UPC Language Specifications
V1.2

A publication of the UPC Consortium

May 31, 2005

Acknowledgments

1 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: the support and participation of Compaq, Cray, HP, Sun, and CSC; the abundant input of Kevin Harris and Sébastien Chauvin and the efforts of Lauren Smith; and the efforts of Brian Wibecan and Greg Fischer were invaluable in bringing these specifications to version 1.0.

2 Version 1.1 is the result of the contributions of many in the UPC community. In addition to the continued support of all those mentioned above, the efforts of Dan Bonachea were invaluable in this effort.

3 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 that specification. Steve Seidel organized the effort to refine it 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. Version 1.2 also includes the UPC I/O specification which is the result of efforts by Tarek El Ghazawi, Francois Cantonnet, Proshanta Saha, Rajeev 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.

4 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

Introduction

1 UPC is a parallel extension to the C Standard. UPC follows the partitioned global address space [CAG93] programming model. The first version of UPC, known as version 0.9, was published in May of 1999 as technical report [CDC99] 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 initially discussed at the UPC workshop held in Phoenix, AZ, 20 November 2003, and finalized in May 2005.

1 Scope

1 This document focuses only on the UPC specifications that extend the C Standard to an explicit parallel C based on the partitioned global address space 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 C 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

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

3.2

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

3.2.1

1 **shared object**
an object allocated using a shared-qualified declarator or by a library function defined to create shared objects.

2 NOTE All threads may access shared objects.[1]

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

3.2.2

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]

3.2.3

1 **shared array**
an array with elements that have shared qualified type.

3.3

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.

[2]The file scope declaration `int y;` creates a separate object for each thread to access.

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 6.5.1.1 of this specification.

3.6.1.2

1 **strict shared write**
a shared modify access which is determined to be strict according to section 6.5.1.1 of this specification.

3.6.1.3

1 **relaxed shared read**
a shared read access which is determined to be relaxed according to section 6.5.1.1 of this specification.

3.6.1.4

1 **relaxed shared write**
a shared modify access which is determined to be relaxed according to section 6.5.1.1 of this specification.

3.6.2

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

3.7

1 **collective**
a constraint placed on some language operations which requires evaluation

of such operations to be matched[3] across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

3.8

1 **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

1 **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

1 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.

2 If a "shall" or "shall not" requirement is violated, the behavior is undefined. Undefined behavior is indicated by "undefined behavior" or by the omission of any explicit definition of behavior from the UPC specification.

[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 allocate 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` (6.6.1).

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()` function[4].

5.1.2.2 Program termination

1 A program is terminated by the termination of all the threads[5] 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 accesses to shared objects. 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 may be non-compliant.

2 All shared accesses are classified as being either strict or relaxed, as described in sections 6.5.1.1 and 6.7.1. 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.

[4] e.g., in the program `main(){ printf("hello"); }` , each thread prints `hello`.

[5] A barrier is automatically inserted at thread termination.

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 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 [Lam79] 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 any of the 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 NOTE: It is anticipated that 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 B), are guaranteed to behave in a sequentially consistent manner.

Forward references: `upc_fence`, `upc_notify`, `upc_wait`, `upc_barrier` (6.6.1). `upc_lock`, `upc_unlock` (7.2.4).

6 Language

6.1 Notations

1 In the syntax notation used in this section, 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. An optional symbol is indicated by the subscript "opt", so that

$$\{\ expression_{opt}\ \}$$

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 Keywords

1 This subsection provides the UPC extensions of [ISO/IEC00 Sec 6.4.1].

Syntax

2 *upc_keyword:*

MYTHREAD	**upc_barrier**	**upc_localsizeof**
relaxed	**upc_blocksizeof**	**UPC_MAX_BLOCKSIZE**
shared	**upc_elemsizeof**	**upc_notify**
strict	**upc_fence**	**upc_wait**
THREADS	**upc_forall**	

Semantics

3 In addition to the keywords defined in [ISO/IEC00 Sec 6.4.1], the above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords and shall not be otherwise used.

6.3 Predefined identifiers

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

6.3.1 `THREADS`

1 `THREADS` is an expression with 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.

6.3.2 `MYTHREAD`

1 `MYTHREAD` is an expression with a value of type int; it specifies the unique thread index. The range of possible values is `0..THREADS-1`[6].

6.3.3 `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.

6.4 Expressions

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec. 6.5]. In particular, the unary operator expressions in [ISO/IEC00 Sec. 6.5.3] are extended with new syntax.

6.4.1 Unary Operators

[6] e.g., the program `main(){ printf("%d ",MYTHREAD); }` , prints the numbers 0 through THREADS-1, in some order.

[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 translation errors.

Syntax

1 *unary-expression*

...

sizeof *unary-expression*

sizeof (*type-name*)

upc_localsizeof *unary-expression*

upc_localsizeof (*type-name*)

upc_blocksizeof *unary-expression*

upc_blocksizeof (*type-name*)

upc_elemsizeof *unary-expression*

upc_elemsizeof (*type-name*)

6.4.1.1 The sizeof operator

Semantics

1 The `sizeof` operator will result in an integer value which is not constant when applied to a definitely blocked shared array under the *dynamic THREADS* environment.

6.4.1.2 The upc_localsizeof operator

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 Sec. 6.5.3.4] also apply to this operator.

Semantics

2 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 an upper bound of the size

allocated with affinity to any single thread and may include an unspecified amount of padding. The result of `upc_localsizeof` is an integer constant.

3 The type of the result is `size_t`.

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

6.4.1.3 The upc_blocksizeof operator

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 Sec. 6.5.3.4] also apply to this operator.

Semantics

2 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 an integer constant.

3 If the operand of `upc_blocksizeof` has indefinite block size, the value of `upc_blocksizeof` is 0.

4 The type of the result is `size_t`.

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

Forward references: indefinite block size (6.5.1.1).

6.4.1.4 The upc_elemsizeof operator

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 Sec. 6.5.3.4] also apply to this operator.

Semantics

2 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 an integer constant.

3 The type of the result is `size_t`.

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

6.4.2 Pointer-to-shared arithmetic

Constraints

1 No binary operators shall be applied to one pointer-to-shared and one pointer-to-local.

Semantics

2 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:

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

p1 = p + i;
```

3 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]

[8] The C "`%`" and "`/`" operators do not have the necessary properties

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

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

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

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

5 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 ((`(ptrdiff_t) upc_addrfield` (S2) - `(ptrdiff_t) upc_addrfield(S1)`) shall evaluate to the same value as ((P2 - P1) * sizeof(T)).

6 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.

7 When two pointers-to-shared are subtracted, as described in [ISO/IEC00 Sec. 6.5.6], the result is undefined unless there exists an integer x, representable as a `ptrdiff_t`, such that (pts1 + x) == pts2 AND `upc_phaseof(pts1 + x) == upc_phaseof(pts2)`. In this case (pts2 - pts1) evaluates to x.

Forward references: `upc_threadof` (7.2.3.1), `upc_phaseof` (7.2.3.2), `upc_addrfield` (7.2.3.4).

6.4.3 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 *null pointer-to-shared*.[10]

Semantics

2 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 `shared void*`, the *generic pointer-to-shared*, in which case the phase is preserved unchanged in the resulting pointer value.

3 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.

4 If a non-null pointer-to-shared is cast[11] to a pointer-to-local[12] and the affinity of the pointed-to shared object is not to the current thread, the result is undefined.

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

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

7 EXAMPLE 1:

```
int i, *p;
shared int *q;
```

[9] i.e., pointers-to-local cannot be cast to pointers-to-shared.

[10] [ISO/IEC00 Sec. 6.3.2.3/6.5.16.1] imply that an implicit cast is allowed for zero and that all null pointers-to-shared compare equal.

[11] As such pointers are not type compatible, explicit casts are required.

[12] Accesses through such cast pointers are local accesses and behave accordingly.

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

6.4.4 Address operators

Semantics

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

6.5 Declarations

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

Constraints

2 The declaration specifiers in a given declaration shall not include, either directly or through one or more typedefs, both `strict` and `relaxed`.

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

Syntax

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

5 *declaration:*

declaration-specifiers init-declarator-list$_{opt}$ `;`

6 *declaration-specifiers:*

storage-class-specifier declaration-specifiers$_{opt}$

type-specifier declaration-specifiers$_{opt}$

type-qualifier declaration-specifiers$_{opt}$

function-specifier declaration-specifiers$_{opt}$

7 *init-declarator-list:*

init-declarator

init-declarator-list **,** *init-declarator*

8 *init-declarator:*

declarator

declarator = *initializer*

Forward references: strict and relaxed type qualifiers (6.5.1.1).

6.5.1 Type qualifiers

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

Syntax

2 *type-qualifier:*

const

restrict

volatile

shared-type-qualifier

reference-type-qualifier

6.5.1.1 The shared and reference type qualifiers

Syntax

1 *shared-type-qualifier:*

shared $layout\text{-}qualifier_{opt}$

2 *reference-type-qualifier:*

relaxed

strict

3 *layout-qualifier:*

[*constant-expression*$_{opt}$]

[*]

Constraints

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

5 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's referenced type, nor shall it appear in any declarator where prohibited by section 6.5.2.[13]

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

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

Semantics

8 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.

9 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.

10 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 (i.e. `[]`), this indicates an *indefinite blocking factor* where all elements 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.

[13] E.g., `struct S1 { shared char * p1; };` is allowed, while `struct S2 { char * shared p2; };` is not.

11 A layout qualifier which does not specify an indefinite block size is said to specify a *definite block size* .

12 The block size is a part of the type compatibility[14]

13 For purposes of assignment compatibility, generic pointers-to-shared behave as if they always have a compatible block size.

14 If the layout qualifier is of the form '[*]', the shared object is distributed as if it had a block size of

```
( sizeof(a) / upc_elemsizeof(a) + THREADS - 1 ) / THREADS,
```

where 'a' is the array being distributed.

15 EXAMPLE 1: declaration of a shared scalar

```
strict shared int y;
```

`strict shared` is the type qualifier.

16 EXAMPLE 2: automatic storage duration

```
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*/
... }
```

17 EXAMPLE 3: inside a structure

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

Forward references: shared array (6.5.2.1)

[14]This is a powerful statement which allows, for example, that in an implementation `sizeof(shared int *)` may differ from `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.

6.5.2 Declarators

Syntax

1 The following is the declarator definition as per [ISO/IEC00 Sec. 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

type-qualifier-list type-qualifier

Constraints

6 No type qualifier list shall specify more than one block size, either directly or indirectly through one or more typedefs.[15]

[15] While layout qualifiers are most often seen in array or pointer declarators, they are allowed in all declarators. For example, `shared [3] int y` is allowed.

7 No type qualifier list shall include both `strict` and `relaxed` either directly or indirectly through one or more typedefs.

8 No object with automatic storage duration shall have a type that is shared-qualified and no array object with automatic storage duration shall have an element type that is shared-qualified.

Semantics

9 All shared objects created by non-array static declarators have affinity with thread zero.

6.5.2.1 Array declarators

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

Constraints

2 When a UPC program is translated in the *dynamic THREADS* environment and an array with shared-qualified elements is declared with definite block-size, the THREADS expression shall occur exactly once in one dimension of the array declarator (including through typedefs). Further, the THREADS expression shall only occur either alone or when multiplied by an integer constant expression.[16] [17]

Semantics

3 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`).

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

5 For any shared array, `a`, `upc_phaseof (&a)` is zero.

6 EXAMPLE 1: declarations allowed in either *static THREADS* or *dynamic THREADS* translation environments:

[16] In the *static THREADS* environment THREADS is an integer constant expression, and is therefore valid in all dimensions.

[17] This implies the THREADS expression shall not appear anywhere in the declarator of a shared array with indefinite blocksize under the *dynamic THREADS* environment.

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

7 EXAMPLE 2: declarations allowed only in *static THREADS* translation environment:

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

8 EXAMPLE 3: declaration of a shared array

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

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

9 EXAMPLE 4:

```
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:

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

10 EXAMPLE 5:

```
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.

11 EXAMPLE 6:

```
shared [] long *p;
```

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.

6.6 Statements and blocks

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec. 6.8].

Syntax

2 *statement:*

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

synchronization-statement

6.6.1 Barrier statements

Syntax

1 *synchronization-statement:*

upc_notify $expression_{opt}$ **;**

upc_wait $expression_{opt}$ **;**

upc_barrier $expression_{opt}$ **;**

upc_fence ;

Constraints

2 *expression* shall have type `int`.

Semantics

3 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`.[18] A synchronization phase consists of the execution of all statements between the completion of one `upc_wait` and the start of the next.

4 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.[19] `upc_wait` and `upc_notify` are *collective* operations.

5 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.[20]

6 A null strict access is implied before[21] a `upc_notify` statement and after a `upc_wait` statement.[22]

7 The `upc_wait` statement shall interrupt the execution of the program in an implementation defined manner if the value of its expression differs from the value of the expression on the `upc_notify` statement issued by any thread in the current synchronization phase. After such an interruption, subsequent behavior is undefined. No "difference" exists if either statement is missing this optional expression.

8 The `upc_barrier` statement is equivalent to the compound statement[23]:

```
{ upc_notify barrier_value; upc_wait barrier_value; }
```

where `barrier_value` is the result of evaluating *expression* if present, otherwise omitted.

[18] This effectively prohibits issuing any collective operations between a `upc_notify` and a `upc_wait`.

[19] Therefore, all threads are entering the same synchronization phase as they complete the `upc_wait` statement.

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

[21] After the evaluation of *expression*, if present

[22] This 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.

[23] This equivalence is explicit with respect to matching expressions in semantic 7 and collective status in semantic 3.

9 The barrier operations at thread startup and termination have a value of *expression* which is not in the range of the type `int`.[24]

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

```
{ upc_notify; upc_barrier; upc_wait; }
```

as it is equivalent to

```
{ upc_notify; upc_notify; upc_wait; upc_wait; }
```

6.6.2 Iteration statements

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec. 6.8.5].

Syntax

2 *iteration-statement:*

while (*expression*) *statement*

do *statement* **while** (*expression*) ;

for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*

for (*declaration* $expression_{opt}$; $expression_{opt}$) *statement*

upc_forall ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$; $affinity_{opt}$) *statement*

upc_forall (*declaration* $expression_{opt}$; $expression_{opt}$; $affinity_{opt}$) *statement*

3 *affinity:*

expression

continue

Constraints:

[24] These barriers are never expressed in a UPC source program and this semantic says these barrier values can never match one expressed in a user program.

4 The *expression* for affinity shall have pointer-to-shared type or integer type.

Semantics:

5 `upc_forall` is a *collective* operation in which, for each execution of the loop body, the controlling expression and affinity expression are *single-valued*.[25]

6 The *affinity* field specifies the executions of the loop body which are to be performed by a thread.

7 When *affinity* is of pointer-to-shared type, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value of `upc_threadof(`*affinity*`)`. Each iteration of the loop body is executed by precisely one thread.

8 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`.

9 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.

10 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`.

11 Every thread evaluates the first three clauses of a `upc_forall` statement in accordance with the semantics of the corresponding clauses for the `for` statement, as defined in [ISO/IEC00 Sec. 6.8.5.3]. Every thread evaluates the fourth clause of every iteration.

12 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[26], the behavior

[25] Note that single-valued implies that all thread agree on the total number of iterations, their sequence, and which threads execute each iteration.

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

is undefined.

13 If any thread terminates or executes a collective operation 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.[27]

14 EXAMPLE 1: Nested `upc_forall`:

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

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

15 EXAMPLE 2: Evaluation of upc_forall arguments:

```
int i;
upc_forall((foo1(), i=0); (foo2(), i<10); (foo3(), i++); i) {
     foo4(i);
}
```

Each thread evaluates foo1() exactly once, before any further action on that thread. Each thread will execute foo2() and foo3() in alternating sequence, 10 times on each thread. Assuming there is no enclosing upc_forall loop, foo4() will be evaluated exactly 10 times total before the last thread exits the loop, once with each of i=0..9. Evaluations of foo4() may occur on different threads

[27] The `continue` statement behaves as defined in [ISO/IEC00 Sec. 6.8.6.2]; equivalent to a `goto` the end of the loop body.

(as determined by the affinity clause) with no implied synchronization or serialization between foo4() evaluations or controlling expressions on different threads. The final value of i is 10 on all threads.

6.7 Preprocessing directives

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

6.7.1 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_relaxed.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.

6.7.2 Predefined macro names

1 The following macro names shall be defined by the implementation[28]

__UPC__ The integer constant 1, indicating a conforming implementation.

__UPC_VERSION__ The integer constant 200505L.

UPC_MAX_BLOCK_SIZE The integer constant as defined in section 6.3.3.

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.

THREADS The integer constant as defined in section 6.3.1 in the *static THREADS* translation environment.

[28] In addition to these macro names, the semantics of [ISO/IEC00 Sec. 6.10.8] apply to the identifier MYTHREAD.

7 Library

7.1 Standard headers

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

2 The standard headers are

```
<upc_strict.h>
<upc_relaxed.h>
<upc_collective.h>
<upc.h>
```

3 Every inclusion of `<upc_strict.h>` asserts the upc strict pragma and has the effect of including `<upc.h>`.

4 Every inclusion of `<upc_relaxed.h>` asserts the upc relaxed pragma and has the effect of including `<upc.h>`.

5 By convention, all UPC standard library functions are named using the prefix `upc_`. Those which are collective have prefix `upc_all_`.

7.2 UPC utilities <upc.h>

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

7.2.1 Termination of all threads

Synopsis

1
```
#include <upc.h>
 void upc_global_exit(int status);
```

Description

2 `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

1 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);
```

Description

2 The `upc_global_alloc` allocates shared space compatible with the declaration:

```
shared [nbytes] char[nblocks * nbytes].
```

3 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);
```

Description

2 `upc_all_alloc` is a *collective* function with *single-valued* arguments.

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

```
shared [nbytes] char[nblocks * nbytes].
```

4 The `upc_all_alloc` function returns the same pointer value on all threads. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.

5 The dynamic lifetime of an allocated object extends from the time any thread completes the call to `upc_all_alloc` until any thread has deallocated the object.

7.2.2.3 The `upc_alloc` function

Synopsis

1
```
#include <upc.h>
shared void *upc_alloc(size_t nbytes);
```

Description

2 The `upc_alloc` function allocates shared space of at least `nbytes` bytes with affinity to the calling thread.

3 `upc_alloc` is similar to malloc() except that it returns a pointer-to-shared value. It is not a *collective* function. If `nbytes` is zero, the result is a null pointer-to-shared.

7.2.2.4 The `upc_local_alloc` function *deprecated*

Synopsis

1
```
#include <upc.h>
shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
```

Description

2 The `upc_local_alloc` function is deprecated and should not be used. UPC programs should use the `upc_alloc` function instead. Support may be removed in future versions of this specification.

3 The `upc_local_alloc` function allocates shared space of at least `nblocks * nbytes` bytes with affinity to the calling thread. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.

4 `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

1
```
#include <upc.h>
void upc_free(shared void *ptr);
```

Description

2 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,[29] to `upc_free`, the behavior is undefined.

7.2.3 Pointer-to-shared manipulation functions

7.2.3.1 The `upc_threadof` function

Synopsis

1
```
#include <upc.h>
size_t upc_threadof(shared void *ptr);
```

Description

2 The `upc_threadof` function returns the index of the thread that has affinity to the shared object pointed to by `ptr`.[30]

3 If `ptr` is a null pointer-to-shared, the function returns 0.

7.2.3.2 The `upc_phaseof` function

Synopsis

1
```
#include <upc.h>
size_t upc_phaseof(shared void *ptr);
```

Description

2 The `upc_phaseof` function returns the phase component of the pointer-to-shared argument.[31]

3 If `ptr` is a null pointer-to-shared, the function returns 0.

[29] i.e., only one thread may call `upc_free` for each allocation

[30] This function is used in defining the semantics of pointer-to-shared arithmetic in Section 6.4.2

[31] This function is used in defining the semantics of pointer-to-shared arithmetic in Section 6.4.2

7.2.3.3 The `upc_resetphase` function

Synopsis

1
```
#include <upc.h>
shared void *upc_resetphase(shared void *ptr);
```

Description

2 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

2 The `upc_addrfield` function returns an implementation-defined value reflecting the "local address" of the object pointed to by the pointer-to-shared argument.[32]

7.2.3.5 The `upc_affinitysize` function

Synopsis

1
```
#include <upc.h>
size_t upc_affinitysize(size_t totalsize, size_t nbytes,
     size_t threadid);
```

Description

[32] This function is used in defining the semantics of pointer-to-shared arithmetic in Section 6.4.2

2 `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`.

3 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.

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

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

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

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

7.2.4 Lock functions

7.2.4.1 Type

1 The type declared is

```
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 [ISO/IEC00 Sec 6.2.5]). Objects of type `upc_lock_t` may therefore only be manipulated through pointers. Such objects have two states called *locked* and *unlocked*.

3 Two pointers to that reference the same lock object will compare as equal. The results of applying `upc_phaseof()`, `upc_threadof()`, and `upc_addrfield()` to such pointers are undefined.

7.2.4.2 The `upc_global_lock_alloc` function

Synopsis

1

```
#include <upc.h>
upc_lock_t *upc_global_lock_alloc(void);
```

Description

2 The `upc_global_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.

3 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

2 The `upc_all_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.

3 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

2 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`,[33] the behavior is undefined.

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

4 Any subsequent calls to locking functions from any thread using `ptr` have undefined effects. This also applies to any thread currently calling `upc_lock`.

7.2.4.5 The `upc_lock` function

Synopsis

1
```
#include <upc.h>
void upc_lock(upc_lock_t *ptr);
```

Description

2 The `upc_lock` function sets the state of the lock pointed to by *ptr* to locked.

3 If the lock is already in locked state due to the calling thread setting it to locked state, the result is undefined.

4 If the lock is already in locked state, then the calling thread waits for some other thread to set the state to unlocked.[34]

5 Once the lock is in state unlocked, a single calling thread sets the state to locked and the function returns.

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

7.2.4.6 The `upc_lock_attempt` function

Synopsis

1
```
#include <upc.h>
int upc_lock_attempt(upc_lock_t *ptr);
```

[33] i.e., only one thread may call `upc_lock_free` for each allocation

[34] If no other thread calls `upc_unlock` on *ptr* the calling thread will never return from this function.

Description

2 The `upc_lock_attempt` function attempts to set the state of the lock pointed to by *ptr* to locked.

3 If the lock is already in locked state due to the calling thread setting it to locked state, the result is undefined.

4 If the lock is already in locked state the function returns 0.

5 If the lock is in state unlocked, a single calling thread sets the state to locked and the function returns 1.

6 A null strict access is implied after a call to `upc_lock_attempt()` that returns 1.

7.2.4.7 The `upc_unlock` function

Synopsis

1
```
#include <upc.h>
void upc_unlock(upc_lock_t *ptr);
```

Description

2 The `upc_unlock` function sets the state of the lock pointed to by *ptr* to unlocked.

3 Unless the lock is in locked state and the calling thread is the locking thread, the result is undefined.

4 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 * restrict dst,
```

```
        shared const void * restrict src, size_t n);
```

Description

2 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.

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

```
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

1
```
#include <upc.h>
void upc_memget(void * restrict dst,
     shared const void * restrict src, size_t n);
```

Description

2 The `upc_memget` function copies `n` characters from a shared object with affinity to any single thread to an object on the calling thread.

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

```
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

```
char[n]
```

7.2.5.3 The `upc_memput` function

Synopsis

1
```
#include <upc.h>
void upc_memput(shared void * restrict dst,
     const void * restrict src, size_t n);
```

Description

2 The `upc_memput` function copies `n` characters from an object on the calling thread to a shared object with affinity to any single thread.

3 The `upc_memput` function is equivalent to copying the entire contents from an array object (the `src` array) declared with the type

```
char[n]
```

to a shared array object (the `dst` array) with the type

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

7.2.5.4 The `upc_memset` function

Synopsis

1
```
#include <upc.h>
void upc_memset(shared void *dst, int c, size_t n);
```

Description

2 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`.

3 The `upc_memset` function treats the `dst` pointer as if had type:

```
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`.

7.2.6 Memory Semantics of Library Functions

1 `upc_flag_t` is an integral type defined in `<upc.h>` which is used to control the data synchronization semantics of certain collective UPC library functions. Values of function arguments having type `upc_flag_t` are formed by or-ing together a constant of the form `UPC_IN_`*X*`SYNC` and a constant of the form `UPC_OUT_`*Y*`SYNC`, where *X* and *Y* may be `NO`, `MY`, or `ALL`.

2 If an argument of type `upc_flag_t` has value `(UPC_IN_`*X*`SYNC | UPC_OUT_`*Y*`SYNC)`, then if *X* is

`NO` the function may begin to read or write data when the first thread has entered the collective 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 function call[35]

3 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[36], and

`ALL` the function call may return only after all reads and writes of data are complete.[37]

---

[35] `UPC_IN_ALLSYNC` requires the function to guarantee that after all threads have entered the function call all threads will read the same values of the input data.

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

[37] `UPC_OUT_ALLSYNC` requires the collective function to guarantee that after a thread returns from the 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 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.

4 Passing `UPC_IN_`*`X`*`SYNC` alone has the same effect as `(UPC_IN_`*`X`*`SYNC | UPC_OUT_ALLSYNC)`, passing `UPC_OUT_`*`X`*`SYNC` alone has the same effect as `(UPC_IN_ALLSYNC | UPC_OUT_`*`X`*`SYNC)`, and passing 0 has the same effect as `(UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC)`, where *`X`* is `NO`, `MY`, or `ALL`.

5 Each of the six flags `UPC_{IN,OUT}_{NO,MY,ALL}SYNC` are macros which expand to integer constant expressions. The expressions are defined such that bitwise ORs of all combinations of the macros result in distinct positive values less than 64.

7.3 UPC Collective Utilities <upc_collective.h>

1 Implementations that support this interface shall predefine the feature macro __UPC_COLLECTIVE__ to the value 1.

2 The following requirements apply to all of the functions defined in this section.

3 All of the functions are collective.

4 All collective function arguments are single-valued.

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

6 The standard header is

<upc_collective.h>

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 * restrict dst,
     shared const void * restrict src, size_t nbytes,
     upc_flag_t flags);
```

Description

2 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 bytes in each block is nbytes.

3 nbytes must be strictly greater than 0.

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

```
shared [] char[nbytes]
```

5 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:

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

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

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

8 EXAMPLE 1 shows `upc_all_broadcast`

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

9 EXAMPLE 2:

```
#include <upc.h>
#include <upc_collective.h>
#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 );
```

10 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])`.

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

1

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

Description

2 The `upc_all_scatter` function copies the ith block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the ith thread. The number of bytes in each block is `nbytes`.

3 `nbytes` must be strictly greater than 0.

4 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]
```

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

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

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

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

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

9 EXAMPLE 1 `upc_all_scatter` for the dynamic THREADS translation environment.

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

10 EXAMPLE 2 `upc_all_scatter` for the *static THREADS* translation environment.

```
#include <upc.h>
#include <upc_collective.h>
#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 );
```

7.3.1.3 The `upc_all_gather` function

Synopsis

1
```
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather(shared void * restrict dst,
     shared const void * restrict src, size_t nbytes,
     upc_flag_t flags);
```

Description

2 The `upc_all_gather` function copies a block of shared memory that has affinity to the ith thread to the ith block of a shared memory area that has affinity to a single thread. The number of bytes in each block is `nbytes`.

3 `nbytes` must be strictly greater than 0.

4 The `upc_all_gather` function treats the `src` pointer as if it pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type:

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

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

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

6 The target of the `src` pointer must have affinity to thread 0.

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

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

9 EXAMPLE 1 `upc_all_gather` for the *static THREADS* translation environment.

```
#include <upc.h>
#include <upc_collective.h>
#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 );
```

10 EXAMPLE 2 `upc_all_gather` for the *dynamic THREADS* translation environment.

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

1
```
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather_all(shared void * restrict dst,
     shared const void * restrict src, size_t nbytes,
     upc_flag_t flags);
```

Description

2 The `upc_all_gather_all` function copies a block of memory from one shared memory area with affinity to the ith thread to the ith block of a shared memory area on each thread. The number of bytes in each block is `nbytes`.

3 `nbytes` must be strictly greater than 0.

4 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:

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

5 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]
```

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

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

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

9 EXAMPLE 1 `upc_all_gather_all` for the *static THREADS* translation environment.

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

10 EXAMPLE 2 `upc_all_gather_all` for the *dynamic THREADS* translation environment.

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

1
```
#include <upc.h>
#include <upc_collective.h>
void upc_all_exchange(shared void * restrict dst,
     shared const void * restrict src, size_t nbytes,
     upc_flag_t flags);
```

Description

2 The `upc_all_exchange` function copies the ith block of memory from a shared memory area that has affinity to thread j to the jth block of a shared memory area that has affinity to thread i. The number of bytes in each block is `nbytes`.

3 `nbytes` must be strictly greater than 0.

4 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]
```

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

8 EXAMPLE 1 `upc_all_exchange` for the *static THREADS* translation environment.

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

9 EXAMPLE 2 `upc_all_exchange` for the *dynamic THREADS* translation environment.

```
#include <upc.h>
#include <upc_collective.h>
#define NELEMS 10
shared int *Adata, *Bdata;
shared [] int *myA, *myB;
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 * restrict dst,
    shared const void * restrict src,
    shared const int * restrict perm,
    size_t nbytes, upc_flag_t flags);
```

Description

2 The `upc_all_permute` function copies a block of memory from a shared memory area that has affinity to the ith thread to a block of a shared memory that has affinity to thread `perm[i]`. The number of bytes in each block is `nbytes`.

3 `nbytes` must be strictly greater than 0.

4 `perm[0..THREADS−1]` must contain `THREADS` distinct values: 0, 1, ..., `THREADS`−1.

5 The `upc_all_permute` function treats the `src` pointer and the `dst` pointer

as if each pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type:

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

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

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

8 The effect is equivalent to copying the block of `nbytes` bytes that has affinity to thread i pointed to by `src` to the block of `nbytes` bytes that has affinity to thread `perm`[i] pointed to by `dst`.

9 EXAMPLE 1 `upc_all_permute`.

```
#include <upc.h>
#include <upc_collective.h>
#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 non-commutative 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.[38]

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, prefix reduction (7.3.2.1).

7.3.2.1 The `upc_all_reduce` and `upc_all_prefix_reduce` functions

Synopsis

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

[38] 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.

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

Description

2 The function prototypes above represents the 22 variations of the `upc_all_reduce`*T* and `upc_all_prefix_reduce`*T* functions where *T* and *TYPE* have the following correspondences: [39]

T	*TYPE*	*T*	*TYPE*
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

3 On completion of the `upc_all_reduce` variants, the value of the *TYPE* shared object referenced by `dst` is `src[0]` $\oplus$ `src[1]` $\oplus \cdots \oplus$ `src[nelems-1]` where "$\oplus$" is the operator specified by the variable `op`.

4 On completion of the `upc_all_prefix_reduce` variants, the value of the *TYPE* shared object referenced by `dst[i]` is `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`.

5 If the value of `blk_size` passed to these functions is greater than 0 then they treat 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]
```

[39] For example, if *T* is `C`, then *TYPE* must be `signed char`.

6 If the value of `blk_size` passed to these functions is 0 then they treat the `src` pointer as if it pointed to a shared memory area of `nelems` elements of type *TYPE* with an indefinite layout qualifier, and therefore had type[40]:

```
shared []  TYPE[nelems]
```

7 The phase of the `src` pointer is respected when referencing array elements, as specified above.

8 `upc_all_prefix_reduce`*T* treats the `dst` pointer equivalently to the `src` pointer as described in the past 3 paragraphs.

9 `upc_all_prefix_reduce`*T* requires the affinity and phase of the `src` and `dst` pointers to match – ie. `upc_threadof(src) == upc_threadof(dst) && upc_phaseof(src) == upc_phaseof(dst)`. 10 `upc_all_reduce`*T* treats the `dst` pointer as having type:

```
shared TYPE *
```

11 EXAMPLE 1 `upc_all_reduce` of type `long UPC_ADD`.

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

12 EXAMPLE 2 `upc_all_prefix_reduce` of type `long UPC_ADD`.

[40] 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`.

```
#include <upc.h>
#include <upc_collective.h>
#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 );
```

A Proposed Additions and Extensions

1 This section contains proposed additions and extensions to the UPC specification. Such proposals are included when stable enough for developers to implement and for users to study and experiment with them. However, their presence does not suggest long term support. When fully stable and tested, they will be moved to the main body of the specification.

2 This section also describes the process used to add new items to the specification, which starts with inclusion in this section. Requirements for inclusion are:[41]

1. A documented API which shall use the format and conventions of this specification and [ISO/IEC00].
2. Either a complete, publicly available, implementation of the API or a set of publicly available example programs which demonstrate the interface.
3. The concurrence of the UPC consortium that its inclusion would be in the best interest of the language.

3 If all implementations drop support for an extension and/or all interested parties no longer believe the extension is worth pursuing, then it may simply be dropped. Otherwise, the requirements for inclusion of an extension in the main body of the specification are:

1. Six months residence in this section.
2. The existence of either one (or more) publicly available "reference" implementation written in standard UPC OR at least two independent implementations (possibly specific to a given UPC implementation).

[41] These requirements ensure that most of the semantic issues that arise during initial implementation have been addressed and prevents the accumulation of interfaces that no one commits to implement. Nothing prevents the circulation of more informal *what if* interface proposals from circulating in the community before an extension reaches this point.

3. The existence of a significant base of experimental user experience which demonstrates positive results with a substantial portion of the proposed API.

4. The concurrence of the UPC consortium that its inclusion would be in the best interest of the language.

4 For each extension, there shall be a predefined *feature macro* beginning with `__UPC` which will be defined by an implementation to be the interface version of the extension if it is supported, otherwise undefined.

A.1 UPC Parallel I/O <upc_io.h>

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec 7.19]. All the characteristics of library functions described in [ISO/IEC00 Sec 7.1.4] apply to these as well. Implementations that support this interface shall predefine the feature macro `__UPC_IO__` to the value 1.

Common Constraints

2 All UPC-IO functions are collective and must be called by all threads collectively.[42]

3 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.

4 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.

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

6 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.

7 All generic pointers-to-shared 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).

8 All UPC-IO read/write functions take an argument `flags` of type `upc_flag_t`. The semantics of this argument is defined in Section 7.2.6. These semantics apply only to memory locations in user-provided buffers, not to the

[42] Note that collective does not necessarily imply barrier synchronization. The synchronization behavior of the UPC-IO data movement library functions is explicitly controlled by using the `flags` flag argument. See Section 7.2.6 for details.

read/write operations on the storage medium or any buffer memory internal to the library implementation.

9 The `flags` 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 `flags` flag provides no useful additional guarantees and is recommended to be passed as `UPC_IN_NOSYNC|UPC_OUT_NOSYNC` to maximize performance.

10 The arguments to all UPC-IO functions are single-valued except where explicitly noted in the function description.

11 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.

12 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.

13 Byte-level data consistency is supported.

14 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

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

16 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.

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

18 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 `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.

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

20 UPC-IO functions can not be called between `upc_notify` and corresponding `upc_wait` statements.

A.1.1 Background

A.1.1.1 File Accessing and File Pointers

1 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 `upc_all_fopen` function and can be changed using `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

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

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 `upc_all_fopen`, or the common file pointer by passing the flag `UPC_COMMON_FP` to `upc_all_fopen`.

```
#include <upc.h>
#include <upc_io.h>
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);
/* or equivalently:
 *  upc_all_fread_shared( fd, buffer, 5, sizeof(float), 20,
                                  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
 */
```

A.1.1.2 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.

2 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_ftest_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.

A.1.1.3 Consistency and Atomicity Semantics

1 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`.

2 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_SEMANTICS`, 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 regions 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 A.1.6). While strong consistency and atomicity semantics are selected on a given file handle, the `flags` 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 A.1.6).

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 I/O function.

7 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}
```

8 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
```

9 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.

10 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  2
```

11 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.

A.1.1.4 File Interoperability

1 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.

A.1.2 Predefined Types

1 The following types are defined in `<upc_io.h>`

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

3 `upc_file_t` is an opaque shared data type of incomplete type (as defined in [ISO/IEC00 Sec 6.2.5]) that represents an open file handle.

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

5 `upc_file_t` is a shared data type. It is allowed 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

6 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:

```
#include <upc.h>
#include <upc_io.h>
/* 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;
}
```

7 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.

A.1.3 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.

3 Common file pointers cannot be used in conjunction with pointers-to-local (and hence cannot operate on private objects).

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

A.1.3.1 The `upc_all_fopen` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_file_t *upc_all_fopen(const char *fname, int flags,
     size_t numhints, struct upc_hint const *hints);
```

Description

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

3 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. If the named file does not exist and this flag is not passed, the function fails with an error.

`UPC_EXCL` Must be 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. The file must be opened before writing.

`UPC_DELETE_ON_CLOSE` Delete the file automatically on close

4 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.

5 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.

6 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, anonymous thread which is different from any compute thread (and different for each operation)."[43]

7 Hints can be passed to the UPC-IO library as an array of key-value pairs[44] 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. The following type is defined in `<upc_io.h>`:

```
struct upc_hint
```

holds each element of the `hints` array and contain at least the following initial members, in this order.

```
const char *key;
const char *value;
```

8 `upc_all_fopen` defines a number 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. All

[43] 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.

[44] The contents of the key/value pairs passed by all the threads must be single-valued.

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. Allowed 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 allowed 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.

9 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.

10 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.

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

12 `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.

A.1.3.2 The `upc_all_fclose` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fclose (upc_file_t *fd);
```

Description

2 `upc_all_fclose` executes an implicit `upc_all_fsync` on `fd` and then closes the file associated with `fd`.

3 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.

4 `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.

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

A.1.3.3 The `upc_all_fsync` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fsync(upc_file_t *fd);
```

Description

2 `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).

3 There is an implied barrier immediately before `upc_all_fsync` returns.

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

A.1.3.4 The `upc_all_fseek` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fseek(upc_file_t *fd, upc_off_t offset,
     int origin);
```

Description

2 `upc_all_fseek` sets the current position of the file pointer associated with `fd`.

3 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.

4 The origin parameter can be specified as `UPC_SEEK_SET`, `UPC_SEEK_CUR`, or `UPC_SEEK_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.

5 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.

6 It is allowed 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 A.1.3 for more details.

7 The current position of the file pointer can be determined by calling `upc_all_fseek(fd, 0, UPC_SEEK_CUR)`.

8 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.

A.1.3.5 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

2 `upc_all_fset_size` executes an implicit `upc_all_fsync` on `fd` and resizes the file associated with `fd`. The file handle must be open for writing.

3 `size` is measured in bytes from the beginning of the file.

4 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.

5 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.

6 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 allowed and is equivalent to seeking past the end of file (see the Common Constraints in Section A.1.3 for the semantics of seeking past the end of file).

7 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.

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

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

A.1.3.6 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

2 `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.

A.1.3.7 The `upc_all_fpreallocate` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fpreallocate(upc_file_t *fd, upc_off_t size);
```

Description

2 `upc_all_fpreallocate` ensures that storage space is allocated for the first `size` bytes of the file associated with `fd`. The file handle must be open for writing.

3 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.

4 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.

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

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

A.1.3.8 The `upc_all_fcntl` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fcntl(upc_file_t *fd,  int cmd, void *arg);
```

Description

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

`UPC_GET_CA_SEMANTICS` Get the current consistency and atomicity semantics for `fd`. The argument `arg` is ignored. The return value is `UPC_STRONG_CA` for strong consistency and atomicity semantics and 0 for the default weak consistency and atomicity semantics.

`UPC_SET_WEAK_CA_SEMANTICS` Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the weak consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_SET_STRONG_CA_SEMANTICS` Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the strong consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_GET_FP` Get the type of the current file pointer for `fd`. The argument `arg` is ignored. The return value is either `UPC_COMMON_FP` in case of a common file pointer, or `UPC_INDIVIDUAL_FP` for individual file pointers.

`UPC_SET_COMMON_FP` Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `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 `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_SET_INDIVIDUAL_FP` Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `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 `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `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 filename 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_fnctl` 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.

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

4 It *is* allowed to call `upc_all_fcntl(UPC_ASYNC_OUTSTANDING)` when an asynchronous operation is outstanding (but it is still disallowed to call `upc_all_fcntl` with any other argument when an asynchronous operation is outstanding).

A.1.4 Reading Data

Common Constraints

1 No function in this section A.1.4 may be called while an asynchronous operation is pending on the file handle.

A.1.4.1 The `upc_all_fread_local` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fread_local(upc_file_t *fd, void *buffer,
     size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_local` reads data from a file into a local buffer on each thread.

3 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.

4 `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.

5 Each thread reads `(size*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.

6 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.

A.1.4.2 The `upc_all_fread_shared` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fread_shared(upc_file_t *fd,
     shared void *buffer, size_t blocksize, size_t size,
     size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_shared` reads data from a file into a shared buffer in memory.

3 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.

4 `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.

5 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.

6 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.

7 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 A.1.3.1).

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

A.1.5 Writing Data

Common Constraints

1 No function in this section A.1.5 may be called while an asynchronous operation is pending on the file handle.

A.1.5.1 The `upc_all_fwrite_local` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fwrite_local(upc_file_t *fd, void *buffer,
     size_t size, size_t nmemb, upc_flag_t flags);
```

Description

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

3 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.

4 `buffer` is a pointer to an array from which data will be written, and each thread may pass a different value for `buffer`.

5 Each thread writes `(size*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 `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.

6 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 `fd` (see A.1.3.1).

7 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 `errno` appropriately.

A.1.5.2 The `upc_all_fwrite_shared` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fwrite_shared(upc_file_t *fd,
```

```
shared void *buffer, size_t blocksize,  size_t size,
size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_shared` writes data from a shared buffer in memory to a file.

3 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.

4 `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. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.

5 In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `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.

6 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.

7 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 A.1.3.1).

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

A.1.6 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 Types declared in `<upc_io.h>`are

```
struct upc_local_memvec
```

which contains at least the initial members, in this order:

```
void *baseaddr;
size_t len;
```

and is a memory vector element pointing to a contiguous region of local memory.

5 `struct upc_shared_memvec`

which contains at least the initial members, in this order:

```
shared void *baseaddr;
size_t blocksize;
size_t len;
```

and is a memory vector element pointing to a blocked region of shared memory.

6 `struct upc_filevec`

which contains at least the initial members, in this order:

```
upc_off_t offset;
size_t len;
```

and is a file vector element pointing to a contiguous region of a file.

For all cases these vector element types specify regions which are `len` bytes long. If `len` is zero, the entry is ignored. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

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

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

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

10 The offsets passed in `filevec` must be in monotonically non-decreasing order.

11 No function in this section (A.1.6) may be called while an asynchronous operation is pending on the file handle.

12 No function in this section (A.1.6) 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.

13 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.

14 EXAMPLE 1: 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.

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

```
#include <upc_io.h>
char buffer[12];
struct upc_local_memvec memvec[2] = {(&buffer[0],4},{&buffer[7],3}};
struct upc_filevec filevec[2];
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
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);
```

A.1.6.1 The `upc_all_fread_list_local` function

Synopsis

1

```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fread_list_local(upc_file_t *fd,
     size_t memvec_entries, struct upc_local_memvec const *memvec,
     size_t filevec_entries, struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

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

3 `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.

4 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`.

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

A.1.6.2 The `upc_all_fread_list_shared` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fread_list_shared(upc_file_t *fd,
     size_t memvec_entries, struct upc_shared_memvec const *memvec,
     size_t filevec_entries, struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

2 `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.

3 `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.

4 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`.

5 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 A.1.3.1).

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

A.1.6.3 The `upc_all_fwrite_list_local` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fwrite_list_local(upc_file_t *fd,
     size_t memvec_entries, struct upc_local_memvec const *memvec,
     size_t filevec_entries, struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

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

3 `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.

4 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`.

5 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 A.1.3.1).

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

A.1.6.4 The `upc_all_fwrite_list_shared` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fwrite_list_shared(upc_file_t *fd,
```

```
size_t memvec_entries, struct upc_shared_memvec const *memvec,
size_t filevec_entries, struct upc_filevec const *filevec,
upc_flag_t flags);
```

Description

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

3 `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.

4 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`.

5 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 A.1.3.1).

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

A.1.7 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.

5 The semantics of the flags of type `upc_flag_t` when applied to the async variants of the fread/fwrite functions should be interpreted as follows: constraints that 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 the flags which govern an asynchronous operation are passed to the library during the asynchronous initiation call.

A.1.7.1 The `upc_all_fread_local_async` function

Synopsis

1
```
#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 flags);
```

Description

2 `upc_all_fread_local_async` initiates an asynchronous read from a file into a local buffer on each thread.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_local`.

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

A.1.7.2 The `upc_all_fread_shared_async` function

Synopsis

1
```
#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 flags);
```

Description

2 `upc_all_fread_shared_async` initiates an asynchronous read from a file into a shared buffer.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_shared`.

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

A.1.7.3 The `upc_all_fwrite_local_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fwrite_local_async(upc_file_t *fd, void *buffer,
     size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_local_async` initiates an asynchronous write from a local buffer on each thread to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_local`.

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

A.1.7.4 The `upc_all_fwrite_shared_async` function

Synopsis

1
```
#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 flags);
```

Description

2 `upc_all_fwrite_shared_async` initiates an asynchronous write from a shared buffer to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_shared`.

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

A.1.7.5 The `upc_all_fread_list_local_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fread_list_local_async(upc_file_t *fd,
     size_t memvec_entries, struct upc_local_memvec const *memvec,
     size_t filevec_entries,  struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

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

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

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

A.1.7.6 The upc_all_fread_list_shared_async function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fread_list_shared_async(upc_file_t *fd,
     size_t memvec_entries, struct upc_shared_memvec const *memvec,
     size_t filevec_entries,  struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

2 upc_all_fread_list_shared_async initiates an asynchronous read of data from a file into various locations of a shared buffer in memory.

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

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

A.1.7.7 The upc_all_fwrite_list_local_async function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fwrite_list_local_async(upc_file_t *fd,
     size_t memvec_entries, struct upc_local_memvec const *memvec,
     size_t filevec_entries, struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

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

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

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

A.1.7.8 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, struct upc_shared_memvec const *memvec,
     size_t filevec_entries, struct upc_filevec const *filevec,
     upc_flag_t flags);
```

Description

2 `upc_all_fwrite_list_shared_async` initiates an asynchronous write of data from various locations of a shared buffer in memory to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_list_shared`.

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

A.1.7.9 The `upc_all_fwait_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fwait_async(upc_file_t *fd)
```

Description

2 `upc_all_fwait_async` completes the previously issued asynchronous I/O operation on the file handle `fd`, blocking if necessary.

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

4 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.

A.1.7.10 The `upc_all_ftest_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_ftest_async(upc_file_t *fd, int *flag)
```

Description

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

3 If the operation has completed, the function sets `flag=1` and the asynchronous operation becomes no longer outstanding;[45] otherwise it sets `flag=0`. The same value of `flag` is set on all threads.

4 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.

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

[45] This implies it is disallowed to call `upc_all_fwait_async` or `upc_all_ftest_async` immediately after a successful `upc_all_ftest_async` on that file handle.

B Formal UPC Memory Consistency 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 in general 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 a naïve translation of the program, i.e., without any program transformations during translation, where each thread issues operations as defined by the abstract machine defined in [ISO/IEC00 Sec. 5.1.2.3]. [ISO/IEC00 Sec. 5.1.2.3] allows a UPC implementation to perform various program transformations to improve performance, provided they are not visible to the programmer - specifically, provided those transformations do not affect the external behavior of the program. UPC extends this constraint, requiring the set of externally-visible behaviors (the input/output dynamics and volatile behavior defined in [ISO/IEC00 Sec. 5.1.2.3]) from any execution of the transformed program be indistinguishable from those of the original program executing on the abstract machine and adhering to the memory consistency model as defined in this appendix.

3 This appendix assumes some familiarity with memory consistency models, partial orders, and basic set theory.

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.

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, as defined in Section 3:

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

3 In addition, each memory operation m is associated with exactly one of the T threads, denoted $Thread(m)$, and the accessor $Location(m)$ is defined to return the location l accessed by m.

4 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

5 The set of all writes in M is denoted as $W(M)$:

$$W(M) \stackrel{def}{=} SW(M) \cup RW(M) \cup LW(M)$$

and the set of all strict accesses in M is denoted as $Strict(M)$:

$$Strict(M) \stackrel{def}{=} SR(M) \cup SW(M)$$

B.2 Memory Access Model

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

$$StrictPairs(M) \stackrel{def}{=} \left\{ (m_1, m_2) \;\middle|\; \begin{array}{l} m_1 \neq m_2 \;\wedge\; m_1 \in Strict(M) \;\wedge \\ m_2 \in Strict(M) \end{array} \right\}$$

$$StrictOnThreads(M) \stackrel{def}{=} \left\{ (m_1, m_2) \;\middle|\; \begin{array}{l} m_1 \neq m_2 \;\wedge \\ Thread(m_1) = Thread(m_2) \;\wedge \\ (\; m_1 \in Strict(M) \;\vee\; m_2 \in Strict(M) \;) \end{array} \right\}$$

$$AllStrict(M) \stackrel{def}{=} StrictPairs(M) \;\cup\; StrictOnThreads(M)$$

2 Thus, $StrictPairs(M)$ is the set of all pairs of strict memory accesses, including those between threads, and $StrictOnThreads(M)$ is the set of all pairs of memory accesses from the same thread in which at least one is strict. $AllStrict(M)$ is their union, which intuitively is the set of operation pairs for which all threads must agree upon a unique ordering (i.e. all threads must agree on the directionality of each pair). In general, the determination of that ordering will depend on the resolution of race conditions at runtime.

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

$$Conflicting(M) \stackrel{def}{=} \left\{ (m_1, m_2) \;\middle|\; \begin{array}{l} Location(m_1) = Location(m_2) \;\wedge \\ (\; m_1 \in W(M) \;\vee\; m_2 \in W(M) \;) \end{array} \right\}$$

$$DependOnThreads(M) \stackrel{def}{=} \left\{ \langle m_1, m_2 \rangle \;\middle|\; \begin{array}{l} m_1 \neq m_2 \;\wedge \\ Thread(m_1) = Thread(m_2) \;\wedge \\ Precedes(m_1, m_2) \;\wedge \\ \left(\begin{array}{l} (m_1, m_2) \in Conflicting(M) \;\vee \\ (m_1, m_2) \in StrictOnThreads(M) \end{array} \right) \end{array} \right\}$$

4 $DependOnThreads(M_t)$ establishes an ordering between operations issued by a given thread t that involve a data dependence (i.e. those operations in $Conflicting(M_t)$) – this ordering is the one maintained by serial compilers and hardware. $DependOnThreads(M_t)$ additionally establishes an ordering between operations appearing in $StrictOnThreads(M_t)$. In both cases, the

ordering imposed is the one dictated by $Precedes(m_1, m_2)$, a predicate which intuitively is an ordering relationship defined by serial program order.[46] It's important to note that $DependOnThreads(M_t)$ 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 demonstrated in Section B.5, 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 and 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 (unlike other program transformation optimizations which are implicitly permitted by [ISO/IEC00 Sec. 5.1.2.3]) is because the results of this reordering may be "visible" to other threads in the UPC program (as demonstrated in Section B.5) and therefore could impact the program's "input/output dynamics".

5 A UPC program execution on T threads with memory accesses M is considered *UPC consistent* if there exists a partial order $<_{Strict}$ that provides an orientation for each pair in $AllStrict(M)$ 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. 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/IEC00 Sec. 6.7.8/7.2.0.3].[47]

[46] The formal definition of *Precedes* is given in Section B.6.

[47] 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 zero. Objects with automatic storage duration lacking an initializer have an indeterminate (but fixed) initial value. The initial value for a dynamically allocated object is described by the memory allocation function used to create the object.

- The order of operations in O_t is consistent with the ordering dependencies in $DependOnThreads(M_t)$.

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

6 The set of $<_t$ orderings that satisfy the above constraints are said to be the *enabling orderings* for the execution. An execution is UPC consistent if each UPC thread has at least one such enabling ordering in this set. Conformant UPC implementations shall only produce UPC consistent executions.

7 The definitions of $DependOnThreads(M)$ and $<_t$ provide well-defined consistency semantics for local accesses to shared objects, making them behave similarly to relaxed shared accesses. Note that private objects by definition may only be accessed by a single thread, and therefore local accesses to private objects trivially satisfy the constraints of the model – provided the serial data dependencies across sequence points mandated by [ISO/IEC00 Sec. 5.1.2.3] are preserved for the accesses to private objects on each thread.

B.3 Consistency Semantics of Standard Libraries and Language Operations

B.3.1 Consistency Semantics of Synchronization Operations

1 UPC provides several synchronization operations in the language and standard library that can be used to strengthen the consistency requirements of a program. Sections 7.2.4 and 6.6.1 define the consistency effects of these operations in terms of a "null strict reference". The formal definition presented here is operationally equivalent to that normative definition, but is more explicit and therefore included here for completeness.

2 The memory consistency semantics of the synchronization operations are defined in terms of equivalent accesses to a fresh variable $l_{synch} \in L$ that does not appear elsewhere in the program.[48]

[48] Note: These definitions do not give the synchronization operations their synchronizing effects – they only define the memory model behavior.

- A *upc_fence* statement implies a strict write followed by a strict read: $SW(l_{synch}, 0)$; $SR(l_{synch}, 0)$
- A *upc_notify* statement implies a strict write: $SW(l_{synch}, 0)$ immediately after evaluation of the optional argument (if any) and before the notification operation has been posted.
- A *upc_wait* statement implies a strict read: $SR(l_{synch}, 0)$ immediately after the completion of the statement.
- A *upc_lock*() call or a successful *upc_lock_attempt*() call implies a strict read: $SR(l_{synch}, 0)$ immediately before return.
- A *upc_unlock* call implies a strict write: $SW(l_{synch}, 0)$ immediately upon entry to the function.

3 The actual data values involved in these implied strict accesses is irrelevant. The strict operations implied by the synchronization operations are present only to serve as a consistency point, introducing orderings in $<_{Strict}$ that restrict the relative motion in each $<_t$ of any surrounding non-strict accesses to shared objects issued by the calling thread.

B.3.2 Consistency Semantics of Standard Library Calls

1 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. *upc_mem*-$\{put, get, cpy\}$) or collectively (e.g. *upc_all_broadcast*, etc). This section defines the consistency semantics of the accesses to shared objects which are implied to take place within the implementation of these library functions, to provide well-defined semantics in the presence of concurrent explicit reads and writes of the same shared objects. For example, an application which calls a function such as *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.

2 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 Non-Collective Standard Library Calls

1 For *non-collective* functions in the UPC standard library (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:

```
#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 ($x, y, z[0]$ 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 observe a final result of $local\_y == 2$, $local\_z1 == -4$, $local\_z0 == 0$ 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.[49] Finally, note that all of these observations remain true even if z had instead been declared as:

```
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.

3 If upc_fence operations were inserted in the blank lines immediately preceding and following the upc_memput invocation in the example above, then $<_{Strict}$ 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.[50] Explicit use of 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 cases where such ordering guarantees are required for program correctness.

[49] This is a consequence of the byte-oriented nature of shared data movement functions (which is assumed in the absence of further specification) and is orthogonal to the issue of write atomicity.

[50] 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 $local\_z1$.

B.3.2.2 Collective Standard Library Calls

1 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).

2 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`.

3 The upc_fence operations implied by the rules above are sufficient to ensure the results one would naturally expect in the presence of relaxed or local accesses to shared objects issued immediately preceding or following an `ALLSYNC` or `MYSYNC` collective library call that accesses the same shared objects. Without such fences, nothing would prevent prior or subsequent non-strict 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.[51]

[51] 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 rather subtle in some points, but as described in Section 5.1.2.3, most programmers need not worry about these details. There are some simple properties that are helpful in understanding the semantics.[52] The first property is:

- A UPC program which accesses shared objects using only strict operations[53] will be sequentially consistent.

2 This property is trivially true due to the global total order that $<_{Strict}$ imposes over strict operations (which is respected in every thread's $<_t$), but may not very useful in practice – because the exclusive use of strict operations for accessing shared objects may incur a noticeable performance penalty. Nevertheless, this property may still serve as a useful debugging mechanism, because even in the presence of data races a fully strict program is guaranteed to only produce behaviors allowed under sequential consistency [Lam79], which is generally considered the simplest parallel 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.

4 $PotentialRaces(M)$ is defined as a set of unordered pairs (m_1, m_2):

$$PotentialRaces(M) \stackrel{def}{=} \left\{ (m_1, m_2) \;\middle|\; \begin{array}{l} Location(m_1) = Location(m_2) \;\wedge \\ Thread(m_1) \;\neq\; Thread(m_2) \;\wedge \\ (\; m_1 \in W(M) \;\vee\; m_2 \in W(M) \;) \end{array} \right\}$$

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

$$\forall (m_1, m_2) \in PotentialRaces(M) : (\; m_1 <_{Strict} m_2 \;) \;\vee\; (\; m_2 <_{Strict} m_1 \;)$$

[52] Note the properties described in this section and in Section 5.1.2.3 apply only to programs which are "conforming" as defined by [ISO/IEC00 Sec. 4] – namely, those where no thread performs an operation which is labelled as having undefined behavior (e.g. dereferencing an uninitialized pointer).

[53] i.e. no relaxed shared accesses, and no accesses to shared objects via pointers-to-local

6 Note this implies that all threads t and all enabling orderings $<_t$ agree upon the ordering of each $(m_1, m_2) \in PotentialRaces(M)$ (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.

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

B.5 Examples

1 The subsequent examples demonstrate the semantics of the memory model by presenting hypothetical execution traces and explaining how the memory model either allows or disallows the behavior exhibited in each trace. The examples labelled "disallowed" denote a trace which is not UPC consistent and therefore represent a violation of the specified memory model. Such an execution trace shall never be generated by a conforming UPC implementation. The examples labelled "allowed" denote a trace which is UPC consistent and therefore represent a permissible execution that satisfies the constraints of the memory model. Such an execution trace *may* be generated by a conforming UPC implementation.[54]

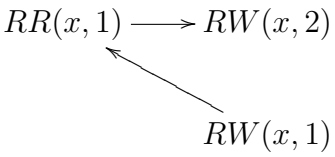
2 In the figures below, each execution is shown by the linear graph which is the $Precedes()$ program order for each thread, generated by an execution of the source program on the abstract machine. Pairs of memory operations that are ordered by the global ordering over memory operations in $AllStrict(M)$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

[54] The memory model specifies guarantees which must be true of any conformant UPC implementation and therefore may be portably relied upon by users. A given UPC implementation may happen to provide guarantees which are stronger than those required by the model, thus in general the set of behaviors which can be generated by conformant implementation will be a subset of those behaviors permitted by the model.

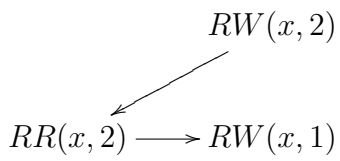
Arcs that are implied by transitivity are omitted. Assume all variables are initialized to 0.

3 EXAMPLE 1: **Allowed behavior** that would not be allowed under sequential consistency. There are only relaxed operations, so threads need not observe the program order of other threads. Because all operations are relaxed, there are no $\Rightarrow$ orderings between operations.

$T0$:	RR(x,1);	RW(x,2)
$T1$:	RR(x,2);	RW(x,1)

$<_0$: 

$T0$ observes $T1$'s write happening before its own read.

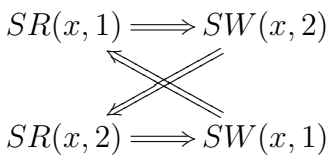
$<_1$: 

$T1$ must observe its own program order for conflicting operations, but it sees $T0$'s write as the first operation.

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

4 EXAMPLE 2: **Disallowed behavior** which is the same as the previous example, but with all accesses made strict. All edges in the graph below must therefore be $\Rightarrow$ edges. This also implies the program order edges must be observed in $<_{Strict}$ 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 $<_{Strict}$ cannot be defined that satisfies the write-to-read data flow requirements for a valid $<_t$.

$T0$:	SR(x,1);	SW(x,2)
$T1$:	SR(x,2);	SW(x,1)

$<_{Strict}$: 

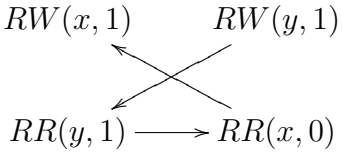
All of the edges shown are required, but this is not a valid $<_{Strict}$, since it contains a cycle.

5 EXAMPLE 3: **Allowed behavior** that would be disallowed (as in the first example) 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 spin lock, where y is the lock protecting the variable x. The implication is that UPC programmers should not build synchronization out of relaxed operations.

$T0$:	RW(x,1);	RW(y,1)
$T1$:	RR(y,1);	RR(x,0)

$<_0$: $RW(x,1) \longrightarrow RW(y,1)$

$T0$ observes only its own writes.
The writes are non-conflicting, so either ordering constitutes a valid $<_0$.

$<_1$: 

To satisfy write-to-read data flow in $<_1$, RW(x,1) must follow RR(x,0) and RR(y,1) must follow RW(y,1). There are three other valid $<_1$ orderings which satisfy these constraints.

6 EXAMPLE 4: **Allowed behavior** that would be disallowed 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 thread's operations happening out of order.

$T0$:	RW(x,1);	RR(y,0)
$T1$:	RW(y,1);	RR(x,0)

$<_0$: $RW(x,1) \longrightarrow RR(y,0)$

$RW(y,1)$

The only constraint on $<_0$ is RW(y,1) must follow RR(y,0). Several other valid $<_0$ orderings are possible.

$<_1$: $RW(x,1)$

$RW(y,1) \longrightarrow RR(x,0)$

Analogous situation with a write-after-read, this time on x. Several other valid $<_1$ orderings are possible.

7 EXAMPLE 5: **Disallowed behavior** because 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 one attempts to construct $<_{Strict}$.

$T0$: SW(x,2); SR(x,1)
$T1$: SW(x,1); SR(x,2)

$<_{Strict}$: $SW(x,2) \Longrightarrow SR(x,1)$

$\Updownarrow$

$SW(x,1) \Longrightarrow SR(x,2)$

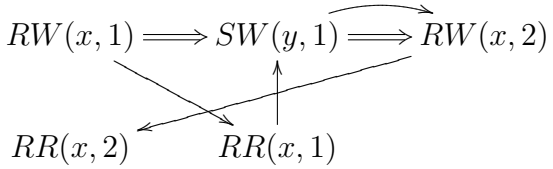
8 EXAMPLE 6: **Allowed behavior** where a thread observes its own reads occurring out-of-order. Reordering of reads is commonplace in serial compilers/hardware, but in this case an intervening modification by a different thread makes this reordering visible. Strengthening the model to prohibit such reordering of relaxed reads to the same location 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 to different locations or no other thread was writing the location.

$T0$: RW(x,1); SW(y,1); RW(x,2)

$T1$: RR(x,2); RR(x,1)

$<_{Strict}$: $RW(x,1) \Longrightarrow SW(y,1) \Longrightarrow RW(x,2)$ $DependOnThreads(M_0)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$: $RW(x,1) \Longrightarrow SW(y,1) \Longrightarrow RW(x,2)$ $<_0$ conforms to $<_{Strict}$

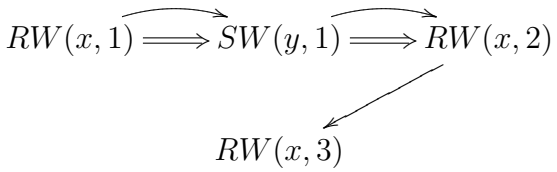
$<_1$: 

$<_1$ conforms to $<_{Strict}$. T1's operations on x do not conflict because they are both reads, and hence may appear relatively reordered in $<_1$. One other $<_1$ ordering is possible.

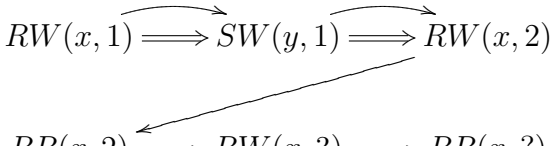
9 EXAMPLE 7: **Disallowed behavior** similar to the previous example, but in this case the addition of a relaxed write on thread 1 introduces dependencies in $DependOnThreads(M_1)$, such that (all else being equal) the model requires T1's second read to return the value 3. If T1's write were to any location other than x, the behavior shown would be allowed.

$T0$: RW(x,1); SW(y,1); RW(x,2)
$T1$: RR(x,2); RW(x,3); RR(x,1)

$<_{Strict}$: $RW(x,1) \Longrightarrow SW(y,1) \Longrightarrow RW(x,2)$ $DependOnThreads(M_0)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$: 

$<_0$ conforms to $<_{Strict}$. Other orderings are possible.

$<_1$: 

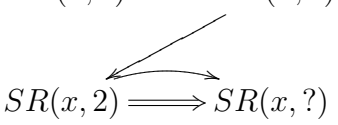
This is the only $<_1$ that conforms to $<_{Strict}$ and $DependOnThreads(M_1)$. The second read of x cannot return 1 - it must return 3.

10 EXAMPLE 8: **Disallowed behavior** demonstrating why strict reads appear in every $<_t$, rather than just for the thread that issued them. If the strict reads were absent from $<_0$, this behavior would be allowed.

$T0$:	RW(x,1);	RW(x,2)
$T1$:	SR(x,2);	SR(x,1)

$<_{Strict}$: $SR(x,2) \Longrightarrow SR(x,1)$

$DependOnThreads(M_1)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$: 

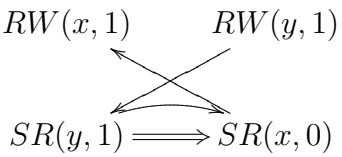
This is the only $<_0$ that conforms to $<_{Strict}$ and $DependOnThreads(M_0)$. The second read of x cannot return 1 - it must return 2.

11 EXAMPLE 9: **Allowed behavior** similar to the previous example, but the writes are no longer conflicting, and therefore not ordered by $DependOnThreads(M_0)$.

$T0$:	RW(x,1);	RW(y,1)
$T1$:	SR(y,1);	SR(x,0)

$<_{Strict}$: $SR(y,1) \Longrightarrow SR(x,0)$

$DependOnThreads(M_1)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0, <_1$: 

The writes are non-conflicting, therefore not ordered by $DependOnThreads(M_0)$.

12 EXAMPLE 10: **Allowed behavior** Another example of a thread observing its own relaxed reads out of order, regardless of location accessed.

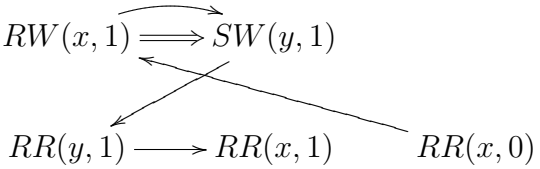
$T0$:	RW(x,1);	SW(y,1)	
$T1$:	RR(y,1);	RR(x,1);	RR(x,0)

$<_{Strict}$: $RW(x,1) \Longrightarrow SW(y,1)$

$DependOnThreads(M_0)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$: $RW(x,1) \Longrightarrow SW(y,1)$

Relaxed reads from thread 1 do not appear in $<_0$

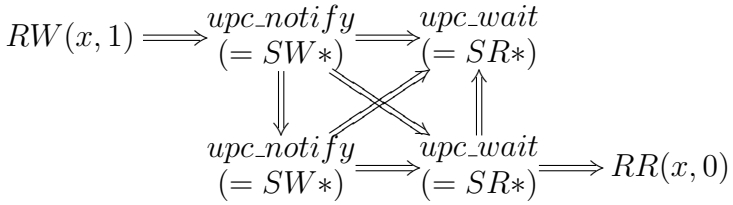
$<_1$: 

Relaxed reads have been reordered. Other $<_1$ orders are possible.

13 EXAMPLE 11: **Disallowed behavior** demonstrating that a barrier synchronization orders relaxed operations as one would expect.

$T0$:	RW(x,1);	upc_notify;	upc_wait	
$T1$:		upc_notify;	upc_wait;	RR(x,0)

$<_{Strict}$:



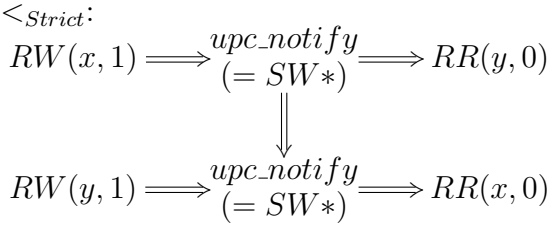
$DependOnThreads(M)$ and the synchronization semantics of barrier imply that $<_{Strict}$ must respect all the edges shown.[55]

There is no valid $<_1$ which respects $<_{Strict}$ – write-to-read data flow along the chain $RW(x,1) \Rightarrow upc\_notify \Rightarrow upc\_wait \Rightarrow RR(x,0)$ implies the read must return 1 (i.e. because $RW(x,1) <_{Strict} RR(x,0)$ and there are no intervening writes of x).

14 EXAMPLE 12: **Disallowed behavior** $<_{Strict}$ is an ordering over the pairs in $AllStrict(M)$, which includes an edge between two upc_notify operations. Every $<_t$ must conform to a single $<_{Strict}$ 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.

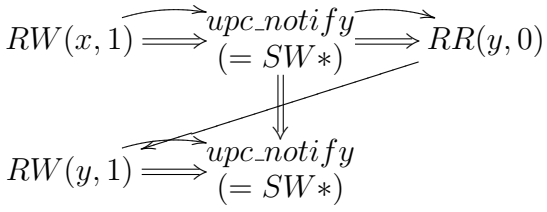
$T0$:	RW(x,1);	upc_notify;	RR(y,0);	(upc_wait not shown)
$T1$:	RW(y,1);	upc_notify;	RR(x,0);	(upc_wait not shown)

$<_{Strict}$:



$DependOnThreads(M_0)$ implies these edges in $StrictOnThreads(M)$ must be respected by $<_{Strict}$.[56]

$<_0$:



[55] except for the edge between the upc_wait operations and the edge between the upc_notify operations, both of which can point either way.

[56] except the edge between the upc_notify operations, which can point either way.

$<_1$: $RW(x,1) \Longrightarrow^{upc\_notify}_{(=SW*)} \Downarrow RW(y,1) \Longrightarrow^{upc\_notify}_{(=SW*)} \Longrightarrow RR(x,0)$ Read cannot return 0.

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

B.6 Formal Definition of Precedes

1 This section outlines a formal definition for the $Precedes(m_1, m_2)$ partial order, a predicate which inspects two memory operations in the execution trace that were issued by the same thread and returns true if and only if m_1 is required to precede m_2, according to the sequential abstract machine semantics of [ISO/IEC00 Sec. 5.1.2.3], applied to the given thread. Intuitively, this partial order serves to constrain legal serial program behavior based on the order of the statements a programmer wrote in the source program. For most purposes, it is sufficient to rely upon an intuitive understanding of sequential program order when interpreting the behavior of $Precedes()$ in the memory model - this section provides a more concrete definition which may be useful to compiler writers.

2 In general, the memory model affects the instructions which are issued (and therefore, the illusory "program order", if we were endeavoring to construct a total order on memory operations given only a static program). Luckily, providing a functional definition for $Precedes()$ does not require us to embark on the problematic exercise of defining a totally-ordered "program order" of legal executions based only on the static program. All that's required is a way to determine after-the-fact (i.e. given an execution trace) whether two memory operations that *did* execute on a single thread were generated by source-level operations that are required to have a given ordering by the sequential abstract machine semantics. Finally, note that $Precedes()$ is a partial order and not a total order - two accesses from a given thread which

are not separated by a sequence point in the abstract machine semantics will not be ordered by *Precedes*() (and by extension, their relative order will not be constrained by the memory model).

3 Given any memory access in the trace, it is assumed that we can decide uniquely which source-level operation generated the access. One mechanism for providing this mapping would be to attach an abstract "source line number" tag to every memory access indicating the source-level operation that generated it.[57]

In practice, this abstract numbering needs to be slightly different from actual source line number because the user may have broken a line in the middle of an expression where the abstract machine guarantees no ordering - but we can conceptually add or remove line breaks as necessary to make the line numbers match up with abstract machine sequence points without changing the meaning of the program (ie whitespace is not significant). Also, without lack of generality we can assume the program consists only of a single UPC source file, and therefore the numbering within this file covers every access the program could potentially execute.[58]

4 Now, notice that given the numbering and mapping above, we could immediately define an adequate *Precedes*() relation if our program consisted of only straight-line code (ie a single basic block in CFG terminology). Specifically, in the absence of branches there is no ambiguity about how to define *Precedes*() - a simple integer less-than (<) comparison of the line number tags is sufficient.

Additionally, notice that a program containing only straight-line code and forward branches can also easily be incorporated in this approach (ie the CFG for our program is a DAG). In this case, the basic blocks can be arranged such that abstract machine execution always proceeds through line numbers in monotonically non-decreasing order, so a simple integer less-than (<) comparison of the line number tags attached to the dynamic operations

[57] Compiler optimizations which coalesce accesses or remove them entirely are orthogonal to this discussion - specifically, the correctness of such optimizations are defined in terms of a behavioral equivalence to the unoptimized version. Therefore, as far as the memory model is concerned, every operation in the execution trace is guaranteed to map to a unique operation at the source level.

[58] Multi-file programs are easily accomodated by stating the source files are all concatenated together into a single master source file for the purposes of defining *Precedes*.

is still a sufficient definition for *Precedes*.

5 Obviously we want to also describe the behavior of programs with backward branches. We handle them by defining a sequence of abstract rewriting operations on the original program that generate a new, simplified representation of the program with equivalent abstract machine semantics but without any backward branches (so we reduce to the case above). Here are the rewriting steps on the original program:

Step 1. Translate all the high-level control-flow constructs in the program into straight-line code with simple conditional or unconditional branches. Lower all compound expressions into "simple expressions" with equivalent semantics, introducing private temporary variables as necessary. Each "simple expression" should involve at most one memory access to a location in the original program. Order the simple expressions such that the abstract machine semantics of the original program are preserved, placing line breaks as required to respect sequence point boundaries. In cases where the abstract machine semantics leave evaluation order unspecified, place the relevant simple expressions on the same line.

At this point rewritten program code consists solely of memory operations, arithmetic expressions, built-in operations (like *upc_notify*), and conditional or unconditional goto operations. For example this program:

```
1: i = 0;
2: while ( i < 10 ) {
3:   A[i] = i;
4:   i = i + 1;
5: }
6: A[10] = -1;
```

Conceptually becomes:

```
1: i = 0;
2: if ( i >= 10 ) goto 6;
3: tmp_1 = i; A[i] = tmp_1;
4: tmp_2 = i; i = tmp_2 + 1;
5: goto 2;
6: A[10] = -1;
```

The translation for the other control-flow statements is similarly straightforward and well-documented in the literature of assembly code generation techniques for C-like languages. All control flow (including function call/return, setjmp/longjmp, etc) can be represented as (un)conditional branches in this manner. Call this rewritten representation the *step-1 program*.

Step 2. Compute the maximum line number (MLN) of the step-1 program ($MLN = 6$ in the example). Clone the step-1 program an infinite number of times and concatenate the copies together, adjusting the line numbering for the 2nd and subsequent copies appropriately (note, this is an abstract transformation, so the infinite length of the result is not a practical issue). While cloning, rewrite all the goto operations as follows:

For a goto operation in copy C of the step-1 program (zero-based numbering), which is a copy of line number N in the step-1 program and targeting original line number T:

```
if (T > N) set goto target = C*MLN + T  // step-1 forward branch
else       set goto target = (C+1)*MLN + T // step-1 backward branch
```

In other words, step-1 forward branches branch to the same relative place in the current copy of the step-1 program, and backward branches become forward branches to the *next* copy of the step-1 program. So our example above conceptually becomes:

```
1: i = 0;
2: if ( i >= 10 ) goto 6;
3: tmp_1 = i; A[i] = tmp_1;
4: tmp_2 = i; i = tmp_2 + 1;
5: goto 8;                      // rewritten backward goto
6: A[10] = -1;

7:  i = 0;
8:  if ( i >= 10 ) goto 12;     // rewritten forward goto
9:  tmp_1 = i; A[i] = tmp_1;
10: tmp_2 = i; i = tmp_2 + 1;
11: goto 14;                    // rewritten backward goto
12: A[10] = -1;
```

```
13: i = 0;
...
```

After this transformation, all branches are forward branches. Now, the memory model describes behavior of the step-2 rewritten program, and $Precedes()$ is defined as a simple integer less-than ($<$) comparison of the step-2 program's line number tags attached to any two given memory accesses in the execution trace.

C UPC versus C Standard Section Numbering

UPC specifications subsection	C Standard specifications subsection
1	1
2	2
3	3
4	4
5	5
6	6
6.1	6.1
6.2	6.4.2.2
6.3	6.5
6.3.6	6.5.3.1
6.4	6.7
6.4.1	6.7.3
6.4.3	6.7.5
6.5	6.8
6.6	6.10
7	7
7.1	7.1.2

Table A1. Mapping UPC subsections to C Standard specifications subsections

References

[CAG93] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, Katherine A. Yelick. *Parallel programming in Split-C*, Proceedings of Supercomputing 1993, p. 262-273.

[CDC99] W. W. Carlson, J. M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. CCS-TR-99-157. IDA/CCS, Bowie, Maryland. May, 1999.

[ISO/IEC00] ISO/IEC. Programming Languages-C. ISO/IEC 9899. May, 2000.

[Lam79] L. Lamport. *How to make a Multicomputer that Correctly Executes Multiprocess Programs.* IEEE Transactions on Computers, C-28(9):690-69, September 1979.

[MPI2] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, July 18, 1997.

Index