exceed the maximum block size, then the maximum block size was used.² Typically larger block sizes meant faster execution time, and as a result larger problem sizes(N) were used when testing larger block sizes.

Loop Fission

Using loop fission and a block size of 1, there is a decrease in the number of cache misses from fission1 to fission2 with 4 and 8 threads (Table 5.1). These numbers are consistent with the intended benefit of loop fission, reducing cache misses by separating array references. Since the block size is small, when remote data are cached the array elements are not contiguous. This results in more cache misses when we reference more remote data in a single iteration of the loop. When the block size is increased we encounter interesting results (Tables 5.2). Using an indefinite blocking size and 8 threads there is no benefit of using fission. In this case each thread has a single block of contiguous array elements. When remote data are cached several contiguous elements are brought in, resulting in fewer cache misses in both cases. The situation is similar on 4 threads when each thread maintains a single block of contiguous array elements. However, when a thread maintains multiple disjoint blocks of size 2^{10}, due to the block size restriction, fission1 performs much worse. This happens when t = 4 and N = 2^{19}, as well as, t = 2 and N = \{2^{18}, 2^{19}\}. This situation is the only time when any difference in miss-ratio materializes using 2 threads. It is unclear why the number of cache-misses jumps when a thread maintains multiple disjoint blocks.

¹The source code is included in the appendix.
²In MuPC the maximum block size is 2^{16}.
<table>
<thead>
<tr>
<th>$N$</th>
<th>$t$</th>
<th>fission1 misses</th>
<th>fission2 misses</th>
<th>total references</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td>1</td>
<td>0</td>
<td>65</td>
<td>12021</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>65</td>
<td>7447</td>
<td>9923</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>16384</td>
<td>24576</td>
</tr>
<tr>
<td></td>
<td>2^{14}</td>
<td>1</td>
<td>129</td>
<td>26397</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>129</td>
<td>15703</td>
<td>22275</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>32768</td>
<td>49152</td>
</tr>
<tr>
<td></td>
<td>2^{15}</td>
<td>1</td>
<td>257</td>
<td>55149</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>257</td>
<td>32215</td>
<td>46979</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>65536</td>
<td>98304</td>
</tr>
<tr>
<td></td>
<td>2^{16}</td>
<td>1</td>
<td>513</td>
<td>112653</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>513</td>
<td>65239</td>
<td>96387</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>131072</td>
<td>196608</td>
</tr>
<tr>
<td></td>
<td>2^{17}</td>
<td>1</td>
<td>1025</td>
<td>227701</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1025</td>
<td>131303</td>
<td>195251</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>262144</td>
<td>393216</td>
</tr>
</tbody>
</table>

Table 5.1: fission cache info, block size = 1

<table>
<thead>
<tr>
<th>$N$</th>
<th>$t$</th>
<th>fission1 misses</th>
<th>fission2 misses</th>
<th>total references</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td>1</td>
<td>257</td>
<td>387</td>
<td>455</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>257</td>
<td>391</td>
<td>467</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>65536</td>
<td>98304</td>
</tr>
<tr>
<td></td>
<td>2^{16}</td>
<td>1</td>
<td>513</td>
<td>771</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>513</td>
<td>775</td>
<td>915</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>131072</td>
<td>196608</td>
</tr>
<tr>
<td></td>
<td>2^{17}</td>
<td>1</td>
<td>1025</td>
<td>1539</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1025</td>
<td>1543</td>
<td>1811</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>262144</td>
<td>262144</td>
</tr>
<tr>
<td></td>
<td>2^{18}</td>
<td>1</td>
<td>127511</td>
<td>3075</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2050</td>
<td>3079</td>
<td>3603</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>524288</td>
<td>786432</td>
</tr>
<tr>
<td></td>
<td>2^{19}</td>
<td>1</td>
<td>518879</td>
<td>382913</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4099</td>
<td>6161</td>
<td>7187</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>1048376</td>
<td>1572864</td>
</tr>
</tbody>
</table>

Table 5.2: fission cache info, block size = min\{problem size$/t$, $2^{16}$\}
<table>
<thead>
<tr>
<th>( N \downarrow )</th>
<th>( t \Rightarrow )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{13} )</td>
<td></td>
<td>0.16</td>
<td>0.20</td>
<td>5.51</td>
<td>8.89</td>
</tr>
<tr>
<td>( 2^{14} )</td>
<td></td>
<td>0.33</td>
<td>0.44</td>
<td>11.6</td>
<td>20.4</td>
</tr>
<tr>
<td>( 2^{15} )</td>
<td></td>
<td>0.65</td>
<td>0.81</td>
<td>24.3</td>
<td>43.7</td>
</tr>
<tr>
<td>( 2^{16} )</td>
<td></td>
<td>0.13</td>
<td>0.16</td>
<td>48.4</td>
<td>88.9</td>
</tr>
<tr>
<td>( 2^{17} )</td>
<td></td>
<td>0.26</td>
<td>0.32</td>
<td>98.1</td>
<td>182</td>
</tr>
</tbody>
</table>

Table 5.3: fission1 times (secs), block size = 1

<table>
<thead>
<tr>
<th>( N \downarrow )</th>
<th>( t \Rightarrow )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{13} )</td>
<td></td>
<td>0.15</td>
<td>0.02</td>
<td>1.92</td>
<td>3.24</td>
</tr>
<tr>
<td>( 2^{14} )</td>
<td></td>
<td>0.03</td>
<td>0.04</td>
<td>3.96</td>
<td>6.92</td>
</tr>
<tr>
<td>( 2^{15} )</td>
<td></td>
<td>0.64</td>
<td>0.08</td>
<td>8.32</td>
<td>14.39</td>
</tr>
<tr>
<td>( 2^{16} )</td>
<td></td>
<td>0.13</td>
<td>0.16</td>
<td>16.3</td>
<td>29.24</td>
</tr>
<tr>
<td>( 2^{17} )</td>
<td></td>
<td>0.25</td>
<td>0.32</td>
<td>32.9</td>
<td>59.8</td>
</tr>
</tbody>
</table>

Table 5.4: fission2 times (secs), block size = 1

Table 5.5: fission times, block size = 1
Table 5.6: fission1 times (secs), block size = min\{problem size/t, 2^{16}\}

<table>
<thead>
<tr>
<th>$N \downarrow$</th>
<th>$t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td>.093</td>
<td>.104</td>
<td>.251</td>
<td>.331</td>
<td></td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>.180</td>
<td>.205</td>
<td>.503</td>
<td>.671</td>
<td></td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.361</td>
<td>.416</td>
<td>.420</td>
<td>.428</td>
<td></td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>.724</td>
<td>27.9</td>
<td>2.35</td>
<td>2.36</td>
<td></td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>1.42</td>
<td>205</td>
<td>98.2</td>
<td>5.43</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: fission2 times (secs), block size = min\{problem size/t, 2^{16}\}

<table>
<thead>
<tr>
<th>$N \downarrow$</th>
<th>$t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td>.087</td>
<td>.103</td>
<td>.250</td>
<td>.336</td>
<td></td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>.167</td>
<td>.206</td>
<td>.513</td>
<td>.660</td>
<td></td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.340</td>
<td>.413</td>
<td>.419</td>
<td>.446</td>
<td></td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>.677</td>
<td>2.15</td>
<td>2.35</td>
<td>2.40</td>
<td></td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>1.35</td>
<td>5.72</td>
<td>7.03</td>
<td>5.46</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: fission times, block size = min\{problem size/t, 2^{16}\}
Loop Fusion

The loop fusion tests showed consistently fewer cache-misses using 4 and 8 threads with single element block sizes (Tables 5.9) and larger block sizes (Table 5.10). The benefit of loop fusion is reusing previously accessed array elements and thereby reducing cache misses. As with fission the instances using larger block sizes performed better than the smaller blocks. Contrary to intuition fusion2 did not produce better times than fusion1, despite the better cache-miss ratio (Tables 5.13 and 5.16). An explanation for this requires a closer examination of the runtime system. As with fission, when \( t = 2 \), all the results are the same. However, when \( N = 2^{10} \) there is another large jump in the number of cache misses.

Loop Interchange

Loop interchange displays the most dramatic results using interchange2 and a large block size. Using 4 and 8 threads the cache-miss ratio was considerably better than interchange1 (Table 5.18). Here, the elements of the arrays are laid out row wise across multiple threads. Using interchange2, when we bring remote data into cache the data layout matches the way we are iterating the matrix. Using interchange1 there is a large difference between the data layout and the column-wise iteration, resulting in a significant number of cache-misses. The results are mixed when using single element blocks, interchange2 performed better using 4 threads and larger \( N \), while interchange1 performed better using 8 threads (Table 5.17).

Observations

One observation from these results is that the optimized code resulted in a better, or about the same, cache miss ratio in most instances. This means that the optimizations can be applied without usually leading to worse performance. Further experimentation will be necessary to determine how these results generalize to other code samples. The data also seems to indicate that larger block sizes improve the cache-miss ratio in MuPC.

This data shows that the optimized code has no effect in most instances using 1 and 2 threads. The reasons for the 1 thread results are obvious. For two threads there is no benefit because we are only caching from one remote thread. If a remote thread has one block of memory, regardless of the layout, when we cache data from it that cached data will always be used. This will change if we are iterating through an array with a step size greater than 1. Less clear are the results when a thread maintains multiple disjoint blocks. A further examination of the internal caching mechanism will probably be needed to help explain this phenomenon.

When using more than two threads the the benefits of the optimizations become apparent. This makes sense, when we have multiple threads caching data from a single thread it is less beneficial since the thread doing the processing still needs to access data on other remote threads. Our optimizations make better use of cache and produce fewer cache misses.

These preliminary results are promising. They demonstrate, in several instances, that code optimizations allowed under the relaxed mode translate into better cache-miss ratios. They also show that block size is an important consideration when programming on system with high communication cost, like a Beowulf cluster. Further testing on different UPC platforms will be necessary to determine if these results are consistent in different environments. It will also be useful to test other optimization techniques, in particular loop tiling.
<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>49</td>
<td>14289</td>
<td>18157</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>49</td>
<td>6719</td>
<td>7783</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>28672</td>
<td>43008</td>
<td>50176</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>97</td>
<td>30745</td>
<td>42817</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>97</td>
<td>14951</td>
<td>20107</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>57344</td>
<td>86016</td>
<td>100352</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>193</td>
<td>63657</td>
<td>97137</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>193</td>
<td>31415</td>
<td>44755</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>114688</td>
<td>172032</td>
<td>200704</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>385</td>
<td>129481</td>
<td>190777</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>385</td>
<td>64343</td>
<td>94051</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>229376</td>
<td>344064</td>
<td>401408</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>769</td>
<td>261129</td>
<td>388057</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>769</td>
<td>130199</td>
<td>192643</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>458752</td>
<td>688128</td>
<td>802816</td>
</tr>
</tbody>
</table>

Table 5.9: fusion cache data, block size = 1

<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>193</td>
<td>664</td>
<td>768</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>193</td>
<td>291</td>
<td>343</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>114688</td>
<td>172032</td>
<td>200704</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>385</td>
<td>1336</td>
<td>1552</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>385</td>
<td>579</td>
<td>679</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>229376</td>
<td>344064</td>
<td>401408</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>769</td>
<td>2680</td>
<td>3120</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>769</td>
<td>1155</td>
<td>1351</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>458752</td>
<td>458752</td>
<td>458752</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>2049</td>
<td>5368</td>
<td>6256</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>1539</td>
<td>2307</td>
<td>2695</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>917504</td>
<td>1376252</td>
<td>1376252</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fusion1 misses</td>
<td>0</td>
<td>762702</td>
<td>10774</td>
<td>12528</td>
</tr>
<tr>
<td>fusion2 misses</td>
<td>0</td>
<td>1016898</td>
<td>4626</td>
<td>5838</td>
</tr>
<tr>
<td>total references</td>
<td>0</td>
<td>1835008</td>
<td>2752512</td>
<td>3211264</td>
</tr>
</tbody>
</table>

Table 5.10: fusion cache data, block size = \( \min\{\text{problem size}/t, 2^{16}\} \)
<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td>.030</td>
<td>.032</td>
<td>3.65</td>
<td>6.39</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>.061</td>
<td>.065</td>
<td>7.72</td>
<td>13.3</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>.117</td>
<td>.129</td>
<td>15.8</td>
<td>28.5</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>.239</td>
<td>.259</td>
<td>32.4</td>
<td>58.1</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.480</td>
<td>.516</td>
<td>64.5</td>
<td>117</td>
</tr>
</tbody>
</table>

Table 5.11: fusion1 times (secs), block size = 1

<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td>.0298</td>
<td>.0315</td>
<td>4.59</td>
<td>6.72</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>.057</td>
<td>.062</td>
<td>10.3</td>
<td>17.4</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>.118</td>
<td>.133</td>
<td>21.4</td>
<td>38.1</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>2.31</td>
<td>.253</td>
<td>44.3</td>
<td>80.3</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.459</td>
<td>.504</td>
<td>91.1</td>
<td>166</td>
</tr>
</tbody>
</table>

Table 5.12: fusion2 times (secs), block size = 1

Table 5.13: fusion times, block size = 1
Table 5.14: Fusion1 times (secs), block size = min{problem size/t, 2^{16}}

<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td>.159</td>
<td>.166</td>
<td>.422</td>
<td>.545</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>.318</td>
<td>.333</td>
<td>.845</td>
<td>1.10</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.634</td>
<td>.665</td>
<td>.661</td>
<td>.683</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>1.28</td>
<td>2.11</td>
<td>3.48</td>
<td>4.09</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>2.53</td>
<td>2.60</td>
<td>9.32</td>
<td>8.86</td>
</tr>
</tbody>
</table>

Table 5.15: Fusion2 times (secs), block size = min{problem size/t, 2^{16}}

Table 5.16: Fusion times, block size = min{problem size/t, 2^{16}}
<table>
<thead>
<tr>
<th>$N$ (\downarrow) $t$ (\Rightarrow)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>17</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>17</td>
<td>551</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>4096</td>
<td>6144</td>
</tr>
<tr>
<td>$128^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>65</td>
<td>2748</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>65</td>
<td>6743</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>16384</td>
<td>24576</td>
</tr>
<tr>
<td>$256^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>257</td>
<td>23868</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>257</td>
<td>31511</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>65356</td>
<td>98304</td>
</tr>
<tr>
<td>$512^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>1025</td>
<td>195132</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>1025</td>
<td>130583</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>262144</td>
<td>393216</td>
</tr>
<tr>
<td>$768^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>390032</td>
<td>661881</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>2306</td>
<td>327840</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>589824</td>
<td>884736</td>
</tr>
</tbody>
</table>

Table 5.17: interchange cache info, block size = 1

<table>
<thead>
<tr>
<th>$N$ (\downarrow) $t$ (\Rightarrow)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>17</td>
<td>978</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>17</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>4096</td>
<td>6144</td>
</tr>
<tr>
<td>$128^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>65</td>
<td>11052</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>65</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>16384</td>
<td>24576</td>
</tr>
<tr>
<td>$256^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>257</td>
<td>95260</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>257</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>65356</td>
<td>98304</td>
</tr>
<tr>
<td>$512^2$</td>
<td>interchange1 misses</td>
<td>0</td>
<td>1025</td>
<td>388640</td>
</tr>
<tr>
<td></td>
<td>interchange2 misses</td>
<td>0</td>
<td>1025</td>
<td>1539</td>
</tr>
<tr>
<td></td>
<td>total references</td>
<td>0</td>
<td>262144</td>
<td>393216</td>
</tr>
</tbody>
</table>

Table 5.18: interchange cache info , block size = min\{problem size/t, 2^{16}\}

46
<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64^2$</td>
<td>.00398</td>
<td>.0508</td>
<td>.105</td>
<td>.0253</td>
</tr>
<tr>
<td>$128^2$</td>
<td>.0162</td>
<td>.0205</td>
<td>.280</td>
<td>.816</td>
</tr>
<tr>
<td>$256^2$</td>
<td>.0701</td>
<td>.825</td>
<td>11.4</td>
<td>7.91</td>
</tr>
<tr>
<td>$512^2$</td>
<td>.390</td>
<td>.369</td>
<td>90.9</td>
<td>65.5</td>
</tr>
<tr>
<td>$768^2$</td>
<td>.746</td>
<td>160</td>
<td>323</td>
<td>225</td>
</tr>
</tbody>
</table>

Table 5.19: interchange1 times (secs), block size = 1

<table>
<thead>
<tr>
<th>$N \downarrow t \Rightarrow$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64^2$</td>
<td>.00416</td>
<td>.00511</td>
<td>.158</td>
<td>.0241</td>
</tr>
<tr>
<td>$128^2$</td>
<td>.0163</td>
<td>.0203</td>
<td>1.70</td>
<td>2.41</td>
</tr>
<tr>
<td>$256^2$</td>
<td>.0629</td>
<td>.0809</td>
<td>7.86</td>
<td>13.5</td>
</tr>
<tr>
<td>$512^2$</td>
<td>.255</td>
<td>.322</td>
<td>33.7</td>
<td>58.7</td>
</tr>
<tr>
<td>$768^2$</td>
<td>.579</td>
<td>2.04</td>
<td>82.9</td>
<td>133</td>
</tr>
</tbody>
</table>

Table 5.20: interchange2 times (secs), block size = 1

Table 5.21: interchange times, block size = 1
<table>
<thead>
<tr>
<th>( N )</th>
<th>( t \Rightarrow )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>64^2</td>
<td></td>
<td>.00578</td>
<td>.00650</td>
<td>.306</td>
<td>.456</td>
</tr>
<tr>
<td>128^2</td>
<td></td>
<td>.0225</td>
<td>.0260</td>
<td>4.84</td>
<td>5.82</td>
</tr>
<tr>
<td>256^2</td>
<td></td>
<td>.0946</td>
<td>.106</td>
<td>43.9</td>
<td>61.3</td>
</tr>
<tr>
<td>512^2</td>
<td></td>
<td>.408</td>
<td>.455</td>
<td>181</td>
<td>258</td>
</tr>
</tbody>
</table>

Table 5.22: interchange1 times (secs), block size = min{problem size/t, 2^{16}}

<table>
<thead>
<tr>
<th>( N )</th>
<th>( t \Rightarrow )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>64^2</td>
<td></td>
<td>.00563</td>
<td>.00659</td>
<td>.0124</td>
<td>.0125</td>
</tr>
<tr>
<td>128^2</td>
<td></td>
<td>.0225</td>
<td>.0266</td>
<td>.0600</td>
<td>.0746</td>
</tr>
<tr>
<td>256^2</td>
<td></td>
<td>.0887</td>
<td>.107</td>
<td>.248</td>
<td>.320</td>
</tr>
<tr>
<td>512^2</td>
<td></td>
<td>.351</td>
<td>.436</td>
<td>.931</td>
<td>1.27</td>
</tr>
</tbody>
</table>

Table 5.23: interchange2 times (secs), block size = min{problem size/t, 2^{16}}

Table 5.24: interchange times, block size = min{problem size/t, 2^{16}}
Part 6

Future Work

In collaboration with the work presented here a visualization tool was developed. The tool allowed users to automatically simulate sample programs composed of the low level instructions used in our semantics. Visually the tool is similar to our test cases but uses colors, instead of arcs, to show function values. Internally, the AsmL [3] language was used to simulate the UPC memory model ASM, and C# code provided the graphical interface. AsmL is a mature and feature rich ASM simulation language and runs as part of Microsoft’s .NET platform.

During the development of this thesis two noteworthy events took place. First, a new version of the UPC specification was released. The new specification resulted in only a minor change to this existing memory model. In particular, the definition of a synchronization phase was changed to the improved definition already part of our ASM.

The second and more important event was the 2003 UPC Workshop. Here a drastic change was made to the UPC memory model. It was decided that current memory model specification was inadequate in describing the author’s intent. As a result, an entirely new definition of the memory model will be drafted for future versions of the UPC spec. Any future specification will define strict operations as sequentially consistent, a radical difference from what we defined here. This change will require a new ASM formalization and new compliance tests. However, our formalization and test cases should serve as a solid basis for future work. Furthermore, our investigation of UPC performance potential will still be useful.

With the change in definition there is much future work in regard to the UPC memory model and includes the following:

1. **Define new memory model.** As evidenced by the current memory model specification, defining a new memory model will be a challenging task. The new definition must be formal enough to avoid the ambiguity of the current model but must be verbose enough for UPC novices. Envisioning a memory consistency model is one thing but conveying it to others requires precision and detail.

2. **Formalize new model.** Once the definition of the new memory model emerges it will be useful to formalize it. The formalization presented here will be very similar and serve as a solid basis. From our ASM many of the universes, functions and rules can probably be reused. Certainly the OrderNewStrict macro will need to be changed to accommodate sequential consistency. Additionally, a compiler level view of the memory model would be helpful.

3. **Develop test cases.** When sequential consistency becomes part of the memory model many of the “dark corner” tests presented here will become invalid. Our compliance tests will still be valid since the new model will be stronger than the current model. Future compliance tests will need consider the interaction between new strict mode and the relaxed mode. These will demonstrate which behaviors are possible under the new memory model.

4. **Explore performance potential.** From a practical standpoint the key outstanding issue is the performance potential of the UPC memory model. If the performance was already known
none of this work would be necessary. There is room to explore further compiler optimizations similar to those mentioned here. Programming practices and design patterns that can benefit from relaxed consistency will need to be investigated.

5. **Provide visualization tools.** One useful aspect of our formalization is its ability to be visualized easily. This led to the development of an executable tool using the AsmL language. The new memory model will require modifications to the existing tool but when should not be too cumbersome. The visualization of the current model has been instrumental for both research and presenting our work. When the new model is complete a visualization tool would be useful to help others understand the model.

The investigation and formalization in this paper forced the UPC community to reconsider the memory model. With the decision to change the memory model there is much future work to be done. Fortunately, much of the work presented here will be useful in any future efforts. In a narrow sense future work will benefit UPC novices, experienced members of the UPC community and potential UPC users. In a broader sense the future work will help foster an understanding of the difficult nature of memory consistency models.
Acknowledgments

Several people deserve credit for making this work possible. My advisor Dr. Charles Wallace has guided the development of this thesis. His patience and support has made the time working on this thesis a great learning experience. My committee members Dr. Brian Davis, Dr. Nilufer Onder and Dr. Steve Seidel have provided useful discussions and feedback on this work. The time and effort of all my committee members is greatly appreciated.

Several members of the UPC community, at Michigan Tech and abroad, have provided feedback on this work or impacted its development in one way or another. Among those not already mentioned are Dan Bonachea, Bill Carlson, Yongsheng Huang, Phil Merkey, Andy Tomaszewski, Brian Wibecan, and Zhang Zhang.

This work has been funded in part by Hewlett-Packard, and Microsoft Research provided software used in the development of the visualization tool.
Appendix A

UPC Specification

The following excerpt is taken from the UPC Spec version 1.1 [10].

5.1.2.3 Program execution

1. Unless declared objects or references are qualified as strict, there is no change to the ANSI C execution model as applied to an individual thread. This implies that translators are free to reorder and/or ignore operations (including shared operations) as long as the restrictions found in [26] are observed.

2. A further restriction applies to strict references. For each strict reference, the restrictions found in [26] must be observed with respect to all threads if that reference is eliminated (or reordered with respect to all other shared references in its thread).

3. Equally, the behavior of strict shared references can be defined as follows. Label each shared access S(i,j) or R(i,j), where S represents a strict shared access (read or write), R represents a relaxed shared access (read or write), i is the thread number making the access, j is an integer which monotonically increases as the evaluation of the program (in the abstract machine) proceeds from startup through termination. The “abstract order” is a partial ordering of all accesses by all threads such that an access x(a,b) occurs before y(c,d) in the ordering if a == c and b < d. The “actual order(k)” for thread k is another partial order in which x(a,b) occurs before y(c,d) if thread k observes the x access before it observes the y access. A thread observes all accesses present in the abstract order which effect either the data written to files by it or its input and output dynamics as described in [26]. The least requirements on a conforming implementation are that:

   • x(a,b) must “occur before” y(c,d) in actual order(e) if a == c and a == e and b < d
   • x(a,b) must “occur before” y(c,d) in actual order(e) if a == c and b < d and ((x == S) or (y == S))

UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in [26].
Appendix B

Notify/Wait Semantics

The following excerpt is taken from [10].

6.5.1 Barrier Statements

Syntax

1. \textit{synchronization}\text-sc\text-\textit{statement}:
   \begin{itemize}
   \item \texttt{upc\_notify \textit{expression\_opt} ;}
   \item \texttt{upc\_wait \textit{expression\_opt} ;}
   \item \texttt{upc\_barrier \textit{expression\_opt} ;}
   \item \texttt{upc}\texttt{\_fence ;}
   \end{itemize}

Constraints

1. \textit{expression} shall be an integer expression.

2. Each thread shall execute an alternating sequence of \texttt{upc\_notify} and \texttt{upc\_wait} statements, starting with a \texttt{upc\_notify} and ending with a \texttt{upc\_wait} statement. A synchronization phase consists of the execution of all statements between the completion of one \texttt{upc\_wait} and the start of the next.

Semantics

1. A \texttt{upc\_wait} statement completes, and the thread enters the next synchronization phase, only after all threads have completed the \texttt{upc\_notify} statement in the current synchronization phase. \texttt{upc\_wait} and \texttt{upc\_notify} are \textit{collective} operations.

2. The \texttt{upc\_fence} statement is equivalent to a null strict reference. This insures that all shared references issued before the fence are complete before any after it are issued.

3. A null strict reference is implied before a \texttt{upc\_notify} statement and after a \texttt{upc\_wait} statement.

4. The \texttt{upc\_wait} statement will generate a runtime error if the value of its expression does not equal the value of the expression by the \texttt{upc\_notify} statement for the current synchronization phase. No error will be generated if either statement does not have an expression.

5. The \texttt{upc\_wait} statement will generate a runtime error if the value of its expression differs from any expression on the \texttt{upc\_wait} and \texttt{upc\_notify} statements issued by any thread in the current synchronization phase. No error will be generated from a “difference” involving a statement for which no expression is given.

6. The \texttt{upc\_barrier} statement is equivalent to the compound statement:
   \begin{verbatim}
   \{ \texttt{upc\_notify \textit{barrier\_value} ; \texttt{upc\_wait \textit{barrier\_value} ;} \}
   \end{verbatim}

53
7. The barrier operations at thread startup and termination have a value of expression which is not in the range of user expressible values.

8. EXAMPLE 1: The following will result in a runtime error:
   
   ```c
   { upc_notify; upc_barrier; upc_wait; }
   ```
   as it is equivalent to
   
   ```c
   { upc_notify; upc_notify; upc_wait; upc_wait; }
   ```
Appendix C

Test Code

fission1.c

#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>
#define N 128
relaxed shared int a[N], b[N], c[N], d[N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, n = N, t = THREDDS;

    if (MYTHREAD == 0){
        gettimeofday(&tv1, &tz1);
        for (i = 0; i < N; i+=2){
            a[i] = c[i] * 5;
            a[i+1] = c[i+1] * 6;
            b[i] = d[i] * 8;
            b[i+1] = d[i+1] * 9;
        }
        gettimeofday(&tv2, &tz2);
        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }
        printf("N = %in\nthreads = %in", n, t); 
        printf("time = %li.%06li secs\n", sec, usec);
    }
    return 0;
}
fission2.c

#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>

#define N 128
relaxed shared int a[N], b[N], c[N], d[N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, n = N, t = THREADS;

    if (MYTHREAD == 0){
        gettimeofday(&tv1, &tz1);
        for (i = 0; i < N; i+=2){
            a[i] = c[i] * 5;
            a[i+1] = c[i+1] * 6;
        }
        for (i = 0; i < N; i+=2){
            b[i] = d[i] * 8;
            b[i+1] = d[i+1] * 9;
        }
        gettimeofday(&tv2, &tz2);
        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }
        printf("N = %i\n", n);
        printf("threads = %i\n", t);
        printf("time = %li.%06li secs\n", sec, usec);
    }
    return 0;
}
# fusion1.c

#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>

#define N 128
relaxed shared int a[N], b[N], c[N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, n = N, t = THREADS;

    if (MYTHREAD == 0){
        gettimeofday(&tv1, &tz1);
        for (i = 0; i < N; i++)
            a[i] = b[i] * 22;

        for (i = 0; i < N; i++)
            c[i] = b[i] * 33;

        for (i = 0; i < N; i++)
            b[i] = a[i] + c[i];
        gettimeofday(&tv2, &tz2);

        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }

        printf("N = %li
", n);
        printf("threads = %li
", t);
        printf("time = %li.%06li secs
", sec, usec);
    }
    return 0;
}
fusion2.c

#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>

#define N 128
relaxed shared int a[N], b[N], c[N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, n = N, t = THREARDS;

    if (MYTHREAD == 0){

        gettimeofday(&tv1, &tz1);
        for (i = 0; i < N; i++){
            a[i] = b[i] * 22;
            c[i] = b[i] * 33;
            b[i] = a[i] + c[i];
        }
        gettimeofday(&tv2, &tz2);

        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }

        printf("N = %i\n", n);
        printf("threads = %i\n", t);
        printf("time = %li.%06li secs\n", sec, usec);
    }
    return 0;
}
interchange1.c

#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>

#define N 128
relaxed shared int a2[N][N], b2[N][N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, j, n = N, t = THEMADS;

    if (MYTHREAD == 0){
        gettimeofday(&tv1, &tz1);
        for (j = 0; j < N; j++)
            for (i = 0; i < N; i++)
                a2[i][j] = b2[i][j] * 10;
        gettimeofday(&tv2, &tz2);

        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }

        printf("N = %d\n", n);
        printf("threads = %d\n", t);
        printf("time = %li.%06li secs\n", sec, usec);
    }
    return 0;
}


`interchange2.c`

```c
#include <upc.h>
#include <sys/timeb.h>
#include <sys/time.h>

#define N 128
relaxed shared int a2[N][N], b2[N][N];

int main(){
    struct timeval tv1, tv2;
    struct timezone tz1, tz2;
    long usec, sec;
    int i, j, n = N, t = THREADS;

    if (MYTHREAD == 0){
        gettimeofday(&tv1, &tz1);
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                a2[i][j] = b2[i][j] * 10;
        gettimeofday(&tv2, &tz2);

        usec = tv2.tv_usec - tv1.tv_usec;
        sec = tv2.tv_sec - tv1.tv_sec;
        if (usec < 0){
            usec += 1000000;
            sec--;
        }

        printf("N = %i\n", n);
        printf("threads = %i\n", t);
        printf("time = %li.%06li secs\n", sec, usec);
    }
    return 0;
}
```
Bibliography

   <http://www.eecs.umich.edu/gasm>.


[30] UPC mailing list: upc@hermes.gwu.edu. Archives available at [29].

