UPC Language Specifications
Version 1.3

A publication of the UPC Consortium

November 16, 2013

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 Version 1.3 is also the result of many contributors. Gary Funck created a "Google Code Project" that was used to track issues, changes, and contributions. Details can be found at http://code.google.com/p/upc-specification.

5 Members of the UPC consortium may be contacted via the world wide web at http://upc-lang.org, which provides links to many UPC-related resources. This site hosts the official version of this language specification and related library specifications, as well as annotated documents showing the complete and detailed set of changes relative to prior specification revisions.

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.

5 Version 1.3 of UPC was developed throughout 2012 via internet collaboration, and finalized in November 2013.

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 documents and their identified normative references constitute provisions of these UPC specifications. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document applies.

2 ISO/IEC 9899: 1999(E), Programming languages - C [ISO/IEC00]

3 UPC Required Library Specifications [UPC-LIB-REQ]

4 UPC Optional Library Specifications [UPC-LIB-OPT]

5 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.

6 Implementations shall document the exact revisions of [UPC-LIB-REQ] and [UPC-LIB-OPT] to which they conform.

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 **ultimate element type**
for non-array types, the type itself. For an array type “array of T”, the

ultimate element type of T.

3.3

1 **shared type**
a type whose ultimate element type is shared-qualified.

3.4

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

3.4.1

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]

3.4.2

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]

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

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

3.4.3

1 **shared array**
an array with shared type.

3.5

1 **affinity**
logical association between shared objects and threads. Each byte in a shared object has affinity to exactly one thread. The affinity of a shared object is the same as that of the first byte in the object.[3]

3.6

1 **pointer-to-shared**
a pointer whose referenced type is a shared type.

3.7

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

3.8

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

[3]For non-array shared objects, all bytes in the object have the same affinity as the object itself. This is not necessarily true for shared array objects, which may span multiple threads.

3.8.1

1 **shared access**
an access using an expression whose type is a shared type.

3.8.1.1

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

3.8.1.2

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

3.8.1.3

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

3.8.1.4

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

3.8.2

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

3.9

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

such operations to be matched across all threads.[4] The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

3.10

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.

3.11

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 All terminology and requirements defined in [ISO/IEC00 Sec. 4] also apply to this document and UPC implementations.

[4]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 the presence or absence of 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[5].

5.1.2.2 Program termination

1 A program is terminated by the termination of all the threads[6] 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.

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

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

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

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_BLOCK_SIZE**
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 integral value; 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 integral value; it specifies the unique thread index.[7]The range of possible values is `0..THREADS-1`[8].

6.3.3 `UPC_MAX_BLOCK_SIZE`

1 `UPC_MAX_BLOCK_SIZE` is a predefined integer constant value. It indicates the maximum value[9] 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.

[7]The definition of MYTHREAD and THREADS as expressions, not objects or l-values, means one cannot assign to them or take their address.

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

[9] 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.

6.4.1 Unary Operators

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 a shared type or an expression with shared type. 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 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 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 a shared type or an expression with shared type. 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 ultimate element type of the operand. 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 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 a shared type or an expression with shared type. 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 ultimate element type of its operand. For a non-array operand, `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 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.

2 Relational operators (as defined in [ISO/IEC00 Sec 6.5.8]) shall not be applied to a pointer-to-shared with incomplete type.[10]

Semantics

3 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] T *p, *p1; /* B a positive integer,
                         T not a shared type */
int i;

p1 = p + i;
```

4 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:[11]

```
ptrdiff_t elem_delta = i * (sizeof(T) / upc_elemsizeof(*p))
upc_phaseof(p1) == (upc_phaseof(p) + elem_delta) mod B
```

[10]Eg. The (>,<,>=,<=) operators may not have an operand with `(shared void *)` type.

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

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

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

```
T *P1, *P2;     /* T is not a shared type */
shared [] T *S1, *S2;

P1 = (T*) S1;  /* allowed if upc_threadof(S1) == MYTHREAD */
P2 = (T*) S2;  /* allowed if upc_threadof(S2) == MYTHREAD */
```

6 For all S1 and S2 that point to two distinct objects with affinity to the same thread, where both are subobjects contained in the same shared array whose ultimate element type is a qualified version of `T`:

- S1 and P1 shall point to the same object.
- S2 and P2 shall point to the same object.
- The expression `P1 + (S2 - S1) == P2` shall evaluate to 1.[12]

7 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.

8 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.

9 When two pointers-to-shared are compared using a relational operator, as described in [ISO/IEC00 Sec 6.5.8], the expression `pts1` $\oplus$ `pts2` where $\oplus \in$ `{>,<,>=,<=}` is equivalent to: `(pts1 - pts2)` $\oplus$ `0`. If the result of the subtraction is undefined, so is the result of the relational operator.

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

[12]This implies there is no padding inserted between blocks of shared array elements with affinity to a thread.

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 a shared type. [13] An exception is made when a null pointer constant is cast, the result is called the *null pointer-to-shared*.[14]

Semantics

2 The casting or assignment from one pointer-to-shared to another where one of the types is a qualified or unqualified version of `shared void*`, the *generic pointer-to-shared*, preserves the phase component unchanged in the resulting pointer value (except as discussed in the next paragraph). The casting or assignment from one non-generic pointer-to-shared to another in which either the block size or the size of the ultimate element type of the referenced type differs, or either type is incomplete, results in a pointer with a zero phase.

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[15] to a pointer-to-local[16] and the affinity of any byte comprising 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 Bytes with affinity to a given thread containing shared objects can be accessed by either pointers-to-shared or pointers-to-local of that thread.

7 EXAMPLE 1:

```
    int i, *p;
```

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

[14][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.

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

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

```
    shared int *q;
    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 *`.

2 When the unary `&` is applied to an expression with a shared array type, the result is a pointer-to-shared that points to the beginning of the pointed-to shared array, whose referenced type matches that of the expression the unary `&` was applied to.

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-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.[17]

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

7 A layout qualifier of [*] shall not appear in the declaration specifiers of a declaration whose storage-class specifier is typedef.

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

Semantics

9 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.

10 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.

11 The layout qualifier dictates the blocking factor for the type being shared

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

qualified. This factor is the nonnegative number of consecutive objects with ultimate element type of the array (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.

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

13 The block size is a part of the type compatibility[18]

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

15 The *effective type* of a shared object is the type as determined by [ISO/IEC00 Sec. 6.5], with the top-level (rightmost) shared qualifier removed.[19]

16 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.

17 EXAMPLE 1: declaration of a shared scalar

```
strict shared int y;
```

`strict shared` is the type qualifier.

18 EXAMPLE 2: automatic storage duration

```
void foo (void) {
```

[18]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.

[19]For example, in the file-scope declaration `shared [10] int A[10*THREADS];` the effective type of the object `A[0]` is `int`. This implies the following lvalue expressions are all permitted for accessing the object:

```
int x1 = A[0];
int x2 = *(int *)&(A[0]); // valid only for MYTHREAD==0
int x3 = *(shared [] int *)&(A[0]);
```

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

19 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)

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.[20]

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 shared type.

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, the following restrictions apply: Every declaration of a shared array with definite blocksize shall include the THREADS keyword exactly once, in one dimension of the array (including through typedefs). Every array type that is a shared type with definite blocksize shall include the THREADS keyword at most once, in one dimension (including through typedefs). In both cases, the THREADS keyword shall only occur either alone or when multiplied by an integer constant expression (as defined in [ISO/IEC00 Sec. 6.6]) with positive value. [21]

[20]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.

[21]In the *static THREADS* environment THREADS is an integer constant expression,

3 The THREADS keyword shall not appear in any array type that is a shared array with indefinite blocksize under the *dynamic THREADS* environment.

4 If an init-declarator that declares a shared array includes an initializer, the behavior is implementation-defined.

Semantics

5 The objects with ultimate element type that comprise a shared array are distributed in a round robin fashion, by chunks of block-size objects, such that the i-th object has affinity with thread ($\lfloor i/block\_size \rfloor$ `mod THREADS`).

6 In an array declaration, the type qualifier applies to the ultimate element type of the array.

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

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

```
shared int x [10*THREADS];
shared int x [THREADS*(100*20)];
shared [] int x [10];
```

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

```
shared int x [10+THREADS];
shared [] int x [THREADS];
shared int x [10];
shared int x [THREADS][4*THREADS];
shared int x [THREADS*THREADS];
shared int x [THREADS*100*20];
shared int (**p)[THREADS][THREADS];
typedef shared int (*t)[THREADS][13][THREADS];
shared void *p = (shared int (*)[THREADS][THREADS])q;
```

10 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.

and is therefore valid in all dimensions.

11 EXAMPLE 4:

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

`shared [3]` applies to the ultimate element 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];
```

12 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.

13 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 a type such that its value may be assigned to an object of 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`.[22] 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.[23] `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.[24]

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

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

[24]One implementation of `upc_fence` may be achieved by a null strict access: `{ static`

6 A null strict access is implied before[25] a `upc_notify` statement and after a `upc_wait` statement.[26]

7 If one or more threads provide optional expressions to `upc_notify` in the current synchronization phase, then the subsequent `upc_wait` statement of at least one thread shall interrupt the execution of the program in an implementation-defined manner if either of the following two rules are violated: [27] 1) All optional expressions provided to `upc_notify` must have equal values (a consensus). 2) Any optional expression provided to `upc_wait` must equal the consensus value from the `upc_notify`. If no thread provides an optional expression to `upc_notify`, then no interruption shall be generated.

8 The `upc_barrier` statement is equivalent to the compound statement[28]:

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

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

9 The barrier operations at thread startup and termination have a value of *expression* which is not in the range of the type `int`.[29]

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; }
```

`shared strict int x; x = x;}`

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

[26]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.

[27]After such an interruption, subsequent behavior is undefined.

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

[29]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.

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:

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*.[30]

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

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

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`. If the value of *affinity* is negative, behavior is undefined.

9 When *affinity* is `continue` or not specified, each loop body of the `upc_forall` statement is performed by every thread and semantic 5 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`. Nesting of `upc_forall` statements is considered an obsolescent feature, and may be prohibited in a future revision of this specification.

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[31], the behavior 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.[32]

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

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

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, followed by a final call to foo2() on each thread before the loop terminates. 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 (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 Each translation unit has an implicit `#pragma upc relaxed` before the first line.

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[33]

`__UPC__` The integer constant 1, indicating a conforming implementation.

`__UPC_VERSION__` The integer constant 201311L.

`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:

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

`__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.

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.h>
<upc_types.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 Every inclusion of `<upc.h>` has the effect of including `<upc_types.h>`.

6 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.

2 Unless otherwise noted, all of the functions, types and macros specified in Section 7.2 are declared by the header `<upc.h>`.

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.

2 There is no required correspondence between the functions specified in Section 7.2.2 to allocate and free objects. Either of the `upc_free` or `upc_all_free` functions may be used to free shared space allocated using `upc_all_alloc`, `upc_global_alloc` or `upc_alloc`.

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_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` or `upc_all_alloc` functions, or if the space has been deallocated by a previous call to `upc_free` by any thread,[34] or a previous call to `upc_all_free`, the behavior is undefined.

7.2.2.5 The `upc_all_free` function

Synopsis

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

Description

2 `upc_all_free` is a *collective* variant of `upc_free`, provided as a convenience. It must be called collectively by all threads with the *single-valued* argument `ptr`.

3 The `upc_all_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`, or `upc_all_alloc` functions, or if the space has been deallocated by a previous call to `upc_free` or `upc_all_free`, the behavior is undefined.

4 The shared storage referenced by `ptr` is guaranteed to remain valid until all threads have entered the call to `upc_all_free`, but the function does not otherwise guarantee any synchronization or strict reference.

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

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`.[35]

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.[36]

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

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.

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

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

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.

3 Given the following declarations:

```
T *P1, *P2;    /* T is not a shared type */
shared T *S1, *S2;

P1 = (T*) S1;  /* allowed if upc_threadof(S1) == MYTHREAD */
P2 = (T*) S2;  /* allowed if upc_threadof(S2) == MYTHREAD */
```

For all S1 and S2 that point to two distinct elements of the same shared array object which have affinity to the same thread, the expression:
`((ptrdiff_t) upc_addrfield(S2) - (ptrdiff_t)upc_addrfield(S1))`
shall evaluate to the same value as: `((P2 - P1) * sizeof(T))`.

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

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 * upc_elemsizeof (d)`. 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 `upc_lock_t` 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.

4 There is no required correspondence between the functions specified in Section 7.2.4 to allocate and free locks. Either of the `upc_lock_free` or `upc_all_lock_free` functions may be used to free locks allocated using `upc_global_lock_alloc` or `upc_all_lock_alloc`.

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` by any thread, [37] or a previous call to `upc_all_lock_free`, 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 Subsequent or concurrent calls from any thread to functions defined in Section 7.2.4 using the lock referenced by `ptr` have undefined behavior. This also applies to any call to `upc_lock` on the the lock referenced by `ptr` which is blocked at the time of the call to `upc_lock_free`.

7.2.4.5 The `upc_all_lock_free` function

Synopsis

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

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

Description

2 `upc_all_lock_free` is a *collective* variant of `upc_lock_free`, provided as a convenience. It must be called collectively by all threads with the *single-valued* argument `ptr`.

3 The `upc_all_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` or `upc_all_lock_free`, the behavior is undefined.

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

5 The lock referenced by `ptr` is guaranteed to remain valid until all threads have entered the call to `upc_all_lock_free`, but the function does not otherwise guarantee any synchronization or strict reference.

6 Any subsequent calls from any thread to functions defined in Section 7.2.4 using the lock referenced by `ptr` have undefined behavior.

7.2.4.6 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.[38]

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

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()`.

7.2.4.7 The `upc_lock_attempt` function

Synopsis

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

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.

7.2.4.8 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).

4 If copying takes place between objects that overlap, the behavior is undefined.

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]
```

4 If copying takes place between objects that overlap, the behavior is undefined.

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]
```

4 If copying takes place between objects that overlap, the behavior is undefined.

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`.

7.3 UPC standard types `<upc_types.h>`

1 The `<upc_types.h>` header declares several standard types and value macros used by other UPC libraries.

2 Unless otherwise noted, all of the types and macros specified in Section 7.3 are declared by the header `<upc_types.h>`.

3 The `<upc_types.h>` header shall constitute a strictly-conforming translation unit for an [ISO/IEC00] C compiler (ie one that lacks UPC language extensions).

7.3.1 Operation designator (`upc_op_t`)

1 The `<upc_types.h>` header defines the type:

```
upc_op_t
```

which is an integer type whose values are used to designate a library operation or set of operations.

2 The `<upc_types.h>` header defines the following macros, which expand to integer constant expressions with type `upc_op_t`, which are suitable for use in `#if` preprocessing directives. Each macro value designates the specified operation. The expressions are defined such that bitwise or (`|`) of all combinations of the macros result in distinct positive values less than 65536.

Macro name	Specified operation
`UPC_ADD`	`Addition`
`UPC_MULT`	`Multiplication`
`UPC_AND`	`Bitwise and (&)`
`UPC_OR`	`Bitwise inclusive or (|)`
`UPC_XOR`	`Bitwise exclusive or (^)`
`UPC_LOGAND`	`Logical and (&&)`
`UPC_LOGOR`	`Logical or (||)`
`UPC_MIN`	`Minimum value (op1<op2?op1:op2)`
`UPC_MAX`	`Maximum value (op1>op2?op1:op2)`

3 Extension libraries may define additional value macros of type `upc_op_t`, but their values shall not conflict with those defined in `<upc_types.h>`.

7.3.2 Type designator (`upc_type_t`)

1 The `<upc_types.h>` header defines the type:

```
upc_type_t
```

which is an integer type whose values are used to designate a language type.

2 The `<upc_types.h>` header defines the following macros, which expand to integer constant expressions with type `upc_type_t`, distinct positive values less than 65536, and which are suitable for use in `#if` preprocessing directives. Each macro value designates the specified type.

Macro name	Specified type
`UPC_CHAR`	`signed char`
`UPC_UCHAR`	`unsigned char`
`UPC_SHORT`	`short`
`UPC_USHORT`	`unsigned short`
`UPC_INT`	`int`
`UPC_UINT`	`unsigned int`
`UPC_LONG`	`long`
`UPC_ULONG`	`unsigned long`
`UPC_LLONG`	`long long`
`UPC_ULLONG`	`unsigned long long`
`UPC_INT8`	`int8_t`
`UPC_UINT8`	`uint8_t`
`UPC_INT16`	`int16_t`
`UPC_UINT16`	`uint16_t`
`UPC_INT32`	`int32_t`
`UPC_UINT32`	`uint32_t`
`UPC_INT64`	`int64_t`
`UPC_UINT64`	`uint64_t`
`UPC_FLOAT`	`float`
`UPC_DOUBLE`	`double`
`UPC_LDOUBLE`	`long double`
`UPC_PTS`	`shared void *`

3 Extension libraries may define additional value macros of type `upc_type_t`, but their values shall not conflict with those defined in `<upc_types.h>`.

7.3.3 Synchronization flags (`upc_flag_t`)

1 The `<upc_types.h>` header defines the type:

```
upc_flag_t
```

which is an integer type.

2 The following macros are defined in `<upc_types.h>`:

```
UPC_OUT_ALLSYNC
UPC_OUT_MYSYNC
UPC_OUT_NOSYNC
UPC_IN_ALLSYNC
UPC_IN_MYSYNC
UPC_IN_NOSYNC
```

All expand to integer constant expressions with type `upc_flag_t` which are suitable for use in `#if` preprocessing directives. The expressions are defined such that bitwise or (`|`) of all combinations of the macros result in distinct positive values less than 64.

3 The semantics of these macros are defined in Section 7.3.4.

7.3.4 Memory Semantics of Library Functions

1 `upc_flag_t` is an integral type defined in `<upc_types.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[39]

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[40], and

`ALL` the function call may return only after all reads and writes of data are complete.[41]

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`.

[39]`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.

[40]`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.

[41]`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.

A Additions and Extensions

1 The UPC additions and extensions specification is divided into required [UPC-LIB-REQ] and optional [UPC-LIB-OPT] library specifications. Required extensions shall be provided by a conformant UPC implementation, while a conformant UPC implementation is not required to provide optional extensions. The optional extensions specifications 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 required extensions specification.

2 This section also describes the process used to add new items to the additions and extensions specification, which starts with inclusion in the optional extensions specification. Requirements for inclusion are:[42]

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 required extensions specification are:

1. Six months residence in the optional extensions specification.
2. The existence of either one (or more) publicly available "reference" implementation written in standard UPC OR at least two independent

[42]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.

implementations (possibly specific to a given UPC implementation).

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.

5 For each library extension, a separate header file whose name begins with `upc_` shall be specified. This header file shall be provided by an implementation if the extension is supported.

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 3.7:

- 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 $W(M)$:

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

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

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

B.2 Memory Access Model

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

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

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

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

2 Thus, $\mathit{StrictPairs}(M)$ is the set of all pairs of strict memory accesses, including those between threads, and $\mathit{StrictOnThreads}(M)$ is the set of all pairs of memory accesses from the same thread in which at least one is strict. $\mathit{AllStrict}(M)$ 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 $\mathit{DependOnThreads}(M_t)$:

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

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

4 $\mathit{DependOnThreads}(M_t)$ establishes an ordering between operations issued by a given thread t that involve a data dependence (i.e. those operations in $\mathit{Conflicting}(M_t)$) – this ordering is the one maintained by serial compilers and hardware. $\mathit{DependOnThreads}(M_t)$ additionally establishes an ordering between operations appearing in $\mathit{StrictOnThreads}(M_t)$. 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.[43] 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].[44]

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

[44]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.[45]

[45] 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.[46] 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.[47] 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.

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

[46]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.

[47]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 $local\_z1$.

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.[48]

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.[49] The first property is:

[48]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.

[49]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).

- A UPC program which accesses shared objects using only strict operations[50] 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 $\mathit{PotentialRaces}(M)$ is defined as a set of unordered pairs (m_1, m_2):

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

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

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

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

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

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.[51]

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 $m_1 \rightarrow m_2$.
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)

[51]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.

$<_0$:

$$\begin{tikzcd}
RR(x,1) \arrow[r] & RW(x,2) \\
 & RW(x,1) \arrow[ul]
\end{tikzcd}$$

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

$<_1$:

$$\begin{tikzcd}
 & RW(x,2) \arrow[dl] \\
RR(x,2) \arrow[r] & RW(x,1)
\end{tikzcd}$$

$T1$ 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)

$<_{Strict}$:

$$\begin{tikzcd}
SR(x,1) \arrow[r, Rightarrow] & SW(x,2) \arrow[dl, Rightarrow] \\
SR(x,2) \arrow[r, Rightarrow] & SW(x,1) \arrow[ul, Rightarrow]
\end{tikzcd}$$

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)

$<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r] & RW(y,1) \end{tikzcd}$$

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

$<_1$:

$$\begin{tikzcd} RW(x,1) & RW(y,1) \arrow[dl] \\ RR(y,1) \arrow[r] & RR(x,0) \arrow[ul] \end{tikzcd}$$

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)

$<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r] & RR(y,0) \arrow[dl] \\ RW(y,1) & \end{tikzcd}$$

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

$<_1$:

$$\begin{tikzcd} RW(x,1) & \\ RW(y,1) \arrow[r] & RR(x,0) \arrow[ul] \end{tikzcd}$$

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)

$<_{Strict}$:

$$\begin{tikzcd}
SW(x,2) \arrow[r, Rightarrow] \arrow[d, Leftrightarrow] & SR(x,1) \\
SW(x,1) \arrow[r, Rightarrow] & SR(x,2)
\end{tikzcd}$$

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)	

$<_{Strict}$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \arrow[r, Rightarrow] & RW(x,2)
\end{tikzcd}$$

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

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2)
\end{tikzcd}$$

$<_0$ conforms to $<_{Strict}$

$<_1$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[dr] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dll] \\
RR(x,2) & RR(x,1) \arrow[u] &
\end{tikzcd}$$

$<_1$ 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)

$<_{Strict}$:
$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \arrow[r, Rightarrow] & RW(x,2) \end{tikzcd}$$
$DependOnThreads(M_0)$ implies this is the only valid $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$:
$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dl] \\ & RW(x,3) & \end{tikzcd}$$
$<_0$ conforms to $<_{Strict}$. Other orderings are possible.

$<_1$:
$$\begin{tikzcd} & RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dll] \\ & RR(x,2) \arrow[r] & RW(x,3) \arrow[r] & RR(x,?) \end{tikzcd}$$
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)

$<_{Strict}$:

$$\begin{tikzcd} SR(x,2) \arrow[r, Rightarrow] & SR(x,1) \end{tikzcd}$$

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

$<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r] & RW(x,2) \arrow[dl] \\ SR(x,2) \arrow[r, Rightarrow] \arrow[r, bend left] & SR(x,?) \end{tikzcd}$$

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)

$<_{Strict}$:

$$\begin{tikzcd} SR(y,1) \arrow[r, Rightarrow] & SR(x,0) \end{tikzcd}$$

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

$<_0, <_1$:

$$\begin{tikzcd} RW(x,1) & RW(y,1) \arrow[dl] \\ SR(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SR(x,0) \arrow[ul] \end{tikzcd}$$

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)

$<_{Strict}$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \end{tikzcd}$$

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

$<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \end{tikzcd}$$

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

$<_1$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[dl] & \\ RR(y,1) \arrow[r] & RR(x,1) & RR(x,0) \arrow[llu] \end{tikzcd}$$

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)

$<_{Strict}$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & \begin{matrix} upc\_notify \\ (=SW*) \end{matrix} \arrow[r, Rightarrow] \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \begin{matrix} upc\_wait \\ (=SR*) \end{matrix} & \\ & \begin{matrix} upc\_notify \\ (=SW*) \end{matrix} \arrow[r, Rightarrow] \arrow[ur, Rightarrow] & \begin{matrix} upc\_wait \\ (=SR*) \end{matrix} \arrow[r, Rightarrow] \arrow[u, Rightarrow] & RR(x,0) \end{tikzcd}$$

$DependOnThreads(M)$ and the synchronization semantics of barrier imply that $<_{Strict}$ must respect all the edges shown.[52]

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

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

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)

$<_{Strict}$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & \substack{upc\_notify \\ (= SW*)} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & RR(y,0) \\
RW(y,1) \arrow[r, Rightarrow] & \substack{upc\_notify \\ (= SW*)} \arrow[r, Rightarrow] & RR(x,0)
\end{tikzcd}$$

$DependOnThreads(M_0)$ implies these edges in $StrictOnThreads(M)$ must be respected by $<_{Strict}$.[53]

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & \substack{upc\_notify \\ (= SW*)} \arrow[r, Rightarrow] \arrow[r, bend left] \arrow[d, Rightarrow] & RR(y,0) \arrow[dll] \\
RW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & \substack{upc\_notify \\ (= SW*)} &
\end{tikzcd}$$

$<_1$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & \substack{upc\_notify \\ (= SW*)} \arrow[d, Rightarrow] & \\
RW(y,1) \arrow[r, Rightarrow] & \substack{upc\_notify \\ (= SW*)} \arrow[r, Rightarrow] & RR(x,0)
\end{tikzcd}$$

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 $<_0$.

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

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.[54]

[54]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.

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.[55]

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 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

[55]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 $Precedes$.

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 spec section	ISO/IEC 9899 section	Description
1	1	Scope
2	2	Normative references
3	3	Terms, definitions and symbols
4	4	Conformance
5	5	Environment
6	6	Language
6.1	6.1	Notations
6.2	6.4.1	Keywords
6.3	6.4.2.2	Predefined identifiers
6.4.1	6.5.3	Unary operators
6.4.2	6.5.6	Pointer-to-shared arithmetic
6.4.3	6.5.4/6.5.16	Cast and assignment expressions
6.4.4	6.5.3.2	Address operators
6.5	6.7	Declarations
6.5.1	6.7.3	Type qualifiers
6.5.2	6.7.5	Declarators
6.5.2.1	6.7.5.2	Array declarators
6.6	6.8	Statements and blocks
6.6.2	6.8.5	Iteration statements
6.7	6.10	Preprocessing directives
6.7.1	6.10.6	Pragma directive
6.7.2	6.10.8	Predefined macro names
7	7	Library
7.1	7.1.2	Standard headers

Table A1. Mapping UPC spec sections to ISO/IEC 9899 sections

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