The UPC Memory Model: Problems and Prospects*

W. Kuchera C. Wallace
Michigan Technological University
Houghton, Michigan 49931 USA
{wrkucher,wallace}@mtu.edu

Abstract

The memory consistency model underlying the Unified Parallel C (UPC) language remains a promising but underused feature. We report on our efforts to understand the UPC memory model and assess its potential benefits. We describe problems we have uncovered in the current language specification. These results have inspired an effort in the UPC community to create an alternative memory model definition that avoids these problems. We give experimental results confirming the promise of performance gains afforded by the memory model's relaxed constraints on consistency.

1 Introduction

Unified Parallel C (UPC) [15] is a programming language that extends ANSI C [13] with support for parallel processing. It is a *shared-memory* language, in which different threads share data through standard read and write instructions rather than message passing.[1] UPC is emerging as a promising contender in the arena of parallel programming languages, particularly because of the degree of flexibility it allows the programmer. One programmer-controlled parameter is the level of *memory consistency*: the degree to which threads observe operations to shared data in a common order. UPC code at a relaxed level of consistency allows for greater compiler-level and architecture-level optimization, at the price of more complex program behavior. The *memory (consistency) model* underlying UPC is a contract between a program and the underlying machine architecture that constrains the order in which memory operations become visible to threads [3].

Since UPC is a relatively new language, current implementations have not exploited the potential of relaxed memory consistency to any great extent. This paper reports our results from exploring the possible benefits of this feature.

- Our efforts began by looking closely at the current memory model, as defined in the official UPC specification [4]. This brought to light some deficiencies in its definition. UPC and its current memory model are described in §2.
- The problems with the current memory model have inspired an effort in the UPC community to devise an alternative. The proposed new memory model appears in a technical report [18]. While this is still work in progress, we give some of the important features in §3.
- With a strong semantic basis afforded by the revised memory model, we were able to explore the advantages of relaxing constraints on operation order. In §4, we give some initial results demonstrating the potential performance benefits of using relaxed memory consistency.
- §5 summarizes our conclusions and outlines current and future work.

*Sponsorship provided by Hewlett-Packard Company.

[1] A processing unit in UPC is called a *thread*. A typical UPC program is composed of several threads, with each thread running on a separate processor. In this paper, "thread" always refers to a UPC thread.

2 UPC and its memory model

UPC views memory as *partitioned shared memory.* In such a scheme, memory is partitioned among all participating threads, where memory allotted to a given thread is said to have *affinity* to that thread. In UPC, a thread's partition is subdivided into a local portion and a shared portion. Data in the local portion may only be accessed by the thread to which they have affinity, while data in the shared portion are accessible by all threads. In keeping with the shared memory paradigm, threads access shared memory addresses concurrently through standard read and write instructions, rather than through explicit message passing.

2.1 Language basics

The UPC feature of interest here stems from the type qualifier `shared`. Declaring a reference `shared` means its contents will reside in shared memory, accessible to all threads. In UPC, arrays can be stored in shared memory, with elements from the same array distributed among different threads. From the programmer's perspective, updating shared memory data involves a simple C assignment statement.

2.2 Memory model

Programs running in a shared memory context are affected by the order in which memory operations are made visible to threads. By constraining the order of operations, a memory model determines which values may be returned by each read operation.

2.2.1 Memory modes

UPC introduces two *memory modes*, *strict* and *relaxed.* The mode of each shared memory access within a UPC program affects the visibility of those accesses during program execution. As the names suggest, program behavior under strict consistency is more constrained than that under relaxed consistency.

Within a program the memory mode can be toggled via mechanisms using 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, operations in a single line of code may execute in different modes. A programmer has three choices for specifying the mode of a shared access (read or write): through header files (`strict.h` or `relaxed.h`), pragmas (`#pragma upc strict` or `#pragma upc relaxed`), or type qualifiers (`strict` or `relaxed`).

Since the memory mode of a UPC program is relaxed by default, a read or write is relaxed unless otherwise specified. For example, a write to a variable `x` is strict 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 consistency effects of a strict read or strict write without actually making a memory reference, a programmer may issue a `upc_fence` statement. It is defined in the UPC specification [4] as "equivalent to a null strict reference".

2.2.2 Synchronization

UPC includes support for global synchronization. The statement `upc_barrier` causes a thread to halt execution until all other threads have executed their corresponding barriers. Once each thread has reached its barrier, all threads may resume execution. 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 reached the barrier. UPC allows programmers to use a *split-phase* barrier consisting of the paired statements `upc_notify` and `upc_wait`. In fact, `upc_barrier` is defined as a `upc_notify` immediately followed by a `upc_wait` [4]. Separating a barrier into two separate instructions gives a programmer the opportunity to do local computations after notifying but before having to wait. Barriers put special constraints on the visibility of operations, as discussed later.

2.2.3 Memory model definition

We now turn to the restrictions imposed on memory accesses by the UPC specification [4]. Two types of ordering on accesses are formulated — "abstract order" and "actual order" — each with appropriate constraints. The abstract order is a partial ordering of all accesses by all threads that preserves program order for each thread. The actual order(k) for each thread k is another partial order that represents the order in which k observes accesses. For any accesses performed by k itself, k must observe the accesses in program order. For any two accesses performed by some other thread ℓ, k must observe the accesses in program order *only if* one of the accesses is strict.

Thus, program order may be enforced on accesses by a common thread. Note, however, that no ordering is enforced on accesses by *different* threads. Thus even if all accesses are strict, the consistency level is weaker than sequential consistency [10], which establishes a total order over all accesses.

If a given constraint has no effect on either the data written into files at program termination or the ANSI C input/output requirements [13], then it need not be enforced.

2.2.4 Discussion

After examining the UPC language specification for some time, we discovered several problems in the memory model's definition [9, 8]. In some regards, the definition itself is simply inadequate and subject to multiple interpretations. Moreover, under any reasonable interpretation the official memory model is insufficient in its constraints on memory behavior, making it impractical for programmers.

The notion of "observing" is too vague. 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.

The term "observe" is sufficiently vague to lead to different interpretations. Consider an execution of the following code:[2]

T_0:	T_1:	T_2:
shared_x=1;	if (shared_x==2)	local_a=shared_y;
shared_x=2;	shared_y=3;	local_b=shared_x;

We assume that all threads execute in strict mode and shared_x and shared_y are initialized to 0. Say this code is executed, and local_a gets the value 3. Is it legal for local_b to get the value 1? Under one interpretation, the answer is "yes". T_2 observes accesses by T_0 and T_1, but there is no ordering between T_0's and T_1's accesses, so it is entirely possible to observe T_1's write to shared_y before T_0's second write to shared_x.

On the other hand, the answer "no" also seems natural: since T_2 obviously observed T_1's write, and that write was contingent on shared_x having the value 2, T_2 must also have observed T_0's second write. The conclusion would then be that T_2 *cannot* get 1 when it reads shared_x. This uses a more sophisticated, "causal" notion of observation, which seems perfectly reasonable.

Strict consistency is too relaxed. Sequential consistency [10] is the oldest and most intuitive memory model. In sequential consistency, all threads observe all memory accesses in a common order, which follows each thread's execution order. UPC programmers must have some straightforward means to ensure sequential consistency. Yet under the official memory model, there is no way to achieve this consistency level, short of enclosing all memory accesses within barriers. The problem stems from the fact that each thread's "actual order", as defined in the specification, need not include ordering of accesses by different threads. In strict mode, all threads agree on the program order of each thread t, but they need not agree on the relative order of accesses by t and some other thread t'. Thus it is impossible to establish a single order of accesses over all threads.

[2] In this and following examples, we assume that all variables are of type int, all variables beginning with shared_ refer to shared memory, and those beginning with local_ refer to local memory.

3 New memory model proposal

After discussion with UPC developers at a workshop in May 2003, we have been helping to construct a new memory model definition, which tightens the meaning of strict consistency and ties the legality of an execution directly to observed behavior. The current proposal [18] is still under discussion within the UPC community. Here we give an overview, in order to explain our experiments in §4.

3.1 Semantics

Every legal execution of a UPC program is "explainable" in terms of a (global) precedence relation and thread-local precedence relations on the (shared) accesses in the execution[3]. The global relation $<_{strict}$ linearly orders strict operations, including operations by different threads. This serves as a globally agreed-upon sequential order on strict accesses. This ensures sequential consistency for strict consistency mode. Also, for each strict access S by a given thread t, $<_{strict}$ orders S after t's preceding relaxed accesses and after t's following relaxed accesses. Thus a strict access serves to globally order certain relaxed accesses before others.

Each local relation $<_t$ is a linear order, on t's operations and all writes and strict operations, that represents thread t's view of the execution. This relation must agree with $<_{strict}$. Furthermore, it must agree with the observable behavior of the execution (the values returned by reads): each read operation must return the value written by the most recent write operation to the given location (a well-defined notion, since $<_t$ is linear).

Fence and barrier statements are defined as null strict operations. The difference between a fence and a barrier is that the latter is collective; that is, assumed to be executed by all threads. Hence a fence brings a *single* thread up to date in terms of memory consistency, while a barrier does the same for *all* threads.

[3] In this section, we shall use the term "access" to mean "shared access".

3.2 Examples

We give some examples of UPC program behavior and illustrate their legality or illegality in terms of our new definition.[4]

Example 1

relaxed shared int shared_x;

T_0:	T_1:
local_a=shared_x;	local_b=shared_x;
shared_x=0;	shared_x=1;

Observed results: local_a==1, local_b==0

This is legal behavior. An explaining local history for T_0 is the following:

$$\begin{tikzcd} RRx1 \arrow[r] & RWx0 \\ & RWx1 \arrow[ul] \end{tikzcd}$$

T_0's read $RRx1$ returns the value written by the most recent write $RWx1$.

A local history for T_1 is the following:

$$\begin{tikzcd} RRx1 & RWx0 \arrow[d] \\ & RWx1 \arrow[ul] \end{tikzcd}$$

T_1's read $RRx0$ returns the value written by the most recent write $RWx0$.

This behavior is explained in terms of two incompatible local histories: according to T_0, $RWx1 < RWx0$, while according to T_1, $RWx0 < RWx1$. This is legal under relaxed consistency, but perhaps unanticipated by the naïve programmer.

[4] In our graphical representations, accesses are represented by a consistency mode (R or S), an access type (R or W), an address identifier, and a value. Notify, wait, and barrier statements are indicated by N, W, and B, respectively. For accesses i and j, $i <_t j$ is represented by $i \rightarrow j$, and $i <_{strict} j$ is represented by $i \Rightarrow j$. For clarity, most arcs that are implied by transitivity are omitted.

Example 2

strict shared int shared_x;

T_0:	T_1:
shared_x=0;	upc_barrier;
upc_barrier;	local_b=shared_x;
local_a=shared_x;	shared_x=2;
shared_x=1;	

Observed results: local_a==2, local_b==1

Despite its similarities to the previous example, this is illegal behavior, due to the higher consistency level.. One attempt at an explaining history is the following:[5]

$$\begin{tikzcd}
SWx0 \arrow[r, Rightarrow] \arrow[dr, Rightarrow] & B \arrow[r, Rightarrow] & SRx2 \arrow[r, Rightarrow] & SWx1 \arrow[dl, Rightarrow] \\
 & B \arrow[r, Rightarrow] & SRx1 \arrow[r, Rightarrow] & SWx2 \arrow[ul, Rightarrow]
\end{tikzcd}$$

Neither T_0 nor T_1 has a linear local order to explain these results. The difference is that T_0 and T_1 must include both strict reads in their local histories. Intuitively, their histories must explain not only the local reads but also the remote, strict ones. The circularity of the proposed history makes any linear local order impossible.

Example 3

relaxed shared int shared_x,shared_y;

T_0:	T_1:
#pragma upc strict	#pragma upc strict
shared_x=0;	shared_y=0;
#pragma upc relaxed	#pragma upc relaxed
shared_x=1;	shared_y=1;
upc_notify;	upc_notify;
local_a=shared_y;	local_b=shared_x;
upc_wait;	upc_wait;

Observed results: local_a==0, local_b==0

This is legal behavior, as evidenced by the following history:

$$\begin{tikzcd}
SWx0 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & RWx1 \arrow[r, Rightarrow] & N \arrow[r, Rightarrow] \arrow[drr, Rightarrow] & RRy0 \arrow[r, Rightarrow] & W \arrow[d, Rightarrow] \\
SWy0 \arrow[r, Rightarrow] \arrow[urrr, Rightarrow] & RWy1 \arrow[r, Rightarrow] & N \arrow[r, Rightarrow] \arrow[urr, Rightarrow] & RRx0 \arrow[r, Rightarrow] & W
\end{tikzcd}$$

[5]In this and following examples, the global history is identical to each thread's local history, due to the use of strict operations.

This example illustrates the fact that a notify by itself does not enforce any memory consistency; the corresponding wait must complete to form a full barrier.

Example 4

relaxed shared int shared_x,shared_y;

T_0:	T_1:
#pragma upc strict	#pragma upc strict
shared_x=0;	shared_y=0;
#pragma upc relaxed	#pragma upc relaxed
shared_x=1;	shared_y=1;
upc_notify;	upc_notify;
upc_wait;	upc_wait;
local_a=shared_y;	local_b=shared_x;

Observed results: local_a==0, local_b==0

This is illegal behavior, since no explaining history exists. One attempt at such a history is the following:

$$\begin{tikzcd}
SWx0 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & RWx1 \arrow[r, Rightarrow] & N \arrow[r, Rightarrow] \arrow[dr, Rightarrow] & W \arrow[r, Rightarrow] \arrow[d, Rightarrow] & RRy0 \\
SWy0 \arrow[r, Rightarrow] & RWy1 \arrow[r, Rightarrow] & N \arrow[r, Rightarrow] \arrow[ur, Rightarrow] & W \arrow[r, Rightarrow] & RRx0
\end{tikzcd}$$

This explains $T0$'s read $RRy0$ but not T_1's read $RRx0$; the most recent write to x according to this history is $RWx1$. This example illustrates the consistency established by a complete notify-wait pair.

4 Performance potential

The point of relaxed consistency is the potential for increased performance. As discussed previously, if a programmer specifies that an instruction will be executed in relaxed mode, then the compiler has greater flexibility when performing optimizations. Before optimizing 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.

4.1 Legal optimizations

In this paper, we focus on three optimizations that violate strict mode but may be used under relaxed mode.[6] We present experimental results regarding the potential benefits of relaxed mode using these optimizations.

The optimizations are performed by hand, in short but complete UPC programs. Only array references are shared, and all array accesses in the same program obey a single memory mode. More complex interactions between the two memory modes are certainly possible, but the programs given here are probably representative of real world UPC code.

- *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.

- *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.

- *Loop interchange* changes the order in which loops are nested. This is useful when a two-dimensional 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.

4.2 Experimental results

Since there are optimizations that are allowed in relaxed mode but not in strict mode, the question arises: what difference does this make? 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* [12] runtime system for UPC. MuPC is a publicly available implementation of UPC based on 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 that uses remote data caching.

Two programs were used for each optimization technique. The programs fission1, fusion1, and interchange1 contain the UPC code before applying the optimizations. The programs fission2, fusion2, and interchange2 contain the code after applying the optimizations.

The tests were run using one, two, four, and eight threads. We use t to represent the number of threads. It is important to note that although multiple threads were used, only a single thread actually did any processing. Therefore, the timing results do not reflect the parallel running time 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. Thus we focus on caching results (cache misses and total remote references) rather than execution time.

One of the test parameters is 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 layout distributes elements of the array to the threads using a round-robin method. For example, if there are 10 threads (t_0, t_1, ..., t_9) and an array is declared with 100 elements (0-99), then thread t_i gets every tenth element starting at i. A user may declare an array with a blocking factor of x; in this case, 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 sets the block size equal to the ceiling of the number of elements 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 an indeterminate array declaration exceeds the limit.

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

[6] The compiler optimization examples have been adapted from Wolfe [17] and Muchnick [11]. The source code for these and other optimizations can be found in Kuchera's M.S. thesis [7].

indeterminate blocking caused an array block size to exceed the maximum block size, then the maximum block size was used.[7] Typically, larger block sizes meant faster execution time, and as a result larger problem sizes were used when testing larger block sizes. We use N to represent the problem size.

4.2.1 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 four and eight threads (§A.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 more remote data are referenced in a single iteration of the loop.

We encounter interesting results with an increased block size. With an indefinite block size and eight threads, there is no benefit to 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 with four threads when each thread maintains a single block of contiguous array elements. However, when a thread maintains multiple disjoint blocks of size 2^{16}, 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 two threads. It is unclear why the number of cache-misses jumps when a thread maintains multiple disjoint blocks.

4.2.2 Loop fusion

The loop fusion tests show consistently fewer cache-misses using four and eight threads with single element block sizes and larger block sizes (§A.2). 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 perform better than the smaller blocks. Contrary to intuition, fusion2 does not produce better times than fusion1, despite the better cache-miss ratio. 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 $N = 2^{19}$ there is another large jump in the number of cache misses.

4.2.3 Loop interchange

Loop interchange displays the most dramatic results using interchange2 and a large block size. Using four and eight threads, the cache miss ratio is much better than interchange1 (§A.3). Here, the elements of the arrays are laid out row-wise across multiple threads. Using interchange2, when remote data are brought into cache the data layout matches the pattern of iteration through 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 performs better using four threads and larger N, while interchange1 performs better using eight threads.

4.2.4 Discussion

One observation from these results is that the optimized code improved or left unchanged the cache miss ratio in most instances. This suggests that the optimizations can be applied without usually leading to worse performance. Further experimentation is necessary to extend these results to other code samples. The data also indicates that larger block sizes improve the cache miss ratio in MuPC.

The optimized code seems to have no effect in most instances using one or two threads. The reasons for the single-thread results are obvious. For two threads, there is no benefit because there is caching only from one remote thread. If a remote thread has one block of memory, regardless of the layout, the cached data will always be used. This changes if we iterate 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

[7] In MuPC the maximum block size is 2^{16}.

to help explain this phenomenon.

When using more than two threads, the benefits of the optimizations become apparent. This makes sense; when there are 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. The optimizations make better use of the cache and produce fewer cache misses.

5 Conclusion

UPC is a language still in the early stages of its development. After lengthy and energetic discussion [16], Version 1.1 of the language specification has recently appeared. Due to questions raised by this work [9, 8], the memory model definition is an issue to be actively pursued prior to the release of Version 1.2. Of the state-of-the-art implementations of UPC [2, 5, 12], none have yet made much use of the performance benefits afforded by the memory model; for instance, relaxed mode is implemented identically to strict mode. This is soon to change, however; the Hewlett-Packard [2] and MuPC [12] implementations are working to exploit relaxed mode. Thus we feel well positioned to provide useful precision and clarity to these efforts.

Defining memory models is notoriously tricky business, as witnessed not only by the UPC memory model but also by the long and deep discussion on the Java memory model [6]. The current UPC proposal still has many unanswered questions. Even a complete and precise definition is not enough, however. The programmers who are expected to use relaxed consistency must gain an intuitive feel for the memory model, and the typical terse description given in a language specification will not suffice. Multiple ways of learning about the memory model must be made available. We are currently implementing an interactive simulator, using the AsmL [1] tool, for use by UPC programmers wishing to view the results of program behavior across various platforms. This relies on an alternative but equivalent operational memory model specification. We also plan to develop a body of design patterns, in the spirit of Schmidt *et al.* [14], that will allow UPC programmers to exploit the performance benefits of relaxed consistency easily.

Our preliminary experimental 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 systems with high communication cost, such as 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.

Acknowledgments

The following people have given us useful feedback regarding UPC and its memory model: Dan Bonachea, Bill Carlson, Steve Seidel, Brian Wibecan, Kathy Yelick, and Zhang Zhang. Brian Davis, Phil Merkey, and Steve Seidel have advised us regarding the implementation of the experiments and the interpretation of their results.

References

[1] Abstract State Machine Language (AsmL): Home page. <`http://www.research.microsoft.com/foundations/AsmL`>.

[2] Compaq Information Technologies Groups. *Compaq UPC: Unified Parallel C UPC Programmer's Guide*, 2.0 edition, May 2002. Available at <`http://h30097.www3.hp.com/upc/upcuserdoc.pdf`>.

[3] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer architecture: A hardware/software approach*. Morgan Kaufmann, 1999.

[4] T. El-Ghazawi, W. Carlson, and J. Draper. UPC language specifications, v1.1. Technical report, Center for Computing Sciences, 2003. Available at [15].

[5] G. Funck. *GCC UPC Compiler for the SGI Origin 2000*. Intrepid Technology, 1.7 edition, October 2002. Available at <`http://www.intrepid.com/upc/docs/gcc-upc-for-sgi-overview.pdf`>.

[6] Java memory model mailing list. <`mailto:javamemorymodel@cs.umd.edu`>. Archives

available at <http://www.cs.umd.edu/~pugh/java/memoryModel/archive/>.

[7] W. Kuchera. Illuminating the UPC memory model. Master's thesis, Computer Science Department, Michigan Technological University, 2003. Available at <http://www.cs.mtu.edu/~wallace/pubs/kuchera-thesis.pdf>.

[8] W. Kuchera and C. Wallace. Illustrative test cases for the UPC memory model. Technical Report CS-TR-03-02, Computer Science Department, Michigan Technological University, 2003. Available at [15].

[9] W. Kuchera and C. Wallace. Toward a programmer-friendly formal specification of the UPC memory model. Technical Report CS-TR-03-01, Computer Science Department, Michigan Technological University, 2003. Available at [15].

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

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

[12] MuPC Run Time System: Home page. <http://upc.mtu.edu/MuPCdistribution>.

[13] Programming languages — C. ISO/SEC 9899, 2000.

[14] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[15] Unified Parallel C (UPC): Home page. <http://upc.gwu.edu>.

[16] UPC mailing list. <mailto:upc@hermes.gwu.edu>. Archives available at [15].

[17] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[18] K. Yelick, D. Bonachea, and C. Wallace. A proposal for a UPC memory consistency model. Technical report, Lawrence Berkeley National Laboratory, 2003. Available at <http://upc.nersc.gov/publications/>.

A Experimental results

A.1 Loop fission

N		$t=2$	$t=4$	$t=8$
2^{13}	fission1 misses	65	12021	14465
	fission2 misses	65	7447	9923
	total references	16384	24576	28672
2^{14}	fission1 misses	129	26397	35997
	fission2 misses	129	15703	22275
	total references	32768	49152	57344
2^{15}	fission1 misses	257	55149	79061
	fission2 misses	257	32215	46979
	total references	65536	98304	114688
2^{16}	fission1 misses	513	112653	165189
	fission2 misses	513	65239	96387
	total references	131072	196608	229376
2^{17}	fission1 misses	1025	227701	337565
	fission2 misses	1025	131303	195251
	total references	262144	393216	458752

Fission: cache data, block size = 1

N		2	4	8
2^{15}	fission1 misses	257	387	455
	fission2 misses	257	391	467
	total references	65536	98304	114688
2^{16}	fission1 misses	513	771	903
	fission2 misses	513	775	915
	total references	131072	196608	229367
2^{17}	fission1 misses	1025	1539	1799
	fission2 misses	1025	1543	1811
	total references	262144	262144	262144
2^{18}	fission1 misses	127511	3075	3591
	fission2 misses	2050	3079	3603
	total references	524288	786432	786432
2^{19}	fission1 misses	518879	382913	7175
	fission2 misses	4099	6161	7187
	total references	1048576	1572864	1835008

Fission: cache data,
block size = min(problem size/t, 2^{16})

A.2 Loop fusion

N		2	4	8
2^{13}	fusion1 misses	49	14289	18157
	fusion2 misses	49	6719	7783
	total references	28672	43008	50176
2^{14}	fusion1 misses	97	30745	42817
	fusion2 misses	97	14951	20107
	total references	57344	86016	100352
2^{15}	fusion1 misses	193	63657	97137
	fusion2 misses	193	31415	44755
	total references	114688	172032	200704
2^{16}	fusion1 misses	385	129481	190777
	fusion2 misses	385	64343	94051
	total references	229376	344064	401408
	fusion1 misses	769	261129	388057
	fusion2 misses	769	130199	192643
	total references	458752	688128	802816

Fusion: cache data, block size = 1

N		2	4	8
2^{15}	fusion1 misses	193	664	768
	fusion2 misses	193	291	343
	total references	114688	172032	200704
2^{16}	fusion1 misses	385	1336	1552
	fusion2 misses	385	579	679
	total references	229376	344064	401408
2^{17}	fusion1 misses	769	2680	3120
	fusion2 misses	769	1155	1351
	total references	458752	458752	458752
2^{18}	fusion1 misses	2049	5368	6256
	fusion2 misses	1539	2307	2695
	total references	917504	1376252	1376252
2^{19}	fusion1 misses	762702	10774	12528
	fusion2 misses	1016898	4626	5838
	total references	1835008	2752512	3211264

Fusion: cache data,
block size = min(problem size/t, 2^{16})

A.3 Interchange

N		2	4	8
64^2	interchange1 misses	17	252	84
	interchange2 misses	17	551	35
	total references	4096	6144	7168
128^2	interchange1 misses	65	2748	1484
	interchange2 misses	65	6743	7811
	total references	16384	24576	28672
256^2	interchange1 misses	257	23868	13580
	interchange2 misses	257	31511	44867
	total references	65536	98304	114688
512^2	interchange1 misses	1025	195132	113036
	interchange2 misses	1025	130583	193091
	total references	262144	393216	458752
768^2	interchange1 misses	390032	661881	384524
	interchange2 misses	2306	327840	440131
	total references	589824	884736	1032192

Interchange: cache data, block size = 1

N		2	4	8
64^2	interchange1 misses	17	978	476
	interchange2 misses	17	27	35
	total references	4096	6144	7168
128^2	interchange1 misses	65	11052	11452
	interchange2 misses	65	99	119
	total references	16384	24576	28672
256^2	interchange1 misses	257	95260	107596
	interchange2 misses	257	387	455
	total references	65536	98304	114688
512^2	interchange 1misses	1025	388640	449112
	interchange2 misses	1025	1539	1799
	total references	262144	393216	458752

Interchange: cache data,
block size = min(problem size/t, 2^{16})