UPC Language Specifications
V1.2 (pre5)

A publication of the UPC Consortium

This is a prerelease version of the UPC specifications
Please refer to version 1.1.1 for the most recent approved version.

September 18, 2004
Acknowledgments

Many have contributed to the ideas and concepts behind these specifications. William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, and Karen Warren are the authors of the initial UPC language concepts and specifications. Tarek El-Ghazawi, William Carlson, and Jesse Draper are the authors of the first formal version of the specifications. Because of the numerous contributions to the specifications, no explicit authors are currently mentioned. We also would like to acknowledge the role of the participants in the first UPC workshop, held in May 2000 in Bowie, Maryland, and in which the specifications of this version were discussed. In particular we would like to acknowledge the support and participation of Compaq, Cray, HP, Sun, and CSC. We would like also to acknowledge the abundant input of Kevin Harris and Sébastien Chauvin and the efforts of Lauren Smith. The efforts of Brian Wibecan and Greg Fischer were invaluable in bringing these specifications to version 1.0.

Version 1.1 is the result of the contributions of many in the UPC community, most importantly the participants in the second UPC workshop held in March 2002 in Washington, DC. In addition to the continued support of all those mentioned above, the efforts of Dan Bonachea were invaluable in this effort.

Version 1.2 is also the result of many contributors. Worthy of special note (in addition to the continued support of those mentioned above) are the substantial contributions to many aspects of the specifications by Jason Duell; Many have contributed to the ideas and concepts behind the UPC collectives specifications. Elizabeth Wiebel and David Greenberg are the authors of the first draft of this document. Steve Seidel organized the effort to refine that document into its current form. Thanks go to many in the UPC community for their interest and helpful comments, particularly Dan Bonachea, Bill Carlson, Jason Duell and Brian Wibecan. Thanks also go to Lauren Smith for her efforts to support the development of this specification. Version 1.2 also includes the UPC I/O specification which is the result of efforts by Tarek El Ghazawi, Francois Cantonnet, Proshanta Saha, Rajeef Thakur, Rob Ross, and Dan Bonachea. Finally, it also includes the substantial contributions to the UPC memory consistency model by Kathy Yelick, Dan Bonachea, and Charles Wallace.

Members of the UPC consortium may be contacted via the world wide web at http://www.upcworld.org or http://upc.gwu.edu, where an archived
mailing list may be joined. Comments on these specifications are always welcome.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>6</td>
</tr>
<tr>
<td>1 Scope</td>
<td>6</td>
</tr>
<tr>
<td>2 Normative references</td>
<td>6</td>
</tr>
<tr>
<td>3 Terms, definitions and symbols</td>
<td>7</td>
</tr>
<tr>
<td>4 Conformance</td>
<td>9</td>
</tr>
<tr>
<td>5 Environment</td>
<td>10</td>
</tr>
<tr>
<td>5.1 Conceptual models</td>
<td>10</td>
</tr>
<tr>
<td>5.1.1 Translation environment</td>
<td>10</td>
</tr>
<tr>
<td>5.1.2 Execution environment</td>
<td>10</td>
</tr>
<tr>
<td>6 Language</td>
<td>13</td>
</tr>
<tr>
<td>6.1 Notations</td>
<td>13</td>
</tr>
<tr>
<td>6.2 Predefined identifiers</td>
<td>13</td>
</tr>
<tr>
<td>6.2.1 THREADS</td>
<td>13</td>
</tr>
<tr>
<td>6.2.2 MYTHREAD</td>
<td>13</td>
</tr>
<tr>
<td>6.2.3 UPC_MAX_BLOCK_SIZE</td>
<td>14</td>
</tr>
<tr>
<td>6.3 Expressions</td>
<td>14</td>
</tr>
<tr>
<td>6.3.1 The upc.localsizeof operator</td>
<td>14</td>
</tr>
<tr>
<td>6.3.2 The upc.blocksiz eof operator</td>
<td>14</td>
</tr>
<tr>
<td>6.3.3 The upc.elemsizeof operator</td>
<td>15</td>
</tr>
<tr>
<td>6.3.4 Pointer-to-shared arithmetic</td>
<td>16</td>
</tr>
<tr>
<td>6.3.5 Cast and assignment expressions</td>
<td>17</td>
</tr>
<tr>
<td>6.3.6 Address operators</td>
<td>18</td>
</tr>
<tr>
<td>6.4 Declarations</td>
<td>18</td>
</tr>
<tr>
<td>6.4.1 Type qualifiers</td>
<td>19</td>
</tr>
<tr>
<td>6.4.2 The shared and reference type qualifiers</td>
<td>20</td>
</tr>
<tr>
<td>6.4.3 Declarators</td>
<td>22</td>
</tr>
<tr>
<td>6.5 Statements and blocks</td>
<td>25</td>
</tr>
<tr>
<td>6.5.1 Barrier statements</td>
<td>25</td>
</tr>
<tr>
<td>6.5.2 Iteration statements</td>
<td>27</td>
</tr>
<tr>
<td>6.6 Preprocessing directives</td>
<td>29</td>
</tr>
<tr>
<td>6.6.1 UPC pragmas</td>
<td>29</td>
</tr>
</tbody>
</table>
Introduction

1 UPC is a parallel extension to the C Standard. UPC follows the distributed shared-memory programming paradigm. The first version of UPC, known as version 0.9, was published in May of 1999 as technical report [CARLSON99] at the Institute for Defense Analyses Center for Computing Sciences.

2 Version 1.0 of UPC was initially discussed at the UPC workshop, held in Bowie, Maryland, 18-19 May, 2000. The workshop had about 50 participants from industry, government, and academia. This version was adopted with modifications in the UPC mini workshop meeting held during Supercomputing 2000, in November 2000, in Dallas, and finalized in February 2001.

3 Version 1.1 of UPC was initially discussed at the UPC workshop, held in Washington, DC, 3-5 March, 2002, and finalized in October 2003.

4 Version 1.2 of UPC was intially discussed at the UPC workshop held in Phoenix, AZ, 20 November 2003.

1 Scope

1 This document focuses only on the UPC specifications that extend the C Standard to an explicit parallel C based on the distributed shared memory model. All C specifications as per ISO/IEC 9899 [ISO/IEC00] are considered a part of these UPC specifications, and therefore will not be addressed in this document.

2 Small parts of the C Standard [ISO/IEC00] may be repeated for self-containment and clarity of a subsequent UPC extension definition.

2 Normative references

1 The following document and its identified normative references constitute provisions of these UPC specifications.

2 ISO/IEC 9899: 1999(E), Programming languages - C [ISO/IEC00]
3 The relationship between the section numbering used in the C Standard [ISO/IEC00] and that used in this document is given in Appendix A and noted at the beginning of each corresponding section.

3 Terms, definitions and symbols

1 For the purpose of these specifications the following definitions apply.

2 Other terms are defined where they appear in italic type or on the left hand side of a syntactical rule.

3.1 thread
an instance of execution initiated by the execution environment at program startup.

3.2 object
region of data storage in the execution environment which can represent values.

3.2.1 shared object
an object created using a shared-qualified declarator or constructed by a library function defined to create shared objects.

2 NOTE All threads may access shared objects.¹

3.2.1 private object
any object which is not a shared object.

2 NOTE Each thread declares and creates its own private objects which no other thread can access.²

3.3

¹The file scope declaration shared int x; creates a single object which any thread may access.

²The file scope declaration int y; creates a separate object for each thread to access.
1 affinity
logical association between shared objects and threads. Each element of data storage that contains shared objects has affinity to exactly one thread.

3.4
1 pointer-to-shared
a pointer whose referenced type is shared-qualified.

3.5
1 pointer-to-local
a pointer whose referenced type is not shared-qualified.

3.6
1 access
<execution-time action> to read or modify the value of an object by a thread.

3.6.1
1 shared access
an access using an expression whose type is shared-qualified.

3.6.1.1
1 strict shared read
a shared read access which is determined to be strict according to section 6.4.2 of this specification.

3.6.1.2
1 strict shared write
a shared modify access which is determined to be strict according to section 6.4.2 of this specification.

3.6.1.3
1 relaxed shared read
a shared read access which is determined to be relaxed according to section 6.4.2 of this specification.

3.6.1.4
1 relaxed shared write
a shared modify access which is determined to be relaxed according to section 6.4.2 of this specification.

3.6.2
local access
an access using an expression whose type is not shared-qualified.

3.7

collective
a constraint placed on some language operations which requires evaluation of such operations to be matched\textsuperscript{3} across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

3.8

single-valued
an operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.

3.9

phase
an unsigned integer value associated with a pointer-to-shared which indicates the element-offset within an affinity block; used in pointer-to-shared arithmetic to determine affinity boundaries.

4 Conformance

In this document, “shall” is to be interpreted as a requirement on a UPC implementation; conversely, “shall not” is to be interpreted as a prohibition.

If a “shall” or “shall not” requirement of a constraint is violated, the behavior will be undefined. Undefined behavior is indicated by “undefined behavior” or by the omission of any explicit definition of behavior from the UPC specification.

\textsuperscript{3}A collective operation need not provide any actual synchronization between threads, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely upon such unspecified synchronization for correctness.
5 Environment

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Threads environment

1 A UPC program is translated under either a “static THREADS” environment or a “dynamic THREADS” environment. Under the static THREADS environment, the number of threads to be used in execution is indicated to the translator in an implementation-defined manner. If the actual execution environment differs from this number of threads, the behavior of the program is undefined.

5.1.2 Execution environment

1 This subsection provides the UPC parallel extensions of [ISO/IEC00: Sec. 5.1.2]

2 A UPC program consists of a set of threads which may declare, create, and access both shared and private objects. Accesses to these objects are defined as either local or shared, based on the type of the access. Each thread’s local accesses behave independently and exactly as described in [ISO/IEC00]. All shared accesses behave as described herein.

3 There is an implicit upc_barrier at program startup and termination. Except as explicitly specified by upc_barrier operations or by certain library functions (all of which are explicitly documented), there are no other barrier synchronization guarantees among the threads.

Forward references: upc_barrier (6.5.1).

5.1.2.1 Program startup

1 In the execution environment of a UPC program, derived from the hosted environment as defined in the C Standard [ISO/IEC00], each thread calls the
UPC program’s main() function⁴.

5.1.2.2 Program termination

1 A program is terminated by the termination of all the threads⁵ or a call to the function upc_global_exit().

2 Thread termination follows the C Standard definition of program termination in [ISO/IEC00: Sec. 5.1.2.2.3]. A thread is terminated by reaching the } that terminates the main function, by a call to the exit function, or by a return from the initial main. Note that thread termination does not imply the completion of all I/O and that shared data allocated by a thread either statically or dynamically shall not be freed before UPC program termination.

Forward references: upc_global_exit (7.2.1).

5.1.2.3 Program execution

1 Thread execution follows the C Standard definition of program execution in [ISO/IEC00: Sec. 5.1.2.3]. This section describes the additional operational semantics users can expect from shared memory references. In a shared memory model such as UPC, operational descriptions of semantics are insufficient to completely and definitively describe a memory consistency model. Therefore Appendix B presents the formal memory semantics of UPC. The information presented in this section is consistent with the formal semantic description, but not complete. Therefore, implementations of UPC based on this section alone will be non-compliant.

2 All shared accesses are classified as being either strict or relaxed, as described in sections 6.4.2 and 6.6.1. Accesses to shared objects via pointers-to-local behave as relaxed shared accesses with respect to memory consistency. Most synchronization-related language operations and library functions (notably upc_fence, upc_notify, upc_wait, and upc_lock/upc_unlock) imply the consistency effects of a strict access.

3 In general, any sequence of purely relaxed shared accesses issued by a given thread in an execution may appear to be arbitrarily reordered relative to

⁴e.g., in the program main() { printf("hello"); } , each thread prints hello.
⁵A barrier is automatically inserted at thread termination.
program order by the implementation, and different threads need not agree upon the order in which such accesses appeared to have taken place. The only exception to the previous statement is that two relaxed accesses issued by a given thread to the same memory location where at least one is a write will always appear to all threads to have executed in program order. Consequently, relaxed shared accesses should never be used to perform deterministic inter-thread synchronization - synchronization should be performed using language/library operations whenever possible, or otherwise using only strict shared reads and strict shared writes.

4 Strict accesses always appear (to all threads) to have executed in program order with respect to other strict accesses, and in a given execution all threads observe the effects of strict accesses in a manner consistent with a single, global total order over the strict operations. Consequently, an execution of a program whose only accesses to shared objects are strict is guaranteed to behave in a sequentially consistent manner.

5 When a thread’s program order dictates a set of relaxed operations followed by a strict operation, all threads will observe the effects of the prior relaxed operations made by the issuing thread (in some order) before observing the strict operation. Similarly, when a thread’s program order dictates a strict access followed by a set of relaxed accesses, the strict access will be observed by all threads before the any subsequent relaxed accesses by the issuing thread. Consequently, strict operations can be used to synchronize the execution of different threads, and to prevent the apparent reordering of surrounding relaxed operations across a strict operation.

6 Most programs will use the strict synchronization facilities provided by the language and library (e.g. barriers, locks, etc) to synchronize threads and prevent non-determinism arising from data races. A data race may occur whenever two or more relaxed operations from different threads access the same location with no intervening strict synchronization, and at least one such access is a write. Programs which produce executions that are always free of data races (as formally defined in Appendix ??em-semantics), are guaranteed to behave in a sequentially consistent manner.
6 Language

6.1 Notations

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by *bold type*. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words “one of”. An optional symbol is indicated by the subscript “opt”, so that

\[
\{ \text{expression}_{\text{opt}} \}
\]

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

6.2 Predefined identifiers

1 This subsection provides the UPC parallel extensions of section 6.4.2.2 in [ISO/IEC00].

6.2.1 THREADS

1 THREADS is a value of type int; it specifies the number of threads and has the same value on every thread. Under the static THREADS translation environment, THREADS is an integer constant suitable for use in #if preprocessing directives.

6.2.2 MYTHREAD

1 MYTHREAD is a value of type int; it specifies the unique thread index. The range of possible values is 0..THREADS-1\(^6\).

\(^6\)e.g., the program main()
{ printf("%d ",MYTHREAD); } , prints the numbers 0 thru THREADS-1, in some order.
6.2.3 UPC_MAX_BLOCK_SIZE

1 UPC_MAX_BLOCK_SIZE is a predefined integer constant value. It indicates the maximum value\(^7\) allowed in a layout qualifier for shared data. It shall be suitable for use in \#if preprocessing directives.

6.3 Expressions

1 This subsection provides the UPC parallel extensions of section 6.5 in [ISO/IEC00].

6.3.1 The upc_localsizeof operator

\[
\text{upc_localsizeof } \text{unary-expression} \\
\text{upc_localsizeof ( type-name )}
\]

Constraints

1 The upc_localsizeof operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the sizeof operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1 The upc_localsizeof operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a shared-qualified type. It returns the same value on all threads; the value is the maximum of the size allocated to objects with affinity to any single thread. The result of upc_localsizeof is a compile-time constant.

2 The type of the result is size_t.

3 If the the operand is an expression, that expression is not evaluated.

6.3.2 The upc_blocksizeof operator

\[
\text{upc_blocksizeof } \text{unary-expression} \\
\text{upc_blocksizeof ( type-name )}
\]

\(^7\) e.g. shared [UPC_MAX_BLOCK_SIZE+1] char x[UPC_MAX_BLOCK_SIZE+1] and shared [+] char x[(UPC_MAX_BLOCK_SIZE+1)*THREADS] are compile errors.
Constraints
1 The upc_blocksizeof operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the sizeof operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics
1 The upc_blocksizeof operator returns the block size of the operand, which may be a shared object or a shared-qualified type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of upc_blocksizeof is a compile-time constant.
2 If the operator of upc_blocksizeof has indefinite block size, the value of upc_blocksizeof is 0.
3 The type of the result is size_t.
4 If the the operand is an expression, that expression is not evaluated.

Forward references: indefinite block size (6.4.2).

6.3.3 The upc_elemsizeof operator

    upc_elemsizeof unary-expression
    upc_elemsizeof ( type-name )

Constraints
1 The upc_elemsizeof operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the sizeof operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics
1 The upc_elemsizeof operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For non-array objects, upc_elemsizeof returns the same value as sizeof. The result of upc_elemsizeof is a compile-time constant.
2 The type of the result is size_t.
3 If the the operand is an expression, that expression is not evaluated.
6.3.4 Pointer-to-shared arithmetic

1 When an expression that has integer type is added to or subtracted from a pointer-to-shared, the result has the type of the pointer-to-shared operand. If the pointer-to-shared operand points to an element of a shared array object, and the shared array is large enough, the result points to an element of the shared array. If the shared array is declared with indefinite block size, the result of the pointer-to-shared arithmetic is identical to that described for normal C pointers in [ISO/IEC00 sec. 6.5.6], except that the thread of the new pointer shall be the same as that of the original pointer and the phase component is defined to always be zero. If the shared array has a definite block size, then the following example describes pointer arithmetic:

```c
shared [B] int *p, *p1; /* B a positive integer */
int i;

p1 = p + i;
```

2 After this assignment the following equations must hold in any UPC implementation. In each case the `div` operator indicates integer division rounding towards negative infinity and the `mod` operator returns the nonnegative remainder:\(^8\)

```c
upc_phaseof(p1) == (upc_phaseof(p) + i) mod B
upc_threadof(p1) == (upc_threadof(p) + (upc_phaseof(p) + i) div B) mod THREADS
```

3 In addition, the correspondence between shared and local addresses and arithmetic is defined using the following constructs:

```c
T *P1, *P2;
shared T *S1, *S2;

P1 = (T*) S1; /* legal if S1 has affinity to MYTHREAD */
P2 = (T*) S2; /* legal if S2 has affinity to MYTHREAD */
```

\(^8\)The C "\%" and "/" operators do not have the necessary properties
4 For all S1 and S2 that point to two distinct elements of the same shared array object which have affinity to the same thread:

- S1 and P1 shall point to the same object.
- S2 and P2 shall point to the same object.
- The expression \((\text{upc_addrfield}(S2) - \text{upc_addrfield}(S1))\) shall evaluate to the same value as \(((P2 - P1) \times \text{sizeof}(T))\).
- If \(S1 < S2\) then \(\text{upc_addrfield}(S1)\) shall be \(<\) \(\text{upc_addrfield}(S2)\) otherwise \(\text{upc_addrfield}(S1)\) shall be \(>\) \(\text{upc_addrfield}(S2)\)

5 Two compatible pointers-to-shared which point to the same object (i.e. having the same address and thread components) shall compare as equal according to \(==\) and \(!=\), regardless of whether the phase components match.

Forward references: \texttt{upc_threadof (7.2.3.1)}, \texttt{upc_phaseof (7.2.3.2)}, \texttt{upc_addrfield (7.2.3.4)}.

6.3.5 Cast and assignment expressions

Constraints

1 A shared type qualifier shall not appear in a type cast where the corresponding pointer component of the type of the expression being cast is not shared-qualified.\(^9\) An exception is made when the constant expression 0 is cast, the result is called the \textit{null pointer-to-shared}.\(^{10} \)\(^{11}\)

Semantics

1 The casting or assignment from one pointer-to-shared to another in which either the type size or block size differs results in a pointer with a zero phase, unless one of the types is a qualified or unqualified version of \texttt{shared void\*}, the \textit{generic pointer-to-shared}.

\(^9\)
\(^{i.e.,\ pointers\-to\-local\ cannot\ be\ cast\ to\ pointers\-to\-shared.}\n
\(^{10}\)\(^{[ISO/IEC00]}\ \text{sec 6.3.2.3 and 6.5.16.1 imply that an implicit cast is allowed for zero and that all null pointers-to-shared compare equal.}\n
\(^{11}\)\(^{The\ operators\ \texttt{upc_threadof}\ and\ \texttt{upc_phaseof}\ evaluate\ to\ zero\ for\ all\ null\ pointers\-to\-shared.}\n
17
2 If a generic pointer-to-shared is cast to a non-generic pointer-to-shared type with indefinite block size or with block size of one, the result is a pointer with a phase of zero. Otherwise, if the phase of the former pointer value is not within the range of possible phases of the latter pointer type, the result is undefined.

3 If a non-null pointer-to-shared is cast\(^\text{12}\) to a pointer-to-local\(^\text{13}\) and the affinity of the pointed-to shared object is not to the current thread, the result is undefined.

4 If a null pointer-to-shared is cast to a pointer-to-local, the result is a null pointer.

5 Shared objects with affinity to a given thread can be accessed by either pointers-to-shared or pointers-to-local of that thread.

6 EXAMPLE 1:

```c
int i, *p;
shared int *q;
q = (shared int *)p;  /* is not allowed */
if (upc_threadof(q) == MYTHREAD)
    p = (int *) q;  /* is allowed */
```

### 6.3.6 Address operators

**Semantics**

1 When the unary `&` is applied to a shared structure element of type `T`, the result has type `shared [] T *`.

### 6.4 Declarations

1 UPC extends the declaration ability of C to allow shared types, shared data layout across threads, and ordering constraint specifications.

**Constraints**

\(^{12}\)As such pointers are not type compatible, explicit casts are required.

\(^{13}\)Accesses through such cast pointers are local accesses and behave accordingly.
1 The declaration specifiers in a given declaration shall not include, either
directly or through one or more typedefs, both \textbf{strict} and \textbf{relaxed}.

2 The declaration specifiers in a given declaration shall not specify more than
one block size, either directly or indirectly through one or more typedefs.

\textbf{Syntax}

1 The following is the declaration definition as per [ISO/IEC00] section 6.7, re-
peated here for self-containment and clarity of the subsequent UPC extension
specifications.

2 \texttt{declaration:}
   \hspace{1em} \texttt{declaration-specifiers init-declarator-list_{\mathrm{opt}}} ;

3 \texttt{declaration-specifiers:}
   \hspace{1em} \texttt{storage-class-specifier declaration-specifiers_{\mathrm{opt}}}
   \hspace{1em} \texttt{type-specifier declaration-specifiers_{\mathrm{opt}}}
   \hspace{1em} \texttt{type-qualifier declaration-specifiers_{\mathrm{opt}}}
   \hspace{1em} \texttt{function-specifier declaration-specifiers_{\mathrm{opt}}}

4 \texttt{init-declarator-list:}
   \hspace{1em} \texttt{init-declarator}
   \hspace{1em} \texttt{init-declarator-list \texttt{, init-declarator}}

5 \texttt{init-declarator:}
   \hspace{1em} \texttt{declarator}
   \hspace{1em} \texttt{declarator = initializer}

\textbf{Forward references:} strict and relaxed type qualifiers (6.4.2).

6.4.1 Type qualifiers

1 This subsection provides the UPC parallel extensions of section 6.7.3 in
[ISO/IEC00].

\textbf{Syntax}

1 \texttt{type-qualifier:}
   \hspace{1em} \texttt{const}
6.4.2 The shared and reference type qualifiers

Syntax
1 shared-type-qualifier:
   shared layout-qualifier<opt

2 reference-type-qualifier:
   relaxed
   strict

3 layout-qualifier:
   [constant-expression<opt]
   [ * ]

Constraints
1 A reference type qualifier shall appear in a qualifier list only when the list also contains a shared type qualifier.

2 A shared type qualifier can appear anywhere a type qualifier can appear except that it shall not appear in the specifier qualifier list of a structure declaration unless it qualifies a pointer type.

3 A layout qualifier of [*] shall not appear in the declaration specifiers of a pointer.

4 A layout qualifier shall not appear in the type qualifiers for a pointer to void type.

Semantics
1 Shared accesses shall be either strict or relaxed. Strict and relaxed shared accesses behave as described in section 5.1.2.3 of this document.

2 An access shall be determined to be strict or relaxed as follows. If the referenced type is strict-qualified or relaxed-qualified, the access shall be
strict or relaxed, respectively. Otherwise the access shall be determined to be strict or relaxed by the UPC pragma rules, as described in section 6.6.1 of this document.

3 The layout qualifier dictates the blocking factor for the type being shared qualified. This factor is the nonnegative number of consecutive elements (when evaluating pointer-to-shared arithmetic and array declarations) which have affinity to the same thread. If the optional constant expression is 0 or is not specified, all objects have affinity to the same thread. If there is no layout qualifier, the blocking factor has the default value (1). The blocking factor is also referred to as the block size.

4 A layout qualifier indicating that all array elements have affinity to the same thread is said to specify indefinite block size.

5 The block size is a part of the type compatibility.\textsuperscript{14}

6 The generic pointer-to-shared is assignment compatible with any pointer-to-shared type, regardless of block size.

7 If the layout qualifier is of the form \`\*[\*]\`}, the shared object is distributed as if it had a block size of

\[
(\text{sizeof}(a) / \text{upc_elemsizeof}(a) + \text{THREADS} - 1) / \text{THREADS},
\]

where \`a\` is the array being distributed.

8 \textbf{EXAMPLE 1:} declaration of a shared scalar

\begin{verbatim}
strict shared int y;
\end{verbatim}

\textit{strict shared} is the type qualifier.

9 \textbf{EXAMPLE 2:} automatic storage duration

\begin{verbatim}
void foo (void) {
    shared int x; /* a shared automatic variable is not allowed */
    shared int* y; /* a pointer-to-shared is allowed */
    int * shared z; /* a shared automatic variable is not allowed*/
    ... 
}
\end{verbatim}

\textsuperscript{14}This is a powerful statement which allows, for example, that in an implementation \texttt{sizeof(shared int *)} may differ from \texttt{sizeof (shared [10] int *)} and if \(T\) and \(S\) are pointer-to-shared types with different block sizes, then \(T\) and \(S\) cannot be aliases.
EXAMPLE 3: inside a structure

```c
struct foo {
    shared int x; /* this is not allowed */
    shared int* y; /* a pointer-to-shared is allowed */
};
```

Forward references: shared array (6.4.3.1)

### 6.4.3 Declarators

**Syntax**

1. The following is the declarator definition as per [ISO/IEC00] section 6.7.5, repeated here for self-containment and clarity of the subsequent UPC extension specifications.

2. `declarator:`
   - `pointer_opt direct-declarator`

3. `direct-declarator:`
   - `identifier`
   - `( declarator )`
   - `direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]`
   - `direct-declarator [ static type-qualifier-list_opt assignment-expression ]`
   - `direct-declarator [ type-qualifier-list static assignment-expression ]`
   - `direct-declarator [ type-qualifier-list_opt * ]`
   - `direct-declarator ( parameter-type-list )`
   - `direct-declarator ( identifier-list_opt )`

4. `pointer:`
   - `* type-qualifier-list_opt`
   - `* type-qualifier-list_opt pointer`

5. `type-qualifier-list:`
   - `type-qualifier`
Constraints

1 No type qualifier list shall specify more than one block size, either directly or indirectly through one or more typedefs.\textsuperscript{15}

2 No type qualifier list shall include both \texttt{strict} and \texttt{relaxed} either directly or indirectly through one or more typedefs.

3 \texttt{shared} shall not appear in a declarator which has automatic storage duration, unless it qualifies a pointer type.

Semantics

1 All static non-array shared objects have affinity with thread zero.

2 Only pointer type members of a structure or union may be shared-qualified.\textsuperscript{16}

6.4.3.1 Array declarators

1 This subsection provides the UPC parallel extensions of section 6.7.5.2 in [ISO/IEC00].

Constraints

1 When a UPC program is translated in the “dynamic THREADS” environment and the type of the array is shared-qualified but not indefinite layout-qualified, the THREADS lvalue shall occur exactly once in one dimension of the array declarator (including through typedefs). Further, in such cases, the THREADS lvalue shall only occur either alone or when multiplied by a constant expression.

Semantics

1 Elements of shared arrays are distributed in a round robin fashion, by chunks of block-size elements, such that the i-th element has affinity with thread (floor (i/block\_size) mod THREADS).

2 In an array declaration, the type qualifier applies to the elements.

\textsuperscript{15}While layout qualifiers are most often seen in array or pointer declarators, they are legal in all declarators. For example, \texttt{shared [3] int y} is a legal declarator.

\textsuperscript{16}E.g., \texttt{struct S1 \{ shared char * p1; \}} is legal, while \texttt{struct S2 \{ char * shared p2; \}} is not.
For any shared array, $a$, `upc_phaseof (&a)` is zero.

EXAMPLE 1: declarations legal in either static or dynamic translation environments:

```bash
shared int x [10*THREADS];
shared [] int x [10];
```

EXAMPLE 2: declarations legal only in static translation environment:

```bash
shared int x [10+THREADS];
shared [] int x [THREADS];
shared int x [10];
```

EXAMPLE 3: declaration of a shared array

```bash
shared [3] int x [10];
```

`shared [3]` is the type qualifier of an array, $x$, of 10 integers. `[3]` is the layout qualifier.

EXAMPLE 4:

```bash
typedef int S[10];
shared [3] S T[3*THREADS];
```

`shared [3]` applies to the underlying type of $T$, which is `int`, regardless of the typedef. The array is blocked as if it were declared:

```bash
shared [3] int T[3*THREADS][10];
```

EXAMPLE 5:

```bash
shared [] double D[100];
```

All elements of the array $D$ have affinity to thread 0. No `THREADS` dimension is allowed in the declaration of $D$.

```bash
shared [] long *p;
x = p[i];
```

All elements accessed by subscripting or otherwise dereferencing $p$ have affinity to the same thread. That thread is determined by the assignment which sets $p$. 
6.5 Statements and blocks

This subsection provides the UPC parallel extensions of section 6.8 in [ISO/IEC00].

Syntax

statement:
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  synchronization-statement

6.5.1 Barrier statements

Syntax

synchronization-statement:
  upc_notify expressionopt ;
  upc_wait expressionopt ;
  upc_barrier expressionopt ;
  upc_fence ;

Constraints

expression shall be an integer expression.

Each thread shall execute an alternating sequence of upc_notify and upc_wait statements, starting with a upc_notify and ending with a upc_wait statement. After a thread executes upc_notify the next collective operation it executes must be a upc_wait. A synchronization phase consists of the execution of all statements between the completion of one upc_wait and the start of the next.

This effectively prohibits issuing any collective operations between a upc_notify and a upc_wait.
Semantics

1 A upc_wait statement completes, and the thread enters the next synchronization phase, only after all threads have completed the upc_notify statement in the current synchronization phase.18 upc_wait and upc_notify are collective operations.

2 The upc_fence statement is equivalent to a null strict access. This insures that all shared accesses issued before the fence are complete before any after it are issued.19

4 A null strict access is implied before a upc_notify statement and after a upc_wait statement.20

5 The upc_wait statement will generate a runtime error if the value of its expression does not equal the value of the expression by the upc_notify statement for the current synchronization phase. No error will be generated if either statement does not have an expression.

6 The upc_wait statement will generate a runtime error if the value of its expression differs from any expression on the upc_wait and 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.

7 The upc_barrier statement is equivalent to the compound statement21:

{ upc_notify barrier_value; upc_wait barrier_value; }

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

9 EXAMPLE 1: The following will result in a runtime error:

---

18Therefore, all threads are entering the same synchronization phase as they complete the upc_wait statement.

19One implementation of upc_fence may be achieved by a null strict access: { static shared strict int x; x = x; }

20This implies that shared accesses executed after the upc_notify and before the upc_wait may occur in either the synchronization phase containing the upc_notify or the next on different threads.

21This equivalence is explicit with respect to matching expressions in semantic 6 and collective status in semantic 1.
\{ \text{upc\_notify}; \text{upc\_barrier}; \text{upc\_wait}; \}\}

as it is equivalent to

\{ \text{upc\_notify}; \text{upc\_notify}; \text{upc\_wait}; \text{upc\_wait}; \}\}

6.5.2 Iteration statements

This subsection provides the UPC parallel extensions of section 6.8.5 in [ISO/IEC00].

Syntax

1 \textit{iteration-statement:}

\begin{align*}
\text{while ( expression ) statement} \\
\text{do statement while ( expression ) ;} \\
\text{for ( expression\textsubscript{opt}; expression\textsubscript{opt}; expression\textsubscript{opt}) statement} \\
\text{for ( declaration-expression\textsubscript{opt}; expression\textsubscript{opt}) statement} \\
\text{upc\_forall ( expression\textsubscript{opt}; expression\textsubscript{opt}; expression\textsubscript{opt}; expression\textsubscript{opt}; affinity\textsubscript{opt}) statement}
\end{align*}

2 \textit{affinity:}

\begin{align*}
\text{expression\textsubscript{opt}} \\
\text{continue}
\end{align*}

Constraints:

1 The \textit{expression} for affinity shall have pointer-to-shared type or integer type.

Semantics:

1 \texttt{upc\_forall} is a \textit{collective} operation in which, for each execution of the loop body, the controlling expression and affinity expression are \textit{single-valued}.\textsuperscript{22}

2 The \textit{affinity} field specifies the executions of the loop body which are to be performed by a thread.

3 When \textit{affinity} is of pointer-to-shared type, the loop body of the \texttt{upc\_forall} statement is executed for each iteration in which the value of \texttt{MYTHREAD} equals

\textsuperscript{22}Note that single-valued implies that all thread agree on the total number of iterations, their sequence, and which threads execute each iteration.
the value of upc_threadof(affinity). Each iteration of the loop body is executed by precisely one thread.

4 When affinity is an integer expression, the loop body of the upc_forall statement is executed for each iteration in which the value of MYTHREAD equals the value affinity mod THREADS.

5 When affinity is continue or not specified, each loop body of the upc_forall statement is performed by every thread and semantic 1 does not apply.

6 If the loop body of a upc_forall statement contains one or more upc_forall statements, either directly or through one or more function calls, the construct is called a “nested upc_forall” statement. In a “nested upc_forall”, the outermost upc_forall statement that has an affinity expression which is not continue is called the “controlling upc_forall” statement. All upc_forall statements which are not “controlling” in a “nested upc_forall” behave as if their affinity expressions were continue.

7 If the execution of any loop body of a upc_forall statement produces a side-effect which affects the execution of another loop body of the same upc_forall statement which is executed by a different thread, the behavior is undefined.

8 If any thread terminates or executes a upc_barrier, upc_notify, or upc_wait statement within the dynamic scope of a upc_forall statement, the result is undefined. If any thread terminates a upc_forall statement using a break, goto, or return statement, or the longjmp function, the result is undefined. If any thread enters the body of a upc_forall statement using a goto statement, the result is undefined.

9 EXAMPLE 1: Nested upc_forall:

```c
main () {
    int i, j, k;
    shared float *a, *b, *c;

    upc_forall(i=0; i<N; i++; continue)
```

---

23 This semantic implies that side effects on the same thread have defined behavior, just like in the for statement.

24 The continue statement behaves as defined in [ISO/IEC00; Section 6.8.6.2]; equivalent to a goto the end of the loop body.
upc_forall(j=0; j<N; j++; &a[j])
upc_forall (k=0; k<N; k++; &b[k])
a[j] = b[k] * c[i];
}

This example executes all iterations of the “i” and “k” loops on every thread, and executes iterations of the “j” loop on those threads where upc_threadof (&a[j]) equals the value of MYTHREAD.

6.6 Preprocessing directives

1 This subsection provides the UPC parallel extensions of section 6.10 in [ISO/IEC00].

6.6.1 UPC pragmas

Semantics

1 If the preprocessing token upc immediately follows the pragma, then no macro replacement is performed and the directive shall have one of the following forms:

#pragma upc strict

#pragma upc relaxed

2 These pragmas affect the strict or relaxed categorization of shared accesses where the referenced type is neither strict-qualified nor relaxed-qualified. Such accesses shall be strict if a strict pragma is in effect, or relaxed if a relaxed pragma is in effect.

3 Shared accesses which are not categorized by either referenced type or by these pragmas behave in an implementation defined manner in which either all such accesses are strict or all are relaxed. Users wishing portable programs are strongly encouraged to categorize all shared accesses either by using type qualifiers, these directives, or by including upc_strict.h or upc_replaced.h.
4 The pragmas shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When they are outside external declarations, they apply until another such pragma or the end of the translation unit. When inside a compound statement, they apply until the end of the compound statement; at the end of the compound statement the state of the pragmas is restored to that preceding the compound statement. If these pragmas are used in any other context, their behavior is undefined.

6.6.2 Predefined macro names

1 The following macro name shall be defined by the implementation:

  _UPC_ The integer constant 1, indicating a conforming implementation.
  _UPC_VERSION_ The integer constant 200409L.

2 The following macro names are conditionally defined by the implementation:

  _UPC_DYNAMIC_THREADS_ The integer constant 1 in the dynamic THREADS translation environment, otherwise undefined.
  _UPC_STATIC_THREADS_ The integer constant 1 in the static THREADS translation environment, otherwise undefined.

7 Library

7.1 Standard headers

1 This subsection provides the UPC parallel extensions of section 7.1.2 in [ISO/IEC00].

2 The standard headers are

    <upc_strict.h>
    <upc_relaxed.h>
    <upc.h>
7.2 UPC utilities <upc.h>

This subsection provides the UPC parallel extensions of section 7.20 in [ISO/IEC00]. All of the characteristics of library functions described in section 7.1.4 in [ISO/IEC00] apply to these as well.

7.2.1 Termination of all threads

Synopsis

```c
#include <upc.h>

void upc_global_exit(int status);
```

Description

`upc_global_exit()` flushes all I/O, releases all storage, and terminates the execution for all active threads.

7.2.2 Shared memory allocation functions

The UPC memory allocation functions return, if successful, a pointer-to-shared which is suitably aligned so that it may be assigned to a pointer-to-shared of any type. The pointer has zero phase and points to the start of the allocated space. If the space cannot be allocated, a null pointer-to-shared is returned.
7.2.2.1 The upc_global_alloc function

Synopsis
1  #include <upc.h>

    shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
    nblocks : number of blocks
    nbytes : block size

Description
1  The upc_global_alloc allocates shared space compatible with the declaration:

    shared [nbytes] char[nblocks * nbytes].

2  The upc_global_alloc function is not a collective function. If called by multiple threads, all threads which make the call get different allocations. If nblocks*nbytes is zero, the result is a null pointer-to-shared.

7.2.2.2 The upc_all_alloc function

Synopsis
1  #include <upc.h>

    shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
    nblocks : number of blocks
    nbytes : block size

Description
1  upc_all_alloc is a collective function with single-valued arguments.

2  The upc_all_alloc function allocates shared space compatible with the following declaration:

    shared [nbytes] char[nblocks * nbytes].
3 The upc_all_alloc function returns the same pointer value on all threads. If \texttt{nblocks*nbytes} is zero, the result is a null pointer-to-shared.

4 The dynamic lifetime of an allocated object extends from the time any thread completes the call to \texttt{upc_all_alloc} until any thread has deallocated the object.

\textbf{7.2.2.3 The \texttt{upc_alloc} function}

\textbf{Synopsis}

1 \begin{verbatim}
#include <upc.h>

shared void *upc_alloc(size_t nbytes);
\end{verbatim}

\texttt{nbytes} : total number of bytes to allocate

\textbf{Description}

1 The \texttt{upc_alloc} function allocates shared space of at least \texttt{nbytes} bytes with affinity to the calling thread.

2 \texttt{upc_alloc} is similar to malloc() except that it returns a pointer-to-shared value. It is not a \textit{collective} function. If \texttt{nbytes} is zero, the result is a null pointer-to-shared.

\textbf{7.2.2.4 The \texttt{upc_local_alloc} function \textit{deprecated}}

\textbf{Synopsis}

1 \begin{verbatim}
#include <upc.h>

shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
\end{verbatim}

\texttt{nblocks} : number of blocks

\texttt{nbytes} : block size

\textbf{Description}

1 The \texttt{upc_local_alloc} function is deprecated and should not be used. UPC programs should use the \texttt{upc_alloc} function instead. Support may be removed in future versions of this specification.
The upc_local_alloc function allocates shared space of at least \( nblocks \times nbytes \) bytes with affinity to the calling thread. If \( nblocks \times nbytes \) is zero, the result is a null pointer-to-shared.

upc_local_alloc is similar to malloc() except that it returns a pointer-to-shared value. It is not a collective function.

### 7.2.2.5 The upc_free function

**Synopsis**

```c
#include <upc.h>

void upc_free(shared void *ptr);
```

**Description**

The upc_free function frees the dynamically allocated shared storage pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the upc_alloc, upc_global_alloc, upc_all_alloc, or upc_local_alloc, function, or if the space has been deallocated by a previous call, by any thread,\(^{25}\) to upc_free, the behavior is undefined.

### 7.2.3 Pointer-to-shared manipulation functions

#### 7.2.3.1 The upc_threadof function

**Synopsis**

```c
#include <upc.h>

size_t upc_threadof(shared void *ptr);
```

**Description**

The upc_threadof function returns the index of the thread that has affinity to the shared object pointed to by `ptr`.

\(^{25}\)i.e., only one thread may call upc_free for each allocation
7.2.3.2 The upc_phaseof function

Synopsis
1 #include <upc.h>

    size_t upc_phaseof(shared void *ptr);

Description
1 The upc_phaseof function returns the phase component of the pointer-to-shared argument.

7.2.3.3 The upc_resetphase function

Synopsis
1 #include <upc.h>

    shared void *upc_resetphase(shared void *ptr);

Description
1 The upc_resetphase function returns a pointer-to-shared which is identical to its input except that it has zero phase.

7.2.3.4 The upc_addrfield function

Synopsis
1 #include <upc.h>

    size_t upc_addrfield(shared void *ptr);

Description
1 The upc_addrfield function returns an implementation-defined value reflecting the “local address” of the object pointed to by the pointer-to-shared argument.
7.2.3.5 The upc_affinitysize function

Synopsis

```c
#include <upc.h>

size_t upc_affinitysize(size_t totalsize, size_t nbytes, size_t threadid);
```

totalsize: the total size of the allocation in bytes
nbytes: the number of bytes in a block
threadid: the thread whose affinitysize is to be evaluated

Description

1 upc_affinitysize is a convenience function which calculates the exact size of the local portion of the data in a shared object with affinity to threadid.

2 In the case of a dynamically allocated shared object, the totalsize argument shall be nbytes*nblocks and the nbytes argument shall be nbytes, where nblocks and nbytes are exactly as passed to upc_global_alloc or upc_all_alloc when the object was allocated.

3 In the case of a statically allocated shared object with declaration:

```c
shared [b] t d[s];
```

the totalsize argument shall be s * sizeof (t) and the nbytes argument shall be b * sizeof (t). If block size is unspecified, nbytes shall be 1. If the block size is indefinite, nbytes shall be 0.

4 threadid shall be a value in 0..(THREADS-1).

7.2.4 Lock functions

7.2.4.1 Type

1 The type declared is

```c
upc_lock_t
```

2 The type upc_lock_t is an opaque UPC type. upc_lock_t is a shared datatype with incomplete type (as defined in section 6.2.5 of [ISO/IEC00]). Objects of type upc_lock_t may therefore only be manipulated through pointers.
7.2.4.2 The upc_global_lock_alloc function

Synopsis
1 #include <upc.h>

   upc_lock_t *upc_global_lock_alloc(void);

Description
1 The upc_global_lock_alloc function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.
2 The upc_global_lock_alloc function is not a collective function. If called by multiple threads, all threads which make the call get different allocations.

7.2.4.3 The upc_all_lock_alloc function

Synopsis
1 #include <upc.h>

   upc_lock_t *upc_all_lock_alloc(void);

Description
1 The upc_all_lock_alloc function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.
2 The upc_all_lock_alloc is a collective function. The return value on every thread points to the same lock object.

7.2.4.4 The upc_lock_free function

Synopsis
1 #include <upc.h>

   void upc_lock_free(upc_lock_t *ptr);
Description
1 The upc_lock_free function frees all resources associated with the dynamically allocated upc_lock_t pointed to by ptr. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the upc_global_lock_alloc or upc_all_lock_alloc function, or if the lock has been deallocated by a previous call to upc_lock_free, the behavior is undefined.

2 upc_lock_free succeeds regardless of whether the referenced lock is currently unlocked or currently locked (by any thread).

3 Any subsequent calls to locking functions from any threads using ptr have undefined effects.

7.2.4.5 The upc_lock function

Synopsis
1 #include <upc.h>

   void upc_lock(upc_lock_t *ptr);

Description
1 The upc_lock function locks a shared variable, of type upc_lock_t, pointed to by the pointer given as argument.

2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns. Otherwise, the function keeps trying to get access to the lock.

3 A null strict access is implied after a call to upc_lock().

4 If the calling thread is already holding the lock pointed to by ptr (i.e., it has previously locked it using upc_lock() or upc_lock_attempt(), but not unlocked it), the result is undefined.

7.2.4.6 The upc_lock_attempt function

Synopsis
1

38
#include <upc.h>

int upc_lock_attempt(upc_lock_t *ptr);

Description
1 The `upc_lock_attempt` function tries to lock a shared variable, of type `upc_lock_t`, pointed to by the pointer given as argument.
2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0.
3 A null strict access is implied after a call to `upc_lock_attempt()` that returns 1.
4 If the calling thread is already holding the lock pointed to by `ptr` (i.e., it has previously locked it using `upc_lock()` or `upc_lock_attempt()`, but not unlocked it), the result is undefined.

7.2.4.7 The `upc_unlock` function

Synopsis
1 #include <upc.h>

    void upc_unlock(upc_lock_t *ptr);

Description
1 The `upc_unlock` function frees the lock and does not return any value.
2 A null strict access is implied before a call to `upc_unlock()`.

7.2.5 Shared string handling functions

7.2.5.1 The `upc_memcpy` function

Synopsis
1 #include <upc.h>

    void upc_memcpy(shared void *dst, shared const void *src, size_t n);
**Description**

1. The `upc_memcpy` function copies `n` characters from a shared object having affinity with one thread to a shared object having affinity with the same or another thread. If copying takes place between objects that overlap, the behavior is undefined.

2. The `upc_memcpy` function treats the `dst` and `src` pointers as if they had type:

   ```c
   shared [] char[n]
   ```

   The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to another shared array object with this type (the `dst` array).

**7.2.5.2 The upc_memget function**

**Synopsis**

```c
#include <upc.h>

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

**Description**

1. The `upc_memget` function copies `n` characters from a shared object with affinity to any single thread to an object on the calling thread. If copying takes place between objects that overlap, the behavior is undefined.

2. The `upc_memget` function treats the `src` pointer as if it had type:

   ```c
   shared [] char[n]
   ```

   The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to an array object (the `dst` array) declared with the type

   ```c
   char[n]
   ```
7.2.5.3 The upc_memput function

Synopsis
1

```c
#include <upc.h>

void upc_memput(shared void *dst, const void *src, size_t n);
```

Description
1 The upc_memput function copies n characters from the an object on the calling thread to a shared object with affinity to any single thread. If copying takes place between objects that overlap, the behavior is undefined.
2 The upc_memput function is equivalent to copying the entire contents from an array object (the src array) declared with the type

```c
char[n]
```
to a shared array object (the dst array) with the type

```c
shared [] char[n]
```

7.2.5.4 The upc_memset function

Synopsis
1

```c
#include <upc.h>

void upc_memset(shared void *dst, int c, size_t n);
```

Description
1 The upc_memset function copies the value of c, converted to an unsigned char, to a shared object with affinity to any single thread. The number of bytes set is n.
2 The upc_memset function treats the dst pointer as if had type:

```c
shared [] char[n]
```

The effect is equivalent to setting the entire contents of a shared array object with this type (the dst array) to the value c.
7.3 UPC Collectives Operations <upc_collective.h>

1. The following requirements apply to all of the functions defined in this document whose names begin upc_all...

2. All of the functions are collective.

3. All collective function arguments are single-valued.

4. Collective functions may not be called between upc_notify and the corresponding upc_wait.

5. The last argument of each collective function is the variable sync_mode of type upc_flag_t. Values of sync_mode are formed by or-ing together a constant of the form UPC_IN_XSYNC and a constant of the form UPC_OUT_YSYNC, where X and Y may be NO, MY, or ALL.

6. If sync_mode has the value (UPC_IN_XSYNC | UPC_OUT_YSYNC), then if X is

   NO the collective function may begin to read or write data when the first thread has entered the collective function call,

   MY the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and

   ALL the collective function may begin to read or write data only after all threads have entered the collective function call.

7. and if Y is

   NO the collective function may read and write data until the last thread has returned from the collective function call,

   MY the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete, and

   **UPC_IN_ALLSYNC requires the collective function to guarantee that after all threads have entered the collective function call all threads will read the same values of the input data.**

   **UPC_OUT_MYSYNC requires the collective function to guarantee that after a thread returns from the collective function call the thread will not read any earlier values of the output data with affinity to that thread.**

42
the collective function call may return only after all reads and writes of
data are complete. \[28\]

8 UPC_IN_XSYNC alone is equivalent to (UPC_IN_XSYNC | UPC_OUT_ALLSYNC),
UPC_OUT_XSYNC alone is equivalent to (UPC_IN_ALLSYNC | UPC_OUT_XSYNC),
and 0 is equivalent to (UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC), where X is NO,
MY, or ALL.

Forward reference: upc_flag_t (7.3.7.3).

1 The standard header is
<upc_collective.h>

2 upc_collective.h contains the definitions of the types upc_flag_t and upc_op_t.
upc_flag_t and upc_op_t are of integral type.

Forward reference: upc_op_t (7.3.2.7.3.2).

7.3.1 Relocalization Operations

7.3.1.1 The upc_all_broadcast function

Synopsis

1 #include <upc.h>
#include <upc_collective.h>

void upc_all_broadcast(shared void *dst, shared const void *src,
size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block)

Description

1 The upc_all_broadcast function copies a block of memory with affinity to
a single thread to a block of shared memory on each thread. The number of

\[28\] UPC_OUT_ALLSYNC requires the collective function to guarantee that after a thread
returns from the collective function call the thread will not read any earlier values of the
output data.

UPC_OUT_ALLSYNC is not required to provide an “implied” barrier. For example, if the
entire collective operation has been completed by a certain thread before some other
threads have reached their corresponding function calls, then that thread may exit its call.
bytes in each block is nbytes. If copying takes place between objects that overlap, the behavior is undefined.

2 nbytes must be strictly greater than 0.

3 The upc_all_broadcast function treats the src pointer as if it pointed to a shared memory area with the type:

\[
\text{\textbf{shared [\㎏] char[nbytes]}}
\]

4 The effect is equivalent to copying the entire array pointed to by src to each block of nbytes bytes of a shared array dst with the type:

\[
\text{\textbf{shared [nbytes] char[nbytes * THREADS]}}
\]

5 The target of the dst pointer must have affinity to thread 0.

6 The dst pointer is treated as if it has phase 0.

7 EXAMPLE 1 shows upc_all_broadcast

\begin{verbatim}
shared int A[THREADS];
shared int B[THREADS];
// Initialize A.
upc_barrier;
upc_all_broadcast( B, &A[1], sizeof(int),
                  UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
\end{verbatim}

8 EXAMPLE 2:

\begin{verbatim}
#define NELEMS 10
shared [] int A[NELEMS];
shared [NELEMS] int B[NELEMS*THREADS];
// Initialize A.
upc_all_broadcast( B, A, sizeof(int)*NELEMS,
                   UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
\end{verbatim}

9 EXAMPLE 3 shows (A[3], A[4]) is broadcast to (B[0], B[1]), (B[10], B[11]), (B[20], B[21]), ..., (B[NELEMS*(THREADS-1)], B[NELEMS*(THREADS-1)+1]).
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
// Initialize A.
upc_barrier;
upc_all_broadcast( B, &A[3], sizeof(int)*2,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;

7.3.1.2 The upc_all_scatter function

Synopsis

```c
#include <upc.h>
#include <upc_collective.h>

void upc_all_scatter(shared void *dst, shared const void *src,
    size_t nbytes, upc_flag_t sync_mode);
```

nbytes: the number of bytes in a block

Description

1 The upc_all_scatter function copies the $i$th block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the $i$th thread. The number of bytes in each block is $nbytes$. If copying takes place between objects that overlap, the behavior is undefined.

2 $nbytes$ must be strictly greater than 0.

3 The upc_all_scatter function treats the src pointer as if it pointed to a shared memory area with the type:

```
shared [] char[nbytes * THREADS]
```

4 and it treats the dst pointer as if it pointed to a shared memory area with the type:

```
shared [nbytes] char[nbytes * THREADS]
```
The target of the `dst` pointer must have affinity to thread 0.

The `dst` pointer is treated as if it has phase 0.

For each thread `i`, the effect is equivalent to copying the `i`th block of `nbytes` bytes pointed to by `src` to the block of `nbytes` bytes pointed to by `dst` that has affinity to thread `i`.

**EXAMPLE 1** `upc_all_scatter` for the dynamic threads compilation environment.

```c
define NUMELEMS 10
define SRC_THREAD 1
shared int *A;
shared [] int *myA, *srcA;
shared [NUMELEMS] int B[NUMELEMS*THREADS];

// allocate and initialize an array distributed across all threads
A = upc_all_alloc(THREADS, THREADS*NUMELEMS*sizeof(int));
myA = (shared [] int *) &A[MYTHREAD];
for (i=0; i<NUMELEMS*THREADS; i++)
    myA[i] = i + NUMELEMS*THREADS*MYTHREAD; // (for example)
// scatter the SRC_THREAD's row of the array
srcA = (shared [] int *) &A[SRC_THREAD];
upc_barrier;
upc_all_scatter( B, srcA, sizeof(int)*NUMELEMS,
               UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

**EXAMPLE 2** `upc_all_scatter` for the static threads compilation environment.

```c
#define NELEMS 10
shared [] int A[NELEMS*THREADS];
shared [NELEMS] int B[NELEMS*THREADS];
// Initialize A.
upc_all_scatter( B, A, sizeof(int)*NELEMS,
               UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
```

46
7.3.1.3 The upc_all_gather function

Synopsis
1  
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather(shared void *dst, shared const void *src,
  size_t nbytes, upc_flag_t sync_mode);
  nbytes: the number of bytes in a block

Description
1  The upc_all_gather function copies a block of shared memory that has
   affinity to the \textit{i}th thread to the \textit{i}th block of a shared memory area that has
   affinity to a single thread. The number of bytes in each block is \texttt{nbytes}. If
   copying takes place between objects that overlap, the behavior is undefined.
2  \texttt{nbytes} must be strictly greater than 0.
3  The upc_all_gather function treats the \texttt{src} pointer as if it pointed to a
   shared memory area of \texttt{nbytes} bytes on each thread and therefore had type:

   \begin{verbatim}
   shared [nbytes] char[nbytes * THREADS]
   \end{verbatim}

4  and it treats the \texttt{dst} pointer as if it pointed to a shared memory area with
   the type:

   \begin{verbatim}
   shared [] char[nbytes * THREADS]
   \end{verbatim}

5  The target of the \texttt{src} pointer must have affinity to thread 0.
6  The \texttt{src} pointer is treated as if it has phase 0.
7  For each thread \textit{i}, the effect is equivalent to copying the block of \texttt{nbytes} bytes
   pointed to by \texttt{src} that has affinity to thread \textit{i} to the \textit{i}th block of \texttt{nbytes}
   bytes pointed to by \texttt{dst}.
8  EXAMPLE 1 upc_all_gather for the static threads compilation environment.
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [] int B[NELEMS*THREADS];
// Initialize A.
upc_all_gather( B, A, sizeof(int)*NELEMS,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );

EXAMPLE 2 upc_all_gather for the dynamic threads compilation environment.

#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [] int *B;
B = (shared [] int *) upc_all_alloc(1,NELEMS*THREADS*sizeof(int));
// Initialize A.
upc_barrier;
upc_all_gather( B, A, sizeof(int)*NELEMS,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;

7.3.1.4 The upc_all_gather_all function

Synopsis
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather_all(shared void *dst, shared const void *src,
    size_t nbytes, upc_flag_t sync_mode);
    nbytes: the number of bytes in a block

Description
1 The upc_all_gather_all function copies a block of memory from one shared
   memory area with affinity to the i\textsuperscript{th} thread to the i\textsuperscript{th} block of a shared
   memory area on each thread. The number of bytes in each block is nbytes.
   If copying takes place between objects that overlap, the behavior is undefined.
2 nbytes must be strictly greater than 0.
3 The upc_all_gather_all function treats the src pointer as if it pointed to a
   shared memory area of nbytes bytes on each thread and therefore had type:
and it treats the `dst` pointer as if it pointed to a shared memory area with the type:

```
shared [nbytes * THREADS] char[nbytes * THREADS * THREADS]
```

5 The targets of the `src` and `dst` pointers must have affinity to thread 0.

6 The `src` and `dst` pointers are treated as if they have phase 0.

7 The effect is equivalent to copying the `i`th block of `nbytes` bytes pointed to by `src` to the `i`th block of `nbytes` bytes pointed to by `dst` that has affinity to each thread.

8 **EXAMPLE 1** `upc_all_gather_all` for the static threads compilation environment.

```c
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared [NELEMS*THREADS] int B[THREADS][NELEMS*THREADS];

// Initialize A.
upc_barrier;
upc_all_gather_all( B, A, sizeof(int)*NELEMS,
                    UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;
```

9 **EXAMPLE 2** `upc_all_gather_all` for the dynamic threads compilation environment.

```c
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];
shared int *Bdata;
shared [] int *myB;

Bdata = upc_all_alloc(THREADS*THREADS, NELEMS*sizeof(int));
myB = (shared [] int *)&Bdata[MYTHREAD];

// Bdata contains THREADS*THREADS*NELEMS elements.
```
// myB is MYTHREAD’s row of Bdata.
// Initialize A.
upc_all_gather_all( Bdata, A, NELEMS*sizeof(int),
                   UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );

7.3.1.5 The upc_all_exchange function

Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_exchange(shared void *dst, shared const void *src,
                        size_t nbytes, upc_flag_t sync_mode);
```

Description

1 The **upc_all_exchange** function copies the *i*th block of memory from a shared memory area that has affinity to thread *j* to the *j*th block of a shared memory area that has affinity to thread *i*. The number of bytes in each block is *nbytes*. If copying takes place between objects that overlap, the behavior is undefined.

2 *nbytes* must be strictly greater than 0.

3 The **upc_all_exchange** function treats the *src* pointer and the *dst* pointer as if each pointed to a shared memory area of *nbytes* * THREADS bytes on each thread and therefore had type:

```
shared [nbytes * THREADS] char[nbytes * THREADS * THREADS]
```

4 The targets of the *src* and *dst* pointers must have affinity to thread 0.

5 The *src* and *dst* pointers are treated as if they have phase 0.

6 For each pair of threads *i* and *j*, the effect is equivalent to copying the *i*th block of *nbytes* bytes that has affinity to thread *j* pointed to by *src* to the *j*th block of *nbytes* bytes that has affinity to thread *i* pointed to by *dst*.

7 EXAMPLE 1 upc_all_exchange for the static threads compilation environment.
#define NELEMS 10
shared [NELEMS*THREADS] int A[THREADS][NELEMS*THREADS];
shared [NELEMS*THREADS] int B[THREADS][NELEMS*THREADS];
// Initialize A.
upc_barrier;
upc_all_exchange( B, A, NELEMS*sizeof(int),
                   UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;

EXAMPLE 2 upc_all_exchange for the dynamic threads compilation environment.

#define NELEMS 10
shared int *Adata, *Bdata;
shared [] int *myA, *myB;
t int i;

Adata = upc_all_alloc(THREADS*THREADS, NELEMS*sizeof(int));
myA = (shared [] int *)&Adata[MYTHREAD];
Bdata = upc_all_alloc(THREADS*THREADS, NELEMS*sizeof(int));
myB = (shared [] int *)&Bdata[MYTHREAD];

// Adata and Bdata contain THREADS*THREADS*NELEMS elements.
// myA and myB are MYTHREAD’s rows of Adata and Bdata, resp.

// Initialize MYTHREAD’s row of A. For example,
for (i=0; i<NELEMS*THREADS; i++)
   myA[i] = MYTHREAD*10 + i;

upc_all_exchange( Bdata, Adata, NELEMS*sizeof(int),
                   UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );

7.3.1.6 The upc_all_permute function

Synopsis
1
   #include <upc.h>
#include <upc_collective.h>

void upc_all_permute(shared void *dst, shared const void *src, 
                       shared const int *perm, size_t nbytes, 
                       upc_flag_t sync_mode);

nbytes: the number of bytes in a block

Description

1 The upc_all_permute function copies a block of memory from a shared memory area that has affinity to the \( i \)th thread to a block of a shared memory that has affinity to thread \( \text{perm}[i] \). The number of bytes in each block is \( \text{nbytes} \). If copying takes place between objects that overlap, the behavior is undefined.

2 \( \text{nbytes} \) must be strictly greater than 0.

3 \( \text{perm}[0..\text{THREADS}-1] \) must contain \( \text{THREADS} \) distinct values: 0, 1, ..., \( \text{THREADS}-1 \).

4 The upc_all_permute function treats the src pointer and the dst pointer as if each pointed to a shared memory area of \( \text{nbytes} \) bytes on each thread and therefore had type:

\[
\text{shared} \ [\text{nbytes}] \ \text{char}[\text{nbytes}] \ * \ \text{THREADS}
\]

5 The targets of the src, perm, and dst pointers must have affinity to thread 0.

6 The src and dst pointers are treated as if they have phase 0.

7 The effect is equivalent to copying the block of \( \text{nbytes} \) bytes that has affinity to thread \( i \) pointed to by src to the block of \( \text{nbytes} \) bytes that has affinity to thread \( \text{perm}[i] \) pointed to by dst.

8 EXAMPLE 1 upc_all_permute.

```c
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS], B[NELEMS*THREADS];
shared int P[THREADS];
// Initialize A and P.
upc_barrier;
upc_all_permute( B, A, P, sizeof(int)*NELEMS, 
                  UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;
```
7.3.2 Computational Operations

1 A variable of type upc_op_t can have the following values:

UPC_ADD Addition.

UPC_MULT Multiplication.

UPC_AND Bitwise AND for integer and character variables. Results are undefined for floating point numbers.

UPC_OR Bitwise OR for integer and character variables. Results are undefined for floating point numbers.

UPC_XOR Bitwise XOR for integer and character variables. Results are undefined for floating point numbers.

UPC_LOGAND Logical AND for all variable types.

UPC_LOGOR Logical OR for all variable types.

UPC_MIN For all data types, find the minimum value.

UPC_MAX For all data types, find the maximum value.

UPC_FUNC Use the specified commutative function func to operate on the data in the src array at each step.

UPC_NONCOMM_FUNC Use the specified noncommutative function func to operate on the data in the src array at each step.

2 The operations represented by a variable of type upc_op_t (including user-provided operators) are assumed to be associative. A reduction or a prefix reduction whose result is dependent on the order of operator evaluation will have undefined results.\footnote{Implementations are not obligated to prevent failures that might arise because of a lack of associativity of built-in functions due to floating-point roundoff or overflow.}

Forward references: reduction (7.3.2.1), prefix reduction (7.3.2.2).

3 The operations represented by a variable of type upc_op_t (except those provided using UPC_NONCOMM_FUNC) are assumed to be commutative. A reduction
or a prefix reduction (using operators other than UPC_NONCOMM_FUNC) whose result is dependent on the order of the operands will have undefined results.

**Forward references:** reduction (7.3.2.1), prefix reduction (7.3.2.2).

### 7.3.2.1 The upc_all_reduce function

**Synopsis**

```c
#include <upc.h>
#include <upc_collective.h>

void upc_all_reduceT(shared void *dst, shared const void *src,
                      upc_op_t op, size_t nelems, size_t blk_size,
                      TYPE (*func)(TYPE, TYPE), upc_flag_t sync_mode);
```

- **nelems:** the number of elements
- **blk_size:** the number of elements in a block

**Description**

1. The function prototype above represents the 11 variations of the upc_all_reduceT function where T and TYPE have the following correspondences:

<table>
<thead>
<tr>
<th>T</th>
<th>TYPE</th>
<th>T</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>signed char</td>
<td>L</td>
<td>signed long</td>
</tr>
<tr>
<td>UC</td>
<td>unsigned char</td>
<td>UL</td>
<td>unsigned long</td>
</tr>
<tr>
<td>S</td>
<td>signed short</td>
<td>F</td>
<td>float</td>
</tr>
<tr>
<td>US</td>
<td>unsigned short</td>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>I</td>
<td>signed int</td>
<td>LD</td>
<td>long double</td>
</tr>
<tr>
<td>UI</td>
<td>unsigned int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. For example, if T is C, then TYPE must be signed char.

3. If the value of blk_size passed to upc_all_reduceT is greater than 0 then upc_all_reduceT treats the src pointer as if it pointed to a shared memory area of nelems elements of type TYPE and blocking factor blk_size, and therefore had type:

```
shared [blk_size] TYPE [nelems]
```
If the value of \texttt{blk\_size} passed to \texttt{upc\_all\_reduceT} is 0 then \texttt{upc\_all\_reduceT} treats the \texttt{src} pointer as if it pointed to a shared memory area of \texttt{nelems} elements of type \texttt{TYPE} with an indefinite layout qualifier, and therefore had type\textsuperscript{30}:

\begin{verbatim}
shared [] TYPE[nelems]
\end{verbatim}

The phase of the \texttt{src} pointer is respected when referencing array elements, as specified in items 7.3.2.1 and 7.3.2.1 above.

\texttt{upc\_all\_reduceT} treats the \texttt{dst} pointer as having type:

\begin{verbatim}
shared TYPE *
\end{verbatim}

and at function exit the value of the \texttt{TYPE} shared object referenced by \texttt{dst} is \texttt{src[0] ⊕ src[1] ⊕ ⋯ ⊕ src[nelems-1]} where “⊕” is the operator specified by the variable \texttt{op}.

EXAMPLE 1 \texttt{upc\_all\_reduce} of type \texttt{long UPC\_ADD}.

\begin{verbatim}
#define BLK\_SIZE 3
#define NELEMS 10
shared [BLK\_SIZE] long A[NELEMS\*THREADS];
shared long *B;
long result;
// Initialize A. The result below is defined only on thread 0.
upc\_barrier;
upc\_all\_reduceL(B, A, UPC\_ADD, NELEMS\*THREADS, BLK\_SIZE,
        NULL, UPC\_IN\_NOSYNC | UPC\_OUT\_NOSYNC );
upc\_barrier;
\end{verbatim}

7.3.2.2 The \texttt{upc\_all\_prefix\_reduce} function

**Synopsis**

\textsuperscript{30}Note that \texttt{upc\_block\_size(src) == 0} if \texttt{src} has this type, so the argument value 0 has a natural connection to the block size of \texttt{src}. 

55
#include <upc.h>
#include <upc_collective.h>

void upc_all_prefix_reduceT(shared void *dst, shared const void *src
upc_op_t op, size_t nelems, size_t blk_size,
TYPE (*func)(TYPE, TYPE),
upc_flag_t sync_mode);

nelems: the number of elements
blk_size: the number of elements in a block

Description

1 The function prototype above represents the 11 variations of the upc_all_reduceT
function where T and TYPE have the following correspondences:

<table>
<thead>
<tr>
<th>T</th>
<th>TYPE</th>
<th>T</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>signed char</td>
<td>L</td>
<td>signed long</td>
</tr>
<tr>
<td>UC</td>
<td>unsigned char</td>
<td>UL</td>
<td>unsigned long</td>
</tr>
<tr>
<td>S</td>
<td>signed short</td>
<td>F</td>
<td>float</td>
</tr>
<tr>
<td>US</td>
<td>unsigned short</td>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>I</td>
<td>signed int</td>
<td>LD</td>
<td>long double</td>
</tr>
<tr>
<td>UI</td>
<td>unsigned int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 For example, if T is C, then TYPE must be signed char.

3 If the value of blk_size passed to upc_all_prefix_reduceT is greater than
0 then upc_all_prefix_reduceT treats the src pointer and the dst pointer
as if each pointed to a shared memory area of nelems elements of type TYPE
and blocking factor blk_size, and therefore had type:

shared [blk_size] TYPE[nelems]

4 If the value of blk_size passed to upc_all_prefix_reduceT is 0 then
upc_all_prefix_reduceT treats the src pointer and the dst pointer as if
each pointed to a shared memory area of nelems elements of type TYPE with
an indefinite layout qualifier, and therefore had type\(^{31}\):

shared [] TYPE [nelems]

\(^{31}\)Note that upc_blocksize(src) == 0 if src has this type, so the argument value 0
has a natural connection to the block size of src.
5 The phases of the src and dst pointers are respected when referencing array elements, as specified in items 7.3.2.2 and 7.3.2.2 above.

6 If the memory areas pointed to by src and dst overlap, the behavior of this function is undefined.

7 At function exit $dst[i] = src[0] \oplus src[1] \oplus \cdots \oplus src[i]$ for $0 \leq i \leq nelems-1$ and where “$\oplus$” is the operator specified by the variable op.

8 EXAMPLE 1 upc_all_prefix_reduce of type long UPC_ADD.

```c
#define BLK_SIZE 3
#define NELEMS 10
shared [BLK_SIZE] long A[NELEMS*THREADS);
shared [BLK_SIZE] long B[NELEMS*THREADS];
// Initialize A.
upc_all_prefix_reduceL( B, A, UPC_ADD, NELEMS*THREADS, BLK_SIZE, NULL, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
```

7.3.2.3 The upc_all_sort function

**Synopsis**

```c
#include <upc.h>
#include <upc_collective.h>
void upc_all_sort(shared void *A, size_t elem_size, size_t nelems,
                   size_t blk_size, int (*func)(shared void *, shared void *),
                   upc_flag_t sync_mode);
```

- **elem_size**: the size of each element
- **nelems**: the number of elements
- **blk_size**: the number of elements in a block

**Description**

1 The function `upc_all_sort` takes a shared array `A` of `nelems` elements of size `elem_size` bytes each and sorts them in place in ascending order using the function `func` to compare elements.

2 If the value of `blk_size` passed to `upc_all_sort` is greater than 0 then `upc_all_sort` treats the array `A` as if it had blocking factor `blk_size`.  

57
If the value of `blk_size` passed to `upc_all_sort` is 0 then `upc_all_sort` treats the array `A` as if it had an indefinite layout qualifier.

The phase of the `A` is respected when referencing array elements, as specified in items 7.3.2.3 and 7.3.2.3 above.

The function `func` shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

The function `upc_all_sort` performs a stable sort, that is, elements which compare equal are not reordered.

EXAMPLE 1 `upc_all_sort`.

```c
#define NELEMS 10
shared [NELEMS] int A[NELEMS*THREADS];

int lt_int( shared void *x, shared void *y )
{
  int x_val = *(shared int *)x, y_val = *(shared int *)y;
  return x_val > y_val ? -1 : x_val < y_val ? 1 : 0;
}

// Initialize A.
upc_barrier;
upc_all_sort( A, sizeof(int), NELEMS*THREADS, NELEMS,
              lt_int, UPC_IN_NOSYNC | UPC_OUT_NOSYNC );
upc_barrier;
```

7.4 UPC Parallel I/O `<upc_io.h>`

This subsection provides the UPC parallel extensions of Section 7.19 in [ISO/IEC00]. All the characteristics of library functions described in section 7.1.4 in [ISO/IEC00] apply to these as well.

**Common Constraints**

1 All UPC-IO functions are collective and must be called by all threads collectively.\(^{32}\)

\(^{32}\)Note that collective does not necessarily imply barrier synchronization. The synchro-
2 If a program calls `exit`, `upc_global_exit`, or returns from `main` with a UPC file still open, the file will automatically be closed at program termination, and the effect will be equivalent to `upc_all_fclose` being implicitly called on the file.

3 If a program attempts to read past the end of a file, the read function will read data up to the end of file and return the number of bytes actually read, which may be less than the amount requested.

4 Writing past the end of a file increases the file size.

5 If a program seeks to a location past the end of a file and writes starting from that location, the data in the intermediate (unwritten) portion of the file is undefined. For example, if a program opens a new file (of size 0 bytes), seeks to offset 1024 and writes some data beginning from that offset, the data at offsets 0–1023 is undefined. Seeking past the end of file and performing a write causes the current file size to immediately be extended up to the end of the write. However, just seeking past the end of file or attempting to read past the end of file, without a write, does not extend the file size.

6 All “shared void *” pointers passed to the I/O functions (as function arguments or indirectly through the list I/O arguments) are treated as if they had a phase field of zero (that is, the input phase is ignored).

7 All UPC-IO read/write functions take an argument `sync_mode` of type `upc_flag_t`. `sync_mode` values are obtained by performing a bitwise OR of a constant of the form `UPC_IN_XSYNC` and a constant of the form `UPC_OUT_YSYNC`, where X and Y may be `NO`, `MY`, or `ALL`. The `sync_mode` argument is similar to the `sync_mode` argument used in collective operation functions. The `sync_mode` argument and `upc_flag_t` type are further discussed in Section 7.4.2.3.

8 The arguments to all UPC-IO functions are single-valued (must have the same value on all threads) except where explicitly noted otherwise in the function description.

9 UPC-IO, by default, supports weak consistency and atomicity semantics. The default (weak) semantics are as follows. The data written to a file by a thread is only guaranteed to be visible to another thread after all threads have collectively closed or synchronized the file.

Initialization behavior of the UPC-IO data movement library functions is explicitly controlled by using the `sync_mode` flag argument. See Section 7.4.2.3 for details.
Writes to a file from a given thread are always guaranteed to be visible to subsequent file reads by the same thread, even without an intervening call to collectively close or synchronize the file.

Byte-level data consistency is supported.

If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined with the weak consistency and atomicity semantics.

When reading data being concurrently written by another thread, the data that gets read is undefined with the weak consistency and atomicity semantics.

File reads into overlapping locations in a shared buffer in memory using individual file pointers or list I/O functions leads to undefined data in the target buffer under the weak consistency and atomicity semantics.

A given file must not be opened at the same time by the POSIX I/O and UPC-IO libraries.

Except where otherwise noted, all UPC-IO functions return NON-single-valued errors; that is, the occurrence of an error need only be reported to at least one thread, and the \texttt{errno} value reported to each such thread may differ. When an error is reported to ANY thread, the position of ALL file pointers for the relevant file handle becomes undefined.

The error values that UPC-IO functions may set in \texttt{errno} are implementation-defined, however the \texttt{perror()} and \texttt{strerror()} functions are still guaranteed to work properly with \texttt{errno} values generated by UPC-IO.

UPC-IO functions can not be called between \texttt{upc_notify} and corresponding \texttt{upc_wait} statements.

### 7.4.1 Background

#### 7.4.1.1 File Accessing and File Pointers

Collective UPC-IO accesses can be done in and out of shared and private buffers, thus local and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. Note that in UPC-IO, common file pointers cannot be used in conjunction
with pointer-to-local buffers. File pointer modes are specified by passing a flag to the collective \texttt{upc\_all\_fopen} function and can be changed using \texttt{upc\_all\_fcntl}. When a file is opened with the common file pointer flag, all threads share a common file pointer. When a file is opened with the individual file pointer flag, each thread gets its own file pointer.

2 UPC-IO also provides file-pointer-independent list file accesses by specifying explicit offsets and sizes of data that is to be accessed. List IO can also be used with either pointer-to-local buffers or pointer-to-shared buffers.

3 Examples 1-3 and their associated figures, Figures 2-4, give typical instances of UPC-IO usage. Error checking is omitted for brevity.

4 EXAMPLE 1: collective read operation using individual file pointers

```c
double buffer[10]; // and assuming a total of 4 THREADS
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
upc_all_fseek( fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET );
upc_all_fread_local( fd, buffer, sizeof(double), 10,
                     UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
upc_all_fclose(fd);
```

Each thread reads a block of data into a private buffer from a particular thread-specific offset.

5 EXAMPLE 2: a collective read operation using a common file pointer. The data read is stored into a shared buffer, having a block size of 5 elements. The user selects the type of file pointer at file-open time. The user can select either individual file pointers by passing the flag UPC\_INDIVIDUAL\_FP to the function \texttt{upc\_all\_fopen}, or the common file pointer by passing the flag UPC\_COMMON\_FP to \texttt{upc\_all\_fopen}.

```c
shared [5] float buffer[20]; // and assuming a total of 4 static THREADS
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL );
upc_all_fread_shared( fd, buffer, upc_blocksizeof(buffer),
                      upc elemsizeof(buffer), 20, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```
EXAMPLE 3: a collective list I/O read operation. The list I/O functions allow the user to specify noncontiguous accesses both in memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case.

```c
upc_local_memvec_t memvec[2];
upc_filevec_t filevec[2];
upc_file_t *fd;
char buffer[12];

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
memvec[0].baseaddr = &buffer[0];
memvec[0].len = 4;
memvec[1].baseaddr = &buffer[7];
memvec[1].len = 3;
filevec[0].offset = MYTHREAD*5;
filevec[0].len = 2;
filevec[1].offset = 10+MYTHREAD*5;
filevec[1].len = 5;

upc_all_fread_list_local( fd, 2, &memvec, 2, &filevec,
                          UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

7.4.1.2 Synchronous and Asynchronous I/O

I/O operations can be synchronous (blocking) or asynchronous (non-blocking). While synchronous calls are quite simple and easy to use from a programming point of view, asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately. Thus,
the executing process can turn its attention to other processing needs while
the I/O is progressing.

UPC-IO supports both synchronous and asynchronous I/O functionality. The
asynchronous I/O functions have the same syntax and basic semantics as
their synchronous counterparts, with the addition of the “async” suffix in
their names. The asynchronous I/O functions have the restriction that only
one (collective) asynchronous operation can be active at a time on a given file
handle. That is, an asynchronous I/O function must be completed by calling
`upc_all_fsync_async` or `upc_all_fwait_async` before another asynchronous
I/O function can be called on the same file handle. This restriction is similar
to the restriction MPI-IO [MPI2] has on split-collective I/O functions: only
one split collective operation can be outstanding on an MPI-IO file handle
at any time.

7.4.1.3 Consistency and Atomicity Semantics

Let us first define what we mean by the terms consistency semantics and
atomicity semantics. The consistency semantics define when the data written
to a file by a thread is visible to other threads. The atomicity semantics
define the outcome of operations in which multiple threads write concurrently
to a file or shared buffer and some of the writes overlap each other. For
performance reasons, UPC-IO uses weak consistency and atomicity semantics
by default. The user can select stronger semantics either by opening the file
with the flag `UPC_STRONG_CA` or by calling `upc_all_fcntl` with the command
`UPC_SET_STRONG_CA_SEMANTICS`.

The default (weak) semantics are as follows. The data written by a thread is
only guaranteed to be visible to another thread after all threads have called
`upc_all_fclose` or `upc_all_fsync`. (Note that the data may be visible to
other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that
the data may become visible to different threads at different times.) Writes
from a given thread are always guaranteed to be visible to subsequent reads
by the same thread, even without an intervening call to `upc_all_fclose` or
`upc_all_fsync`. Byte-level data consistency is supported. So for example, if
thread 0 writes one byte at offset 0 in the file and thread 1 writes one byte
at offset 1 in the file, the data from both threads will get written to the file.
If concurrent writes from multiple threads overlap in the file, the resulting
data in the overlapping region is undefined. Similarly, if one thread tries to
read the data being concurrently written by another thread, the data that
gets read is undefined. Concurrent in this context means any two read/write
operations to the same file handle with no intervening calls to upc_all_fsync
or upc_all_fclose.

3 For the functions that read into or write from a shared buffer using a common
file pointer, the weak consistency semantics are defined as follows. Each call
to upc_all.{fread,fwrite}_shared[async] with a common file pointer
behaves as if the read/write operations were performed by a single, distinct,
anonymous thread which is different from any compute thread (and different
for each operation). In other words, NO file reads are guaranteed to see the
results of file writes using the common file pointer until after a close or sync
under the default weak consistency semantics.

4 By passing the UPC_STRONG_CA flag to upc_all_fopen or by calling upc_all_fcntl
with the command UPC_SET_STRONG_CA_SEMANSTICS, the user selects strong
consistency and atomicity semantics. In this case, the data written by a
thread is visible to other threads as soon as the file write on the calling
thread returns. In the case of writes from multiple threads to overlapping re-
gions in the file, the result would be as if the individual write function from
each thread occurred atomically in some (unspecified) order. Overlapping
writes to a file in a single (list I/O) write function on a single thread are
not permitted (see Section 7.4.6). While strong consistency and atomicity
semantics are selected on a given file handle, the sync_mode argument to
all fread/fwrite functions on that handle is ignored and always treated as
UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC.

5 The consistency semantics also define the outcome in the case of overlapping
reads into a shared buffer in memory when using individual file pointers or
list I/O functions. By default, the data in the overlapping space is undefined.
If the user selects strong consistency and atomicity, the result would be as if
the individual read functions from each thread occurred atomically in some
(unspecified) order. Overlapping reads into memory buffers in a single (list
I/O) read function on a single thread are not permitted (see Section 7.4.6).

6 Note that in strong consistency and atomicity mode, atomicity is guaranteed
at the UPC-IO function level. The entire operation specified by a single
function is performed atomically, regardless of whether it represents a single,
contiguous read/write or multiple noncontiguous reads or writes as in a list
EXAMPLE 1: three threads write data to a file concurrently, each with a single list I/O function. The numbers indicate file offsets and brackets indicate the boundaries of a listed vector. Each thread writes its own thread id as the data values:

thread 0: {1 2 3} {5 6 7 8}
thread 1: {0 1 2}{3 4 5}
thread 2: {4 5 6} {8 9 10 11}

With the default weak semantics, the results in the overlapping locations are undefined. Therefore, the result in the file would be the following, where x represents undefined data.

File: 1 x x x x x x 0 x 2 2 2

That is, the data from thread 1 is written at location 0, the data from thread 0 is written at location 7, and the data from thread 2 is written at locations 9, 10, and 11, because none of these locations had overlapping writes. All other locations had overlapping writes, and consequently, the result at those locations is undefined.

If the file were opened with the UPC_STRONG_CA flag, strong consistency and atomicity semantics would be in effect. The result, then, would depend on the order in which the writes from the three threads actually occurred. Since six different orderings are possible, there can be six outcomes. Let us assume, for example, that the ordering was the write from thread 0, followed by the write from thread 2, and then the write from thread 1. The (list I/O) write from each thread happens atomically. Therefore, for this ordering, the result would be:

File: 1 1 1 1 1 1 2 0 2 2 2

We note that if instead of using a single list I/O function each thread used a separate function to write each contiguous portion, there would be six write functions, two from each thread, and the atomicity would be at the granularity of the write operation specified by each of those functions.
### 7.4.1.4 File Interoperability

UPC-IO does not specify how an implementation may store the data in a file on the storage device. Accordingly, it is implementation-defined whether or not a file created by UPC-IO can be directly accessed by using C/POSIX I/O functions. However, the UPC-IO implementation must specify how the user can retrieve the file from the storage system as a linear sequence of bytes and vice versa. Similarly, the implementation must specify how familiar operations, such as the equivalent of POSIX `ls`, `cp`, `rm`, and `mv` can be performed on the file.

### 7.4.2 Predefined Types

#### 7.4.2.1 The upc_off_t type

**Synopsis**

```c
#include <upc_io.h>

upc_off_t myOffset;
```

**Description**

`upc_off_t` is a signed integral type that is capable of representing the size of the largest file supported by the implementation.

#### 7.4.2.2 The upc_file_t type

**Synopsis**

```c
#include <upc_io.h>

upc_file_t *myFile;
```

**Description**

An opaque shared data type of incomplete type (as defined in section 6.2.5 of [ISO/IEC00]) that represents an open file handle: `upc_file_t`
Constraints

1. `upc_file_t` objects are always manipulated via a pointer (that is, `upc_file_t *`).

2. `upc_file_t` is a shared data type. It is legal to pass a (`upc_file_t *`) across threads, and two pointers to `upc_file_t` that reference the same logical file handle will always compare equal.

Advice to Implementors

1. The definition of `upc_file_t` does not restrict the implementation to store all its metadata with affinity to one thread. Each thread can still have local access to its metadata. For example, below is a simple approach an implementation could use:

```c
/* for a POSIX-based implementation */
typedef int my_internal_filehandle_t;

#ifdef _UPC_INTERNAL
    typedef struct _local_upc_file_t {
        my_internal_filehandle_t fd;
        ... other metadata ...
    } local_upc_file_t;
#else
    struct _local_upc_file_t;
#endif

typedef shared struct _local_upc_file_t upc_file_t;

upc_file_t *upc_all_fopen(...) {
    upc_file_t *handles =
        upc_all_alloc(THREADS, sizeof(upc_file_t));

    /* get my handle */
    upc_file_t *myhandle = &(handles[MYTHREAD]);

    /* cast to a pointer-to-local */
    local_upc_file_t* mylocalhandle = (local_upc_file_t*)myhandle;
```
/* setup my metadata using pointer-to-local */
mylocalhandle->fd = open(...);
...

return handles;
}

2 The basic idea is that the “handle” exposed to the user actually points to a cyclic, distributed array. As a result, each thread has easy, local access to its own internal handle metadata with no communication, while maintaining the property that the handle that UPC-IO exposes to the client is a single-valued pointer-to-shared. An additional advantage is that a thread can directly access the metadata for other threads, which may occasionally be desirable in the implementation.

7.4.2.3 The upc_flag_t type

Synopsis
1 #include <upc_io.h>

upc_flag_t sync_mode;

Description
1 The sync_mode argument is similar to the corresponding argument in collective operation functions.

2 If the sync_mode has the value (UPC_IN_XSYNC | UPC_OUT_YSYNC), then if X is

NO the function may begin to read or write data when the first thread has entered the I/O function call,

MY the function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
ALL the function may begin to read or write data only after all threads have entered the collective function call\textsuperscript{33} and if Y is NO the function may read and write data until the last thread has returned from the collective function call, MY the function call may return in a thread only after all reads and writes of data with affinity to the thread are complete,\textsuperscript{34} and ALL the function call may return only after all reads and writes of data are complete.\textsuperscript{35}

1 \texttt{UPC\_IN\_XSYNC} alone is equivalent to (\texttt{UPC\_IN\_XSYNC} | \texttt{UPC\_OUT\_ALLSYNC}), where \( X \) is NO, MY, or ALL.

3 \texttt{UPC\_OUT\_XSYNC} alone is equivalent to (\texttt{UPC\_IN\_ALLSYNC} | \texttt{UPC\_OUT\_XSYNC}), where \( X \) is NO, MY, or ALL.

4 0 is equivalent to (\texttt{UPC\_IN\_ALLSYNC} | \texttt{UPC\_OUT\_ALLSYNC}).

5 In the \texttt{sync\_mode} definitions above, “data” refers exclusively to data residing in user-owned memory buffers passed as arguments to the library function. In other words, the \texttt{sync\_mode} flag only governs the behavior of library accesses to memory locations in user-provided buffers — it does not restrict the behavior of read/write operations on the storage medium or any buffer memory internal to the library implementation.

6 The semantics of these flags when applied to the async variants of the \texttt{fread/fwrite} functions should be interpreted as follows: constraints that

\textsuperscript{33}\texttt{UPC\_IN\_ALLSYNC} requires the collective I/O function to guarantee that after all threads have entered the collective function call all threads will read the same values of the input data.

\textsuperscript{34}\texttt{UPC\_OUT\_MYSYNC} requires the collective I/O function to guarantee that after a thread returns from the collective function call the thread will not read any earlier values of the output data with affinity to that thread.

\textsuperscript{35}\texttt{UPC\_OUT\_ALLSYNC} requires the collective I/O function to guarantee that after a thread returns from the collective function call the thread will not read any earlier values of the output data. \texttt{UPC\_OUT\_ALLSYNC} is not required to provide an “implied” barrier. For example, if the entire collective I/O operation has been completed by a certain thread before some other threads have reached their corresponding function calls, then that thread may exit its call.
reference entry to a function call correspond to entering the fread_async/
fwrite_async call that initiates the asynchronous operation, and constraints
that reference returning from a function call correspond to returning from
the upc_all_fwait_async() or successful upc_all_ftest_async() call that
completes the asynchronous operation. Also, note that all the sync_mode
flags which govern an asynchronous operation are passed to the library dur-
ing the asynchronous initiation call.

7 The sync_mode flag is included even on the fread/fwrite_local functions (which
take a pointer-to-local as the buffer argument) in order to provide well-defined
semantics for the case where one or more of the pointer-to-local arguments
references a shared object (with local affinity). In the case where all of the
pointer-to-local arguments in a given call reference only private objects, the
sync_mode flag provides no useful additional guarantees and is recommended
to be passed as UPC_IN_NOSYNC|UPC_OUT_NOSYNC to maximize performance.

7.4.2.4 The upc_local_memvec_t type

Synopsis

#include <upc_io.h>

upc_local_memvec_t myLocalMemoryVector;

Description

upc_local_memvec_t is defined as follows:

typedef struct {
    void *baseaddr;
    size_t len;
} upc_local_memvec_t;

baseaddr and len specify a contiguous memory region in terms of the base
address and length in bytes. len may be zero, in which case that entry is
ignored.
7.4.2.5  The upc_shared_memvec_t type

Synopsis
1  #include <upc_io.h>

upc_shared_memvec_t mySharedMemoryVector;

Description
1  upc_shared_memvec_t is defined as follows:

typedef struct {
    shared void *baseaddr;
    size_t blocksize;
    size_t len;
} upc_shared_memvec_t;

2  baseaddr and len specify a shared memory region in terms of the base address
and length in bytes. len may be zero, in which case that entry is ignored.
blocksize is the block size of the shared buffer in bytes. A blocksize of 0
indicates an indefinite blocking factor.

7.4.2.6  The upc_filevec_t type

Synopsis
1  #include <upc_io.h>

upc_filevec_t myFileVector;

Description
1  upc_filevec_t is defined as follows:

typedef struct {
    upc_off_t offset;
    size_t len;
} upc_filevec_t;

2  offset and len specify a contiguous region in the file in terms of the starting
offset in the file in bytes and the length in bytes.
7.4.2.7 The upc_hint_t type

Synopsis

```c
#include <upc_io.h>

upc_hint_t myHint;
```

Description

1 upc_hint_t is defined as follows:

```c
typedef struct {
    const char *key;
    const char *value;
} upc_hint_t;
```

2 UPC-IO supports a number of predefined hints. An implementation is free to support additional hints. An implementation is free to ignore any hint provided by the user. Implementations should silently ignore any hints they do not support or recognize. The predefined hints and their meanings are defined below. An implementation is not required to interpret these hint keys, but if it does interpret the hint key, it must provide the functionality described. For each hint name introduced, we describe the type of the hint value and its meaning. All hints are single-valued character strings (the content is single-valued, not the location).

**access_style** (comma-separated list of strings): indicates the manner in which the file is expected to be accessed. The hint value is a comma-separated list of any the following: “read_once”, “write_once”, “read_mostly”, “write_mostly”, “sequential”, and “random”. Passing such a hint does not place any constraints on how the file may actually be accessed by the program, although accessing the file in a way that is different from the specified hint may result in lower performance.

**collective_buffering** (boolean): specifies whether the application may benefit from collective buffering optimizations. Legal values for this key are “true” and “false”. Collective buffering parameters can be further directed via additional hints: cb_buffer_size, and cb_nodes.
cb_buffer_size (decimal integer): specifies the total buffer space that the implementation can use on each thread for collective buffering.

cb_nodes (decimal integer): specifies the number of target threads or I/O nodes to be used for collective buffering.

file_perm (string): specifies the file permissions to use for file creation. The set of legal values for this key is implementation defined.

io_node_list (comma separated list of strings): specifies the list of I/O devices that should be used to store the file and is only relevant when the file is created.

nb_proc (decimal integer): specifies the number of threads that will typically be used to run programs that access this file and is only relevant when the file is created.

striping_factor (decimal integer): specifies the number of I/O devices that the file should be striped across and is relevant only when the file is created.

start_io_device (decimal integer): specifies the number of the first I/O device from which to start striping the file and is relevant only when the file is created.

striping_unit (decimal integer): specifies the striping unit to be used for the file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

7.4.3 UPC File Operations

Common Constraints

1. When a file is opened with an individual file pointer, each thread will get its own file pointer and advance through the file at its own pace.

2. When a common file pointer is used, all threads positioned in the file will be aligned with that common file pointer.
Common file pointers cannot be used in conjunction with pointers-to-local (and hence cannot operate on private objects).

No function in this section may be called while an asynchronous operation is pending on the file handle, except where otherwise noted.

### 7.4.3.1 The upc_all_fopen function

**Synopsis**

```
#include <upc.h>
#include <upc_io.h>

upc_file_t *upc_all_fopen(const char *fname,
                          int flags,
                          size_t numhints,
                          upc_hint_t const *hints)
```

**Description**

`upc_all_fopen` opens the file identified by the filename `fname` for input/output operations.

The flags parameter specifies the access mode. The valid flags and their meanings are listed below. Of these flags, exactly one of `UPC_RDONLY`, `UPC_WRONLY`, or `UPC_RDWR`, and one of `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP`, must be used. Other flags are optional. Multiple flags can be combined by using the bitwise OR operator (`|`), and each flag has a unique bitwise representation that can be unambiguously tested using the bitwise AND operator(`&`).

- **UPC_RDONLY** Open the file in read-only mode
- **UPC_WRONLY** Open the file in write-only mode
- **UPC_RDWR** Open the file in read/write mode
- **UPC_INDIVIDUAL_FP** Use an individual file pointer for all file accesses (other than list I/O)
- **UPC_COMMON_FP** Use the common file pointer for all file accesses (other than list I/O)
UPC_APPEND Set the initial position of the file pointer to end of file. (The file pointer is not moved to the end of file after each read/write)

UPC_CREATE Create the file if it does not already exist

UPC_EXCL Used in conjunction with UPC_CREATE. The open will fail if the file already exists

UPC_STRONG_CA Set strong consistency and atomicity semantics

UPC_TRUNC Open the file and truncate it to zero length

UPC_DELETE_ON_CLOSE Delete the file automatically on close

3 The UPC_COMMON_FP flag specifies that all accesses (except for the list I/O operations) will use the common file pointer. The UPC_INDIVIDUAL_FP flag specifies that all accesses will use individual file pointers (except for the list I/O operations). Either UPC_COMMON_FP or UPC_INDIVIDUAL_FP must be specified or upc_all_fopen will return an error.

4 The UPC_STRONG_CA flag specifies strong consistency and atomicity semantics. The data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. In the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order.

5 If the flag UPC_STRONG_CA is not specified, weak semantics are provided. The data written by a thread is only guaranteed to be visible to another thread after all threads have called upc_all_fclose or upc_all_fsync. (Note that the data may be visible to other threads before the call to upc_all_fclose or upc_all_fsync and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the same thread, even without an intervening call to upc_all_fclose or upc_all_fsync. Byte-level data consistency is supported. For the purposes of atomicity and consistency semantics, each call to upc_all_{fread,fwrite}_shared[.async] with a common file pointer behaves as if the read/write operations were performed by a single, distinct,
an anonymous thread which is different from any compute thread (and different for each operation).”

Hints can be passed to the UPC-IO library as an array of key-value pairs of strings. `numhints` specifies the number of hints in the `hints` array; if `numhints` is zero, the `hints` pointer is ignored. The user can free the `hints` array and associated character strings as soon as the open call returns. Each element of the `hints` array is of type `upc_hint_t`.

A file on the storage device is in the `open` state from the beginning of a successful open call to the end of the matching successful close call on the file handle. It is erroneous to have the same file `open` simultaneously with two `upc_all_fopen` calls, or with a `upc_all_fopen` call and a POSIX/C `open` or `fopen` call.

The user is responsible for ensuring that the file referenced by the `fname` argument refers to a single UPC-IO file. The actual argument passed on each thread may be different because the file name spaces may be different on different threads, but they must all refer to the same logical UPC-IO file.

On success, the function returns a pointer to a file handle that can be used to perform other operations on the file.

`upc_all_fopen` provides single-valued errors - if an error occurs, the function returns `NULL` on ALL threads, and sets `errno` appropriately to the same value on all threads.

### 7.4.3.2 The `upc_all_fclose` function

**Synopsis**

```c
#include <upc.h>
#include <upc_io.h>

int upc_all_fclose (upc_file_t *fd);
```

**Description**

[36] In other words, NO reads are guaranteed to see the results of writes using the common file pointer until after a close or sync when `UPC_STRONG_CA` is not specified.

[37] The contents of the key/value pairs passed by all the threads must be single-valued.
upc_all_fclose executes an implicit upc_all_fsync on fd and then closes the file associated with fd.

The function returns 0 on success. If fd is not valid or if an outstanding asynchronous operation on fd has not been completed, the function will return an error.

upc_all_fclose provides single-valued errors - if an error occurs, the function returns −1 on ALL threads, and sets errno appropriately to the same value on all threads.

After a file has been closed with upc_all_fclose, the file can legally be opened and the data in it can be accessed by using regular C/POSIX I/O calls.

7.4.3.3 The upc_all_fsync function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

int upc_all_fsync(upc_file_t *fd)
```

Description

upc_all_fsync ensures that any data that has been written to the file associated with fd but not yet transferred to the storage device is transferred to the storage device. It also ensures that subsequent file reads from any thread will see any previously written values (that have not yet been overwritten).

There is an implied barrier immediately before upc_all_fsync returns.

The function returns 0 on success. On error, it returns −1 and sets errno appropriately.

7.4.3.4 The upc_all_fseek function

Synopsis
#include <upc.h>
#include <upc_io.h>

upc_off_t upc_all_fseek(upc_file_t *fd,
                        upc_off_t offset,
                        int origin)

Description
1  upc_all_fseek sets the current position of the file pointer associated with fd.
2  This offset can be relative to the current position of the file pointer, to the beginning of the file, or to the end of the file. The offset can be negative, which allows seeking backwards.
3  The origin parameter can be specified as UPCSEEK_SET, UPCSEEK_CUR, or UPCSEEK_END, respectively, to indicate that the offset must be computed from the beginning of the file, the current location of the file pointer, or the end of the file.
4  In the case of a common file pointer, all threads must specify the same offset and origin. In the case of an individual file pointer, each thread may specify a different offset and origin.
5  It is legal to seek past the end of file. It is erroneous to seek to a negative position in the file. See the Common Constraints number 5 at the beginning of Section 7.3 for more details.
6  The current position of the file pointer can be determined by calling upc_all_fseek(fd, 0, UPCSEEK_CUR).
7  On success, the function returns the current location of the file pointer in bytes. If there is an error, it returns −1 and sets errno appropriately.

7.4.3.5  The upc_all_fset_size function

Synopsis
1  #include <upc.h>
   #include <upc_io.h>
int upc_all_fset_size(upc_file_t *fd,  
                 upc_off_t size)

Description
1 upc_all_fset_size executes an implicit upc_all_fsync on fd and resizes the  
   file associated with fd.
2 size is measured in bytes from the beginning of the file.
3 If size is less than the current file size, the file is truncated at the position  
   defined by size. The implementation is free to deallocate file blocks located  
   beyond this position.
4 If size is greater than the current file size, the file size increases to size.  
   Regions of the file that have been previously written are unaffected. The  
   values of data in the new regions in the file (between the old size and size)  
   are undefined.
5 If this function truncates a file, it is possible that the individual and common  
   file pointers may point beyond the end of file. This is legal and is equivalent  
   to seeking past the end of file (see the Common Rules in Section 5 for the  
   semantics of seeking past the end of file).
6 It is unspecified whether and under what conditions this function actually  
   allocates file space on the storage device. Use upc_all_fpreallocate to  
   force file space to be reserved on the storage device.
7 Calling this function does not affect the individual or common file pointers.
8 The function returns 0 on success. On error, it returns –1 and sets errno  
   appropriately.

7.4.3.6 The upc_all_fget_size function

Synopsis
1 #include <upc.h>
   #include <upc_io.h>

   upc_off_t upc_all_fget_size(upc_file_t *fd)
Description
1 upc_all_fget_size returns the current size in bytes of the file associated with fd on success. On error, it returns –1 and sets errno appropriately.

### 7.4.3.7 The upc_all_fpreallocate function

**Synopsis**

```c
#include <upc.h>
#include <upc_io.h>

int upc_all_fpreallocate(upc_file_t *fd,
                         upc_off_t size)
```

**Description**

1 upc_all_fpreallocate ensures that storage space is allocated for the first size bytes of the file associated with fd.

2 Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, upc_all_fpreallocate has the same effect as writing undefined data.

3 If size is greater than the current file size, the file size increases to size. If size is less than or equal to the current file size, the file size is unchanged.

4 Calling this function does not affect the individual or common file pointers.

5 The function returns 0 on success. On error, it returns –1 and sets errno appropriately.

### 7.4.3.8 The upc_all_fcntl function

**Synopsis**

```c
#include <upc.h>
#include <upc_io.h>

int upc_all_fcntl(upc_file_t *fd,
                  int cmd,
                  void *arg)
```
Description

upc_all_fcntl performs one of various miscellaneous operations related to the file specified by \textit{fd}, as determined by \textit{cmd}. The valid commands \textit{cmd} and their associated argument \textit{arg} are explained below.

\textbf{UPC\_GET\_CA\_SEMANTICS} Get the current consistency and atomicity semantics for \textit{fd}. The argument \textit{arg} is ignored. The return value is \textbf{UPC\_STRONG\_CA} for strong consistency and atomicity semantics and 0 for the default weak consistency and atomicity semantics.

\textbf{UPC\_SET\_WEAK\_CA\_SEMANTICS} Executes an implicit \textit{upc_all_fsync} on \textit{fd} and sets \textit{fd} to use the weak consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument \textit{arg} is ignored. The return value is 0 on success. On error, this function returns -1 and sets \textit{errno} appropriately.

\textbf{UPC\_SET\_STRONG\_CA\_SEMANTICS} Executes an implicit \textit{upc_all_fsync} on \textit{fd} and sets \textit{fd} to use the strong consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument \textit{arg} is ignored. The return value is 0 on success. On error, this function returns -1 and sets \textit{errno} appropriately.

\textbf{UPC\_GET\_FP} Get the type of the current file pointer for \textit{fd}. The argument \textit{arg} is ignored. The return value is either \textbf{UPC\_COMMON\_FP} in case of a common file pointer, or \textbf{UPC\_INDIVIDUAL\_FP} for individual file pointers.

\textbf{UPC\_SET\_COMMON\_FP} Executes an implicit \textit{upc_all_fsync} on \textit{fd}, sets the current file access pointer mechanism for \textit{fd} to a common file pointer (or leaves it unchanged if that mode is already selected), and seeks to the beginning of the file. The argument \textit{arg} is ignored. The return value is 0 on success. On error, this function returns -1 and sets \textit{errno} appropriately.

\textbf{UPC\_SET\_INDIVIDUAL\_FP} Executes an implicit \textit{upc_all_fsync} on \textit{fd}, sets the current file access pointer mechanism for \textit{fd} to an individual file pointer (or leaves the mode unchanged if that mode is already selected), and seeks to the beginning of the file. The argument \textit{arg} is ignored. The return value is 0 on success. On error, this function returns -1 and sets \textit{errno} appropriately.
UPC_GET_FL Get all the flags specified during the upc_all_fopen call for fd, as modified by any subsequent mode changes using the upc_all_fcntl(UPC_SET_*) commands. The argument arg is ignored. The return value has same format as the flags parameter in upc_all_fopen.

UPC_GET_FN Get the file name provided by each thread in the upc_all_fopen call that created fd. The argument arg is a valid (const char**) pointing to a (const char*) location in which a pointer to file name will be written. Writes a (const char*) into *arg pointing to the file-name in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next upc_all_fcntl call on that file handle.

UPC_GET_HINTS Retrieve the hints applicable to fd. The argument arg is a valid (const upc_hint_t**) pointing to a (const upc_hint_t*) location in which a pointer to the hints array will be written. Writes a (const upc_hint_t*) into *arg pointing to an array of upc_hint_t’s in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next upc_all_fcntl call on that file handle. The number of hints in the array is returned by the call. The hints in the array may be a subset of those specified at file open time, if the implementation ignored some unrecognized or unsupported hints.

UPC_SET_HINT Executes an implicit upc_all_fsync on fd and sets a new hint to fd. The argument arg points to one single-valued upc_hint_t hint to be applied. If the given hint key has already been applied to fd, the current value for that hint is replaced with the provided value. The return value is 0 on success. On error, this function returns -1 and sets errno appropriately.

UPC_ASYNC_OUTSTANDING Returns 1 if there is an asynchronous operation outstanding on fd, or 0 otherwise.

2 In case of a non valid fd, upc_all_fcntl returns -1 and sets errno appropriately.

3 It is legal to call upc_all_fcntl(UPC_ASYNC_OUTSTANDING) when an asynchronous operation is outstanding (but it is still illegal to call upc_all_fcntl with any other argument when an asynchronous operation is outstanding).
7.4.4 Reading Data

Common Constraints

1 No function in this section 7.4.4 may be called while an asynchronous operation is pending on the file handle.

7.4.4.1 The upc_all_fread_local function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_local(upc_file_t *fd,
     void *buffer,
     size_t size,
     size_t nmemb,
     upc_flag_t sync_mode)
```

Description

1 `upc_all_fread_local` reads data from a file into a local buffer on each thread.
2 This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for reading.
3 `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.
4 Each thread reads \((size \times nmemb)\) bytes of data from the file at the position indicated by its individual file pointer into the buffer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.
5 On success, the function returns the number of bytes read into the local buffer of the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.
7.4.4.2 The upc_all_fread_shared function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_shared(upc_file_t *fd,
    shared void *buffer,
    size_t blocksize,
    size_t size,
    size_t nmemb,
    upc_flag_t sync_mode)
```

Description

1. `upc_all_fread_shared` reads data from a file into a shared buffer in memory.
2. The function can be called when the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for reading.
3. `buffer` is a pointer to an array into which data will be read. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.
4. In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread reads `(size*nmemb)` bytes of data from the file at the position indicated by its individual file pointer into its buffer. Each thread may pass a different value for `blocksize`, `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes read by the calling thread, and the individual file pointer of the thread is incremented by that amount.
5. In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter, and the same value for `blocksize`, `size` and `nmemb`. The effect is that `(size*nmemb)` bytes of data are read from the file at the position indicated by the common file pointer into the buffer. If `size` or `nmemb` is zero, the `buffer` argument is
ignored and the operation has no effect. On success, the function returns the total number of bytes read by all threads, and the common file pointer is incremented by that amount.

6 If reading with individual file pointers results in overlapping reads into the shared buffer, the result is determined by whether the file was opened with the UPC_STRONG_CA flag or not (see Section 7.4.3.1).

7 The function returns –1 on error and sets errno appropriately.

7.4.5 Writing Data

Common Constraints

1 No function in this section 7.4.5 may be called while an asynchronous oper-ation is pending on the file handle.

7.4.5.1 The upc_all_fwrite_local function

Synopsis

1 #include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_local(upc_file_t *fd,
    void *buffer,
    size_t size,
    size_t nmemb,
    upc_flag_t sync_mode)

Description

1 upc_all_fwrite_local writes data from a local buffer on each thread into a file.

2 This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for writing.

3 buffer is a pointer to an array from which data will be written, and each thread may pass a different value for buffer.
Each thread writes \((\text{size} \times \text{nmemb})\) bytes of data from the buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for \text{size} and \text{nmemb}. If \text{size} or \text{nmemb} is zero, the \text{buffer} argument is ignored and that thread performs no I/O.

If any of the writes result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for \text{fd} (see 7.4.3.1).

On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns \(-1\) and sets \text{errno} appropriately.

### 7.4.5.2 The \texttt{upc\_all\_fwrite\_shared} function

#### Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_shared(upc_file_t *fd,
                             shared void *buffer,
                             size_t blocksize,
                             size_t size,
                             size_t nmemb,
                             upc_flag_t sync_mode)
```

#### Description

\texttt{upc\_all\_fwrite\_shared} writes data from a shared buffer in memory to a file.

The function can be called if the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for writing.

\texttt{buffer} is a pointer to an array from which data will be written. It must be a pointer to shared data and may have affinity to any thread. \texttt{blocksize} is the block size of the shared buffer in elements (of \texttt{size} bytes each). A \texttt{blocksize} of 0 indicates an indefinite blocking factor.

In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the \texttt{buffer} parameter. Each thread writes
(size*nmemb) bytes of data from its buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for blocksize, size and nmemb. If size or nmemb is zero, the buffer argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount.

In the case of a common file pointer, the following rules apply: All threads must pass the same address for the buffer parameter, and the same value for blocksize, size and nmemb. The effect is that (size*nmemb) bytes of data are written from the buffer to the file at the position indicated by the common file pointer. If size or nmemb is zero, the buffer argument is ignored and the operation has no effect. On success, the function returns the total number of bytes written by all threads, and the common file pointer is incremented by that amount.

If writing with individual file pointers results in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for fd (see Section 7.4.3.1).

The function returns -1 on error and sets errno appropriately.

### 7.4.6 List I/O

**Common Constraints**

1 List I/O functions take a list of addresses/offsets and corresponding lengths in memory and file to read from or write to.

2 List I/O functions can be called regardless of whether the current file pointer type is individual or common.

3 File pointers are not updated as a result of a list I/O read/write operation.

4 The memvec argument passed to any list I/O read function by a single thread must not specify overlapping regions in memory.

5 The base addresses passed to memvec can be in any order.

6 The filevec argument passed to any list I/O write function by a single thread must not specify overlapping regions in the file.

7 The offsets passed in filevec must be in monotonically non-decreasing order.
No function in this section (7.4.6) may be called while an asynchronous operation is pending on the file handle.

No function in this section (7.4.6) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_fread_list_shared` to ensure that the entire shared buffer has been filled up, and similarly, use a barrier before calling `upc_all_fwrite_list_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.

For all the list I/O functions, each thread passes an independent set of memory and file vectors. Passing the same vectors on two or more threads specifies redundant work. The file pointer is a single-valued argument, all other arguments to the list I/O functions are NOT single-valued.

### 7.4.6.1 The `upc_all_fread_list_local` function

#### Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_list_local(upc_file_t *fd,
    size_t memvec_entries,
    upc_local_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

#### Description

`upc_all_fread_list_local` reads data from a file into local buffers in memory. The file handle must be open for reading.

`memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations.
in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4 On success, the function returns the number of bytes read by the calling thread. On error, it returns −1 and sets `errno` appropriately.

### 7.4.6.2 The upc_all_fread_list_shared function

**Synopsis**

```
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fread_list_shared(upc_file_t *fd,
                                size_t memvec_entries,
                                upc_shared_memvec_t const *memvec,
                                size_t filevec_entries,
                                upc_filevec_t const *filevec,
                                upc_flag_t sync_mode)
```

**Description**

1 The `upc_all_fread_list_shared` reads data from a file into various locations of a shared buffer in memory. The file handle must be open for reading.

2 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3 The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4 If any of the reads from different threads result in overlapping regions in memory, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.4.3.1).

5 On success, the function returns the number of bytes read by the calling thread. On error, it returns −1 and sets `errno` appropriately.
Note: With the above definition, there is no way to do with explicit offsets the equivalent of upc_all_fread_shared using a common file pointer, namely, where all threads specify the same access (same parameters), the data gets read collectively into the shared buffer, and the function returns the total amount of data read by all threads.

7.4.6.3 The upc_all_fwrite_list_local function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_list_local(upc_file_t *fd,
 size_t memvec_entries,
 upc_local_memvec_t const *memvec,
 size_t filevec_entries,
 upc_filevec_t const *filevec,
 upc_flag_t sync_mode)
```

Description

1 upc_all_fwrite_list_local writes data from local buffers in memory to a file. The file handle must be open for writing.

2 memvec_entries indicates the number of entries in the array memvec and filevec_entries indicates the number of entries in the array filevec. The values may be 0, in which case the memvec or filevec argument is ignored and no locations are specified for I/O.

3 The result is as if data were written from memory locations in the order specified by the list of locations in memvec to locations in the file in the order specified by the list in filevec. The total amount of data specified by memvec must equal the total amount of data specified by filevec.

4 If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for fd (see Section 7.4.3.1).

5 On success, the function returns the number of bytes written by the calling thread. On error, it returns -1 and sets errno appropriately.
### 7.4.6.4 The `upc_all_fwrite_list_shared` function

#### Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_fwrite_list_shared(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

#### Description

1. The `upc_all_fwrite_list_shared` function writes data from various locations of a shared buffer in memory to a file. The file handle must be open for writing.

2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3. The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

4. If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.4.3.1).

5. On success, the function returns the number of bytes written by the calling thread. On error, it returns -1 and sets `errno` appropriately.

*Note:* With the above definition, there is no way to do with explicit offsets the equivalent of `upc_all_fwrite_shared` using a common file pointer, namely, where all threads specify the same access (same parameters), the data gets written collectively from a shared buffer, and the function returns the total amount of data written by all threads.
7.4.7 Asynchronous I/O

Common Constraints

1 Only one asynchronous I/O operation can be outstanding on a UPC-IO file handle at any time. If an application attempts to initiate a second asynchronous I/O operation while one is still outstanding on the same file handle the behavior is undefined – however, high-quality implementations will issue a fatal error.

2 For asynchronous read operations, the contents of the destination memory are undefined until after a successful upc_all_fwait_async or upc_all_ftest_async on the file handle. For asynchronous write operations, the source memory may not be safely modified until after a successful upc_all_fwait_async or upc_all_ftest_async on the file handle.

3 An implementation is free to block for completion of an operation in the asynchronous initiation call or in the upc_all_ftest_async call (or both). High-quality implementations are recommended to minimize the amount of time spent within the asynchronous initiation or upc_all_ftest_async call.

4 In the case of list I/O functions, the user may modify or free the lists after the asynchronous I/O operation has been initiated.

7.4.7.1 The upc_all_fread_local_async function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_local_async(upc_file_t *fd,
                           void *buffer,
                           size_t size,
                           size_t nmemb,
                           upc_flag_t sync_mode)
```

Description

upc_all_fread_local_async initiates an asynchronous read from a file into a local buffer on each thread.
The meaning of the parameters and restrictions are the same as for the blocking function, \texttt{upc\_all\_fread\_local}.

The status of the initiated asynchronous I/O operation can be retrieved by calling \texttt{upc\_all\_ftest\_async} or \texttt{upc\_all\_fwait\_async}.

### 7.4.7.2 The \texttt{upc\_all\_fread\_shared\_async} function

#### Synopsis

```c
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_shared_async(upc_file_t *fd,
               shared void *buffer,
               size_t blocksize,
               size_t size,
               size_t nmemb,
               upc_flag_t sync_mode)
```

#### Description

- \texttt{upc\_all\_fread\_shared\_async} initiates an asynchronous read from a file into a shared buffer.
- The meaning of the parameters and restrictions are the same as for the blocking function, \texttt{upc\_all\_fread\_shared}.
- The status of the initiated asynchronous I/O operation can be retrieved by calling \texttt{upc\_all\_ftest\_async} or \texttt{upc\_all\_fwait\_async}.

### 7.4.7.3 The \texttt{upc\_all\_fwrite\_local\_async} function

#### Synopsis

```c
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_local_async(upc_file_t *fd,
               void *buffer,
               size_t blocksize,
               size_t size,
               size_t nmemb,
               upc_flag_t sync_mode)
```

---

93
Description

1. The function `upc_all_fwrite_local_async` initiates an asynchronous write from a local buffer on each thread to a file.
2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_local`.
3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.4.7.4 The `upc_all_fwrite_shared_async` function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_shared_async(upc_file_t *fd,
                                 shared void *buffer,
                                 size_t blocksize,
                                 size_t size,
                                 size_t nmemb,
                                 upc_flag_t sync_mode)
```

Description

1. The function `upc_all_fwrite_shared_async` initiates an asynchronous write from a shared buffer to a file.
2. The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_shared`.
3. The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

94
7.4.7.5 The upc_all_fread_list_local_async function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_list_local_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_local_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

Description

upc_all_fread_list_local_async initiates an asynchronous read of data from a file into local buffers in memory.

The meaning of the parameters and restrictions are the same as for the blocking function, upc_all_fread_list_local.

The status of the initiated asynchronous I/O operation can be retrieved by calling upc_all_ftest_async or upc_all_fwait_async.

7.4.7.6 The upc_all_fread_list_shared_async function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

void upc_all_fread_list_shared_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```
7.4.7.7 The upc_all_fwrite_list_local_async function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_list_local_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_local_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

Description

upc_all_fwrite_list_local_async initiates an asynchronous write of data from local buffers in memory to a file.

1 The meaning of the parameters and restrictions are the same as for the blocking function, upc_all_fwrite_list_local.

3 The status of the initiated asynchronous I/O operation can be retrieved by calling upc_all_ftest_async or upc_all_fwait_async.

7.4.7.8 The upc_all_fwrite_list_shared_async function

Synopsis

1
#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_list_shared_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)

Description
1 upc_all_fwrite_list_shared_async initiates an asynchronous write of data
   from various locations of a shared buffer in memory to a file.
2 The meaning of the parameters and restrictions are the same as for the
   blocking function, upc_all_fwrite_list_shared.
3 The status of the initiated asynchronous I/O operation can be retrieved by
   calling upc_all_ftest_async or upc_all_fwait_async.

7.4.7.9 The upc_all_fwait_async function

Synopsis
1 #include <upc.h>
   #include <upc_io.h>

   ssize_t upc_all_fwait_async(upc_file_t *fd)

Description
1 upc_all_fwait_async completes the previously issued asynchronous I/O op-
   eration on the file handle fd, blocking if necessary.
2 It is erroneous to call this function if there is no outstanding asynchronous
   I/O operation associated with fd.
3 On success, the function returns the number of bytes read or written by
   the asynchronous I/O operation as specified by the blocking variant of the
   function used to initiate the asynchronous operation. On error, it returns -1
   and sets errno appropriately, and the outstanding asynchronous operation
   (if any) becomes no longer outstanding.
7.4.7.10 The upc_all_ftest_async function

Synopsis

```c
#include <upc.h>
#include <upc_io.h>

ssize_t upc_all_ftest_async(upc_file_t *fd,
    int *flag)
```

Description

1 upc_all_ftest_async tests whether the outstanding asynchronous I/O operation associated with fd has completed.

2 If the operation has completed, the function sets flag=1 and the asynchronous operation becomes no longer outstanding;\(^38\) otherwise it sets flag=0. The same value of flag is set on all threads.

3 If the operation was completed, the function returns the number of bytes that were read or written as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns −1 and sets errno appropriately, and sets the flag=1, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

4 It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with fd.

References


[MPI2] ??

\(^{38}\)This implies it is illegal to call upc_all_fwait_async or upc_all_ftest_async immediately after a successful upc_all_ftest_async on that file handle.
A  UPC versus C Standard Section Numbering

<table>
<thead>
<tr>
<th>UPC specifications subsection</th>
<th>C Standard specifications subsection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1</td>
</tr>
<tr>
<td>6.2</td>
<td>6.4.2.2</td>
</tr>
<tr>
<td>6.3</td>
<td>6.5</td>
</tr>
<tr>
<td>6.3.6</td>
<td>6.5.3.1</td>
</tr>
<tr>
<td>6.4</td>
<td>6.7</td>
</tr>
<tr>
<td>6.4.1</td>
<td>6.7.3</td>
</tr>
<tr>
<td>6.4.3</td>
<td>6.7.5</td>
</tr>
<tr>
<td>6.5</td>
<td>6.8</td>
</tr>
<tr>
<td>6.6</td>
<td>6.10</td>
</tr>
<tr>
<td>7.1</td>
<td>7.1.2</td>
</tr>
</tbody>
</table>

Table A1. Mapping UPC subsections to C Standard specifications subsections

B  Formal UPC Memory Semantics

1  The memory consistency model in a language defines the order in which the results of write operations may be observed through read operations. The behavior of a UPC program may depend on the timing of accesses to shared variables, so a program defines a set of possible executions, rather than a single execution. The memory consistency model constrains the set of possible executions for a given program; the user may then rely on properties that are true of all of those executions.

2  The memory consistency model is defined in terms of the read and write operations issued by each thread in naïve translation of the code, i.e., without
any code transformations by the compiler – with each thread issuing operations as defined by the abstract machine defined in ISO C 5.1.2.3. A UPC compiler or runtime system may perform various code transformations to improve performance, so long as they are not visible to the programmer – i.e. provided the set of externally-visible behaviors (the input/output dynamics and volatile behavior defined in ISO C 5.1.2.3) from any execution of the transformed program are identical to those of the original program executing on the abstract machine and adhering to the consistency model defined in this document.

B.1 Definitions

1 A UPC program execution is specified by a program text and a number of threads, \( T \). An execution is a set of operations \( O \), each operation being an instance of some instruction in the program text. The set of operations issued by a thread \( t \) is denoted \( O_t \). The program executes memory operations on a set of variables (or locations) \( L \). The set \( V \) is the set of possible values that can be stored in the program variables. 39

2 A memory operation in such an execution is given by a location \( l \in L \) to be written or read and a value \( v \in V \), which is the value to be written or the value returned by the read. A memory operation \( m \) in a UPC program has one of the following forms:

- a strict shared read, denoted \( \text{SR}(l,v) \)
- a strict shared write, denoted \( \text{SW}(l,v) \)
- a relaxed shared read, denoted \( \text{RR}(l,v) \)
- a relaxed shared write, denoted \( \text{RW}(l,v) \)
- a local read, denoted \( \text{LR}(l,v) \)
- a local write, denoted \( \text{LW}(l,v) \)

39This is the point that we could add an atomicity constraint on what types of values are the fundamental unit of a read or write, possibly using something like ISO C’s \text{sig\_atomic\_t}. There are actually two separate issues here, namely atomicity and clobbering a.k.a. word tearing.
(Here shared vs local is determined by the sharing type qualification on the expression used to perform the access, and for shared accesses, strict vs relaxed is determined as described in UPC Spec 6.4.2). In addition, each memory operation $m$ is associated with exactly one of the $T$ threads, denoted $Thread(m)$, and we define the accessor $Location(m)$ to return the location $l$ accessed by $m$.

3 Given a UPC program execution with $T$ threads, let $M \subseteq O$ be the set of memory operations in the execution and $M_t$ be the set of memory operations issued by a given thread $t$. Each operation in $M$ is one of the above six types, so the set $M$ is partitioned into the following six disjoint subsets:

- $SR(M)$ is the set of strict shared reads in $M$
- $SW(M)$ is the set of strict shared writes in $M$
- $RR(M)$ is the set of relaxed shared reads in $M$
- $RW(M)$ is the set of relaxed shared writes in $M$
- $LR(M)$ is the set of local reads in $M$
- $LW(M)$ is the set of local writes in $M$

4 We denote the set of all writes in $M$ as $W(M) = SW(M) \cup RW(M) \cup LW(M)$ and the set of all strict accesses as $Strict(M) = SR(M) \cup SW(M)$.

B.2 Memory Access Model

1 Let $StrictPairs(M)$, $StrictOnThreads(M)$, and $AllStrict(M)$ be unordered pairs of memory operations defined as:

- $StrictPairs(M) \overset{\text{def}}{=} \{(m_1, m_2) | m_1 \neq m_2 \land m_1 \in Strict(M) \land m_2 \in Strict(M)\}$
- $StrictOnThreads(M) \overset{\text{def}}{=} \{(m_1, m_2) | m_1 \neq m_2 \land Thread(m_1) = Thread(m_2) \land (m_1 \in Strict(M) \lor m_2 \in Strict(M))\}$
- $AllStrict(M) \overset{\text{def}}{=} StrictPairs(M) \cup StrictOnThreads(M)$
Thus, $\textit{StrictPairs}(M)$ is the set of all pairs of strict memory accesses, including those between threads, and $\textit{StrictOnThreads}(M)$ is the set of all pairs of memory accesses from the same thread in which at least one is strict. $\textit{AllStrict}(M)$ is their union, which intuitively is the set of operation pairs on which all threads must agree upon an ordering (i.e. all threads must agree on the directionality of each pair), although what that order is may depend on the resolution of race conditions at runtime. We later define an ordering of $\textit{AllStrict}(M)$—a set of ordered pairs that contains all pairs in $\textit{AllStrict}(M)$ but with an orientation for each pair.

UPC programs must preserve the serial dependencies within each thread, defined by the set of ordered pairs $\textit{DependOnThreads}(M_t)$:

\[\textit{Conflicting}(M) \overset{\text{def}}{=} \{ (m_1, m_2) | \text{Location}(m_1) = \text{Location}(m_2) \land (m_1 \in W(M) \lor m_2 \in W(M)) \}\]

\[\textit{DependOnThreads}(M) \overset{\text{def}}{=} \{ (m_1, m_2) | m_1 \neq m_2 \land \text{Thread}(m_1) = \text{Thread}(m_2) \land \text{Precedes}(m_1, m_2) \land ( (m_1, m_2) \in \textit{Conflicting}(M) \lor (m_1, m_2) \in \textit{StrictOnThreads}(M) ) \}\]

$\textit{DependOnThreads}(M_t)$ establishes an ordering between operations issued by a given thread $t$ that involve a data dependence (i.e. those operations in $\textit{Conflicting}(M_t)$) — this ordering is the one maintained by serial compilers and hardware. $\textit{DependOnThreads}(M_t)$ additionally establishes an ordering between operations appearing in $\textit{StrictOnThreads}(M_t)$. In both cases, the ordering imposed is the one dictated by $\textit{Precedes}(m_1, m_2)$, which intuitively is an ordering relationship defined by serial program order.  

\footnote{DOB: We still need to fill in an appropriate formal definition for $\textit{Precedes}(m_1, m_2)$, which will probably be derived from the relative location of $m_1$ and $m_2$ in the execution trace of the program executing on the abstract machine. Chuck has previously argued that program order depends on the consistency model and defining the latter in terms of the former leads to a circular definition, so we should provide some justification about why we believe this is a valid approach. It may be useful to note that we don’t need to construct the entire set of valid serial execution traces in order to specify $\textit{Precedes}(m_1, m_2)$. So for example, we needn’t try to decide whether or not a given statement in the source program which is control-dependent on some read could possibly be dynamically executed (making such a determination in general requires consulting the memory model, leading to circularity) — the only functionality that $\textit{Precedes}(m_1, m_2)$ requires is the ability to look at two dynamic operations that WERE executed by a single thread (and can be mapped back to the statement which generated them in the source program), and state which of the two operations must come first in a valid serial execution on the abstract machine.}
important to note that \( \text{DependOnThreads}(M_t) \) intentionally avoids introducing ordering constraints between non-conflicting, non-strict operations executed by a single thread (i.e. it does not impose ordering between a thread’s relaxed/local operations to independent memory locations, or between relaxed/local reads to any location). As we shall later see, this allows implementations to freely reorder any consecutive relaxed/local operations issued by a single thread, except for pairs of operations accessing the same location where at least one is a write; by design this is exactly the condition that is enforced by serial compilers/hardware to maintain sequential data dependences – requiring any stronger ordering property would complicate implementations and likely degrade the performance of relaxed/local accesses. The reason this flexibility must be directly exposed in the model (rather than lumped together with other code transformation optimizations generally permitted by UPC Spec 5.1.2.3) is because the results of this reordering may be “visible” to other threads in the UPC program (as we’ll see in later examples) and therefore could impact the program’s “input/output dynamics”.

7 A UPC program execution on \( T \) threads with memory accesses \( M \) is considered \textit{UPC consistent} if there exists a partial order \( <_{\text{Strict}} \) that provides an orientation for each pair in \( \text{AllStrict}(M) \) and for each thread \( t \), there exists a total order \( <_t \) on \( O_t \cup W(M) \cup SR(M) \) (i.e. all operations issued by thread \( t \) and all writes and strict reads issued by any thread) such that:

1. \( <_t \) defines a correct serial execution. \(^{41}\) In particular:
   
   - Each read operation returns the value of the “most recent” preceding write to the same location, where “most recent” is defined by \( <_t \). If there is no prior write of the location in question, the read returns the initial value of the referenced object as defined by ISO C 6.7.8 and 7.2.0.3 \(^{42}\)

\(^{41}\) Note these definitions of \( \text{DependOnThreads}(M) \) and \( <_t \) provide well-defined consistency semantics for local accesses, essentially making them behave as relaxed accesses. Some further thought may be required to determine whether this is the right decision. Defining the interaction between shared and local accesses has been neglected in earlier versions of the memory model, but we feel this is an important issue to tackle in order to have a complete memory model.

\(^{42}\) i.e. the initial value of an object declared with an initializer is the value given by the initializer. Objects with static storage duration lacking an initializer have an initial value of
- The order of operations in $O_t$ is consistent with the ordering dependencies in $\text{DependOnThreads}(M_t)$.

2. $<_t$ is consistent with $<_\text{Strict}$. In particular, this implies that all threads agree on a total order over the strict operations ($\text{Strict}(M)$), and the relative ordering of all pairs of operations issued by a single thread where at least one is strict ($\text{StrictOnThreads}(M)$).

For a UPC consistent execution, we say that the set of $<_t$ orderings that satisfy the above constraints are the enabling orderings for the execution. There must be at least one ordering from each thread in this set.

### B.3 Consistency Semantics of Language and Library Operations

#### B.3.1 Consistency Semantics of Synchronization Operations

UPC has several synchronization operations that can be used to strengthen the consistency requirements of a program. Some of these involve no explicit synchronization variable or object, so we define these in terms of a fresh variable $l_{\text{synch}} \in L$ that does not appear elsewhere in the program. Given this machinery, the memory consistency semantics of the synchronization operations are defined in terms of equivalent memory operations.\(^43\)

---

43 Note: These definitions do not give the synchronization operations their synchronizing effects - they only define the memory model behavior.

44 DOB: We don’t actually care what the reads return (the strict operation in the sync operation is only there to act as a consistency point, and the values read/written are not used by the program in any way), we just want ensure a valid data flow of the dummy values in $<_t$.

In actuality, we don’t even need the written values or $l_{\text{lock}}$ locations to be unique - we could probably just replace all the writes with $\text{SW}(l_{\text{synch}}, 0)$ and the reads with $\text{SR}(l_{\text{synch}}, 0)$. All that’s important is that these sync operations appear as strict operations in $<_\text{Strict}$ and $<_t$, so that they are totally ordered and exert the appropriate effects as barriers to reordering of surrounding memory operations.
• A \textit{upc\_fence} operation is equivalent to a strict write followed by a strict read, $SW(l_{synch}, 0)SR(l_{synch}, 0)$.

• A \textit{upc\_notify} operation implies a strict write, $SW(l_{synch}, 0)$.

• A \textit{upc\_wait} implies a strict read, $SR(l_{synch}, 0)$.

• A \textit{upc\_lock} operation of the form \textit{upc\_lock}(\textit{l\_lock}) (or a successful \textit{upc\_lock\_attempt}(\textit{l\_lock})) implies a strict read $SR(l_{lock}, 0)$, where $l_{lock}$ is a unique location associated with each \textit{upc\_lock\_t} object in the execution.

• An \textit{upc\_unlock} operation of the form \textit{upc\_unlock}(\textit{l\_lock}) implies a strict write, $SW(l_{lock}, 0)$, where $l_{lock}$ is a unique location associated with each \textit{upc\_lock\_t} object in the execution.

These represent a slight relaxation to the current language semantics, which state that \textit{upc\_lock}, \textit{upc\_unlock}, \textit{upc\_notify} and \textit{upc\_wait} all imply a full \textit{upc\_fence}. This relaxation is most relevant if we adopt the asymmetric ordering semantics described in section ??, because it permits more aggressive movement of memory operations past synchronization operations.

### B.3.2 Consistency Semantics of Library Operations

Many of the functions in the UPC standard library can be used to access and modify data in shared objects, either non-collectively (e.g. \textit{upc\_mem\{put, get, cpy\}}) or collectively (e.g. \textit{upc\_all\_broadcast}, etc). It is important to define the consistency semantics of the accesses to shared objects which are implied to take place within the implementation of these library functions, because they may interact with concurrent explicit reads and writes of the same shared objects. For example, an application which calls a function such as \textit{upc\_memcpy} may need to know whether surrounding explicit relaxed operations on non-conflicting shared objects could possibly be reordered relative to the accesses that take place inside the library call. This is a subtle but unavoidable aspect to the library interface which needs to be explicitly defined to ensure that applications can be written with portably deterministic behavior across implementations.

The following sections define the consistency semantics of shared accesses implied by UPC standard library functions, in the absence of any explicit
consistency specification for the given function (which would always take precedence in the case of conflict).

### B.3.2.1 Consistency Semantics of Non-Collective Library Operations

1. For *non-collective* library functions (e.g. `upc_mem{put, get, cpy}`), any implied data accesses to shared objects behave as a set of relaxed shared reads and relaxed shared writes of unspecified size and ordering, issued by the calling thread. No strict operations or fences are implied by a non-collective library function call, unless explicitly noted otherwise.

2. **EXAMPLE 1:**

   ```c
   #include <upc_relaxed.h>
   
   shared int x, y; // initial values are zero
   shared [] int z[2]; // initial values are zero
   int init_z[2] = { -3, -4 };
   ...
   if (MYTHREAD == 0) {
     x = 1;
     upc_memput(z, init_z, 2*sizeof(int));
     y = 2;
   } else {
     #pragma upc strict
     int local_y = y;
     int local_z1 = z[1];
     int local_z0 = z[0];
     int local_x = x;
     ...
   }
   ```

   In this example, all of the writes to shared objects are relaxed (including the accesses implied by the library call), and thread 0 executes no strict operations or fences which would inhibit reordering. Therefore, other threads
which are concurrently performing strict shared reads of the shared objects 
\((x, y, z[0] \text{ and } z[1])\) may observe the updates occurring in any arbitrary order 
that need not correspond to thread 0’s program order. For example, thread 1 
may legally end up with \(local_y == 2\), \(local_z1 == -4\), \(local_z0 == 0\) and \(local_x == 0\), or any other permutation of old and new values for the 
result of the strict shared reads. Furthermore, because the shared writes 
implied by the library call have unspecified size, thread 1 may even read 
intermediate values into \(local_z0\) and \(local_z1\) which correspond to neither 
the initial nor the final values for those shared objects.\(^{45}\) Finally, note that 
all of this remains true even if \(z\) had instead been declared as:

\[
\text{strict shared [] int } z[2];
\]

because the consistency qualification used on the shared object declarator is 
irrelevant to the operation of the library call, whose implied shared accesses 
are specified to always behave as relaxed shared accesses.

If \textit{upc\_fence} operations were inserted in the blank lines immediately preced-
ing and following the \textit{upc\_memput} invocation in the example above, then 
\(<\text{Strict}\) would imply that all reading threads would be guaranteed to observe 
the shared writes according to thread 0’s program order. Specifically, any 
thread reading a non-initial value into \(local_y\) would be guaranteed to read 
the final values for all the other shared reads, and any thread reading the 
initial zero value into \(local_x\) would be guaranteed to also have read the initial 
zero values for all the other shared reads.\(^{46}\) Explicit use of \textit{upc\_fence} 
immediately preceding and following non-collective library calls operating on 
shared objects is the recommended method for ensuring ordering with respect 
to surrounding relaxed operations issued by the calling thread, in the rare 
cases where such ordering guarantees are required for program correctness.

\begin{itemize}
\item \textbf{B.3.2.2 Consistency Semantics of Collective Library Operations}
\end{itemize}

\(^{45}\)This issue is a consequence of the byte-oriented nature of the shared data movement 
functions (which we assume in the absence of further specification) and will remain 
regardless of how we resolve the related but orthogonal issue of write atomicity.

\(^{46}\)However, for threads reading the initial value into \(local_y\) and the final value into 
\(local_x\), the writes to \(z[0]\) and \(z[1]\) could still appear to have been arbitrarily reordered or 
segmented, leading to indeterminate values in \(local_z0\) and \(local_z1\).
For collective functions in the UPC standard library, any implied data accesses to shared objects behave as a set of relaxed shared reads and relaxed shared writes of unspecified size and ordering, issued by one or more unspecified threads (unless explicitly noted otherwise).

For collective functions in the UPC standard library that take a `upc_flag_t` argument (e.g. `upc_all_broadcast`), one or more `upc_fence` operations may be implied upon entry and/or exit to the library call, based on the flags selected in the value of the `upc_flag_t` argument, as follows:

- `UPC_IN_ALLSYNC` and `UPC_IN_MYSYNC` imply a `upc_fence` operation on each calling thread, immediately upon entry to the library function call.

- `UPC_OUT_ALLSYNC` and `UPC_OUT_MYSYNC` imply a `upc_fence` operation on each calling thread, immediately before return from the library function call.

- No fence operations are implied by `UPC_IN_NOSYNC` or `UPC_OUT_NOSYNC`.

The `upc_fence` operations implied by the rules above are sufficient to ensure the results one would naturally expect in the presence of relaxed shared reads and writes issued immediately preceding or following a `ALLSYNC` or `MYSYNC` collective library call that accesses the same shared objects. Without such fences, nothing would prevent prior or subsequent relaxed shared operations issued by the calling thread from being reordered relative to some of the accesses implied by the library call (which might not be issued by the current thread), potentially leading to very surprising and unintuitive results. The `NOSYNC` flag provides no synchronization guarantees between the execution stream of the calling thread and the shared accesses implied by the collective library call, therefore no additional fence operations are required.\(^{47}\)

\(^{47}\)Any deterministic program which makes use of `NOSYNC` collective data movement functions is likely to be synchronizing access to shared objects via other means – for example, through the use of explicit `upc_barrier` or `ALLSYNC/MYSYNC` collective calls that already provide sufficient synchronization and fences.
B.4 Properties Implied by the Specification

1 The memory model definition is fairly subtle in some points, but most programmers need not worry about these details. There are some simple properties that are helpful in understanding the semantics. The first is:

- A UPC program which accesses shared objects using only strict operations \(^{48}\) will be sequentially consistent.

2 This property is trivially true due to the global total order that \(<_{\text{strict}}\) imposes over strict operations (which is respected in every thread’s \(<_t\)), but is not very useful in practice – because a UPC program written entirely with strict accesses is likely to be quite slow. However, it may be a useful debugging tool because even in the presence of data races, a fully strict program is guaranteed to only produce behaviors possible under sequential consistency (which is the easiest memory model to understand and the one which naïve programmers typically assume).

3 Of more interest is that programs free of race conditions will also be sequentially consistent. This requires a more formal definition of race condition, because programmers may believe their program is properly synchronized using memory operations when it is not.

\(^{48}\) i.e. no relaxed shared accesses, and no accesses to shared objects via pointers-to-local

\(^{49}\) A pair of memory accesses \(m_1\) and \(m_2\) from \(M\) are a potential race if:

- \(\text{Location}(m_1) = \text{Location}(m_2) \land \text{Thread}(m_1) \neq \text{Thread}(m_2) \land (m_1 \in W(M) \lor m_2 \in W(M))\)

A UPC consistent execution of a program with memory operations \(M\) is considered race free if forall threads \(t\) and all enabling orderings \(<_t\), if \(m_1 <_t m_2\) then there exists a synchronizing pair of operations \(o_1\) and \(o_2\) such that \(m_1 <_t o_1 <_t o_2 <_t m_2\) and one of the following holds:

- \(o_1\) is a upc_notify operation and \(o_2\) is a upc_wait operation.

- \(o_1\) is a upc_unlock operation and \(o_2\) is a upc_lock operation, both on the same lock variable.

This definition ensures that there is some form of synchronization recognized at the language level between any two pairs of conflicting accesses. The “forall” quantification over the enabling executions is guaranteed to be non-vacuous, since the execution is UPC consistent.

109
We define a set $\text{PotentialRaces}(M)$ as unordered pairs $(m_1, m_2)$:

- $\text{PotentialRaces}(M) \overset{\text{def}}{=} \{(m_1, m_2) | \text{Location}(m_1) = \text{Location}(m_2) \land \text{Thread}(m_1) \neq \text{Thread}(m_2) \land (m_1 \in W(M) \lor m_2 \in W(M))\}$

An execution is race-free if every $(m_1, m_2) \in \text{PotentialRaces}(M)$ is ordered by $<_{\text{Strict}}$, i.e. an execution is race-free if and only if:

- $\forall (m_1, m_2) \in \text{PotentialRaces}(M) : m_1 <_{\text{Strict}} m_2 \lor m_2 <_{\text{Strict}} m_1$.

Note this implies that all threads $t$ and all enabling orderings $<_t$ agree upon the ordering of each $(m_1, m_2) \in \text{PotentialRaces}(M)$ (so there is no race).

These definitions allow us to state a very useful property of UPC programs:

- A program that produces only race-free executions will be sequentially consistent.

Note that UPC locks and barriers constrain $\text{PotentialRaces}$ as one would expect, because these language-level synchronization ops imply strict operations which introduce orderings in $<_{\text{Strict}}$ for the operations in question.

### B.5 Examples

In the figures below, each execution is shown by the linear graph which is the $\text{Precedes}(M)$ program order for each thread, generated by an execution of the source program on the abstract machine. Pairs of memory operations that are ordered by the global ordering over memory operations in $\text{AllStrict}(M)$ (i.e. $m_1 <_{\text{Strict}} m_2$) are shown here as $m_1 \Rightarrow m_2$. All threads must agree on the relative ordering imposed by these edges in their $<_t$ orderings. Pairs ordered by a thread $t$ as in $m_1 <_t m_2$ are represented by $m_1 \rightarrow m_2$.

Arcs that are implied by transitivity are omitted. Assume all variables are initialized to 0.

Note: Programmers may build their own synchronization operations from shared variables using strict operations. The above definition doesn’t handle this, i.e., it would not consider them race-free. This should be fixed.
2 Legal behavior that would not be legal under sequential consistency. There are only relaxed operations, so the threads need not observe the program order by other threads. Because all operations are relaxed, there are no ⇒ orderings between operations.

\[ T_0: \quad RR(x,1); \quad RW(x,2) \]
\[ T_1: \quad RR(x,2); \quad RW(x,1) \]

\[ <_0: \quad RR(x,1) \xrightarrow{} RW(x,2) \quad T_0 \text{ observes } T_1's \text{ write happening before its own read.} \]
\[ RW(x,1) \]

\[ <_1: \quad RW(x,2) \quad T_1 \text{ must observe its own program order for conflicting operations, but it sees } T_0's \text{ write as the first operation.} \]
\[ RR(x,2) \xrightarrow{} RW(x,1) \]

Note that relaxed reads issued by thread \( t \) only appear in the \( <_t \) of that thread.

3 Illegal behavior, which is the same as the previous example, but with all accesses marked strict. All edges in the graph below must therefore be ⇒ edges. This also implies the program order edges must be observed and the two threads must agree on the order of the races. The use of unique values in the writes for this example forces an orientation of the cross-thread edges, so an acyclic \(<_{\text{strict}}\) cannot be defined that satisfies the write-to-read data flow requirements for a legal \(<_t\).

\[ T_0: \quad SR(x,1); \quad SW(x,2) \]
\[ T_1: \quad SR(x,2); \quad SW(x,1) \]

\[ <_{\text{strict}}: \quad SR(x,1) \xrightarrow{} SW(x,2) \quad \text{All of the edges above are required, but this} \]
\[ \quad SR(x,2) \xrightarrow{} SW(x,1) \quad \text{is not a legal } <_{\text{strict}}, \text{ since it contains a cycle.} \]
Legal behavior that would, as in the first example, not be legal if all of the accesses were strict. Again one thread may observe the other’s operations happening out of program order. This is the pattern of memory operations that one might see with a spinlock, where \( y \) is the lock protecting the variable \( x \). The implication is that UPC programmers should not build synchronization out of relaxed operations.

\[
\begin{align*}
T0: & \quad \text{RW}(x, 1); \quad \text{RW}(y, 1) \\
T1: & \quad \text{RR}(y, 1); \quad \text{RR}(x, 0)
\end{align*}
\]

\(<_0: \quad \text{RW}(x, 1) \rightarrow \text{RW}(y, 1) \quad T0 \text{ observes only its own writes.} \\
\text{The writes are non-conflicting, so either ordering constitutes a legal } <_0.
\]

\(<_1: \quad \text{RW}(x, 1) \quad \text{RW}(y, 1) \quad \text{To satisfy write-to-read data flow in } <_1, \\
\text{RW}(x, 1) \text{ must follow RR}(x, 0) \text{ and RR}(y, 1) \text{ must follow } \text{RW}(y, 1). \text{ There are three other legal } <_1 \text{ orderings which satisfy these constraints.}
\]

Legal behavior that would not be legal under sequential consistency. This example is similar to the previous ones, but involves a read-after-write on each processor. Neither thread sees the update by the other, but in the \(<_t\) orderings, each thread conceptually observes the other threads operations happening out of order.

\[
\begin{align*}
T0: & \quad \text{RW}(x, 1); \quad \text{RR}(y, 0) \\
T1: & \quad \text{RW}(y, 1); \quad \text{RR}(x, 0)
\end{align*}
\]

\(<_0: \quad \text{RW}(x, 1) \rightarrow \text{RR}(y, 0) \quad \text{The only constraint on } <_0 \text{ is } \text{RW}(y, 1) \text{ must follow RR}(y, 0). \text{ Several other legal } <_0 \text{ orderings are possible.}
\]

112
<1: \[ \text{RW}(x, 1) \searrow \text{RW}(y, 1) \rightarrow \text{RR}(x, 0) \]

Analogous situation with a write-after-read, this time on x. Several other legal \(<_1\) orderings are possible.

6 **Illegal behavior**, since with strict accesses, one of the two writes must “win” the race condition. Each thread observes the other thread’s write happening after its own write, which creates a cycle when we attempt to construct \(<_{\text{Strict}}\).

\[ T0: \quad \text{SW}(x, 2); \quad \text{SR}(x, 1) \],
\[ T1: \quad \text{SW}(x, 1); \quad \text{SR}(x, 2) \]

\(<_{\text{Strict}}\): \[ \text{SW}(x, 2) \rightarrow \text{SR}(x, 1) \]
\[ \text{SW}(x, 1) \rightarrow \text{SR}(x, 2) \]

7 **Legal behavior**, where a thread observes its own reads occurring out-of-order. Reordering of reads is commonplace in serial compilers/hardware, but in this case an intervening modification by a different thread makes this reordering visible. Strengthening the model to prohibit such reordering of conflicting relaxed reads would impose serious restrictions on the implementation of relaxed reads that would likely degrade performance - for example, under such a model an optimizer could not reorder the reads in this example (or allow them to proceed as concurrent non-blocking operations if they might be reordered in the network) unless it could statically prove the reads were non-conflicting or no other thread was writing the location.

\[ T0: \quad \text{RW}(x, 1); \quad \text{SW}(y, 1); \quad \text{RW}(x, 2) \],
\[ T1: \quad \text{RR}(x, 2); \quad \text{RR}(x, 1) \]

\(<_{\text{Strict}}\): \[ \text{RW}(x, 1) \rightarrow \text{SW}(y, 1) \rightarrow \text{RW}(x, 2) \]

DependOnThreads(\(M_0\)) implies this is the only legal \(<_{\text{Strict}}\) ordering over StrictOnThreads(\(M\))
8 **Illegal behavior**, similar to the previous example, but in this case the addition of a relaxed write on thread 1 introduces dependencies in $DependOnThreads(M_1)$, such that (all else being equal) T1’s second read may only legally return the value 3. If T1’s write were to any location other than $x$, the behavior shown would be legal.

$T0$: $\text{RW}(x,1); \text{SW}(y,1); \text{RW}(x,2)$

$T1$: $\text{RR}(x,2); \text{RW}(x,3); \text{RR}(x,1)$

$<_{\text{strict}}$: $\text{RW}(x,1) \rightarrow \text{SW}(y,1) \rightarrow \text{RW}(x,2)$

$DependOnThreads(M_0)$ implies this is the only legal $<_{\text{strict}}$ ordering over $\text{StrictOnThreads}(M)$

$<_{0}$: $\text{RW}(x,1) \rightarrow \text{SW}(y,1) \rightarrow \text{RW}(x,2)

$<_{0}$ conforms to $<_{\text{strict}}$. Other orderings are possible.

$<_{1}$: $\text{RW}(x,1) \rightarrow \text{SW}(y,1) \rightarrow \text{RW}(x,2)$

This is the only $<_{1}$ that conforms to $<_{\text{strict}}$ and $DependOnThreads(M_1)$. The second read of $x$ cannot return 1 - it must be 3.

9 **Illegal behavior** Demonstrating why strict reads appear in every $<_{1}$, rather than just for the thread that issued them. If the strict reads were absent from $<_{0}$, this behavior would be legal.
\textbf{T0:} \hspace{1em} RW(x,1); \hspace{1em} RW(x,2) \\
\textbf{T1:} \hspace{1em} SR(x,2); \hspace{1em} SR(x,1)

\[ <_{\text{Strict}}: \]
\[ SR(x, 2) \implies SR(x, 1) \]

\[ \text{DependOnThreads}(M_1) \quad \text{implies} \]
\[ \text{this is the only legal } <_{\text{Strict}} \text{ ordering} \]
\[ \text{over } \text{StrictOnThreads}(M) \]

\[ <_0: \]
\[ RW(x, 1) \implies RW(x, 2) \]
\[ SR(x, 2) \implies SR(x, ?) \]

\text{This is the only } <_0 \text{ that} \]
\[ \text{conforms to } <_{\text{Strict}} \text{ and} \]
\[ \text{DependOnThreads}(M_0). \text{ The} \]
\[ \text{second read of } x \text{ cannot return 1 -} \]
\[ \text{it must be 2.} \]

\textbf{10 \quad Legal behavior} \text{ Similar to the previous example, but the writes are no} \]
\[ \text{longer conflicting, and therefore not ordered by DependOnThreads}(M_0). \]

\[ \textbf{T0:} \hspace{1em} RW(x,1); \hspace{1em} RW(y,1) \]
\[ \textbf{T1:} \hspace{1em} SR(y,1); \hspace{1em} SR(x,0) \]

\[ <_{\text{Strict}}: \]
\[ SR(y, 1) \implies SR(x, 0) \]

\[ \text{DependOnThreads}(M_1) \quad \text{implies} \]
\[ \text{this is the only legal } <_{\text{Strict}} \text{ ordering} \]
\[ \text{over } \text{StrictOnThreads}(M) \]

\[ <_0, <_1: \]
\[ RW(x, 1) \quad RW(y, 1) \]
\[ SR(y, 1) \implies SR(x, 0) \]

\text{The writes are non-conflicting,} \]
\[ \text{therefore not ordered by} \]
\[ \text{DependOnThreads}(M_0). \]

\textbf{11 \quad Legal behavior} \text{ Another example of a thread observing its own relaxed} \]
\[ \text{reads out of order, regardless of location accessed.} \]
\[ T0: \quad \text{RW}(x,1); \quad \text{SW}(y,1) \]
\[ T1: \quad \text{RR}(y,1); \quad \text{RR}(x,1); \quad \text{RR}(x,0) \]

\[ \triangleleft_{\text{Strict}}: \quad \text{RW}(x,1) \rightarrow \text{SW}(y,1) \quad \text{DependOnThreads}(M_0) \text{ implies this is the only legal } \triangleleft_{\text{Strict}} \text{ ordering over } \text{StrictOnThreads}(M) \]

\[ \triangleleft_0: \quad \text{RW}(x,1) \rightarrow \text{SW}(y,1) \quad \text{Relaxed reads from thread 1 do not appear in } \triangleleft_0 \]

\[ \triangleleft_1: \quad \text{RW}(x,1) \rightarrow \text{SW}(y,1) \quad \text{Relaxed reads have been re-ordered. Other } \triangleleft_1 \text{ orders are possible.} \]

12 **Illegal behavior** Demonstrating that a barrier synchronization orders relaxed operations as one would expect.

\[ T0: \quad \text{RW}(x,1); \quad \text{upc\_notify}; \quad \text{upc\_wait} \]
\[ T1: \quad \text{upc\_notify}; \quad \text{upc\_wait}; \quad \text{RR}(x,0) \]

\[ \triangleleft_{\text{Strict}}: \quad \text{RW}(x,1) \rightarrow \text{upc\_notify} \rightarrow \text{upc\_wait} \rightarrow \text{RR}(x,0) \quad \text{DependOnThreads}(M) \text{ and the synchronization semantics of barrier imply that } \triangleleft_{\text{Strict}} \text{ must respect all the edges shown (except the edge between the upc\_wait's and the edge between the upc\_notify's, both of which can point either way).} \]

There is no legal \( \triangleleft_1 \) which respects \( \triangleleft_{\text{Strict}} \) - write-to-read data flow along the chain \( \text{RW}(x,1) \Rightarrow \text{upc\_notify} \Rightarrow \text{upc\_wait} \Rightarrow \text{RR}(x,0) \) implies the read must return 1 (i.e. because \( \text{RW}(x,1) \triangleleft_{\text{Strict}} \text{RR}(x,0) \) and there are no intervening writes of x).
Illegal behavior $<_{\text{Strict}}$ is an ordering over the pairs in $\text{AllStrict}(M)$, which includes an edge between two upc_notify operations. Every $<_t$ must conform to a single $<_{\text{Strict}}$ ordering – all threads agree on a single total order over $SR(M) \cup SW(M)$ in general, and in particular they all agree on the order of upc_notify operations. Therefore, at least one of the read operations must return 1.

$T_0$: $\text{RW}(x,1);\text{upc\_notify};\text{RR}(y,0);$ (upc_wait not shown)

$T_1$: $\text{RW}(y,1);\text{upc\_notify};\text{RR}(x,0);$ (upc_wait not shown)

$<_{\text{Strict}}$: 
\[
\text{RW}(x,1) \xrightarrow{\text{upc\_notify}} (= SW*) \xrightarrow{\text{RR}(y,0)} \\
\text{RW}(y,1) \xrightarrow{\text{upc\_notify}} (= SW*) \xrightarrow{\text{RR}(x,0)}
\]

$<_0$: 
\[
\text{RW}(x,1) \xrightarrow{\text{upc\_notify}} (= SW*) \xrightarrow{\text{RR}(y,0)} \\
\text{RW}(y,1) \xrightarrow{\text{upc\_notify}} (= SW*)
\]

$<_1$: 
\[
\text{RW}(x,1) \xrightarrow{\text{upc\_notify}} (= SW*) \xrightarrow{\text{RR}(x,0)} \\
\text{RW}(y,1) \xrightarrow{\text{upc\_notify}} (= SW*) \xrightarrow{\text{RR}(x,0)}
\]

Read cannot return 0.

There is no legal $<_1$ which respects $<_{\text{Strict}}$ – write-to-read data flow along the chain $\text{RW}(x,1) \Rightarrow \text{upc\_notify} \Rightarrow \text{upc\_notify} \Rightarrow \text{RR}(x,0)$ implies the read must return 1 (i.e. because $\text{RW}(x,1) <_{\text{Strict}} \text{RR}(x,0)$ and there are no intervening writes of $x$). Reversing the edge between the upc_notify’s in $<_{\text{Strict}}$ causes the same problem for $y$ in $<_0$. 

117
Note that under the alternate asymmetric semantics proposed in section ??, this behavior would be legal (because one or both of the relaxed reads could be moved earlier than the upc_notify’s).  

CW: The individual upc_notify’s in a single collective synchronization operation are totally ordered. I think this is undesirable, as it enforces synchronization “too early”. Consider the following example:

\begin{verbatim}
T0:   RW(x,1); upc_notify; RW(x,2); RR(x,3)
T1:   RW(x,3); upc_notify; RW(x,4); RR(x,1)
\end{verbatim}

I think this should be allowed, since upc_notify by itself doesn’t imply any synchronization; there’s no need for T0 to be aware of T1’s write, and vice versa. But if the upc_notify’s are ordered, one of the two reads will be disallowed. (DOB: again, this is not a problem under the alternate asymmetric semantics.)