UPC Language Specifications

V1.0

Tarek A. El-Ghazawi
School of Computational Sciences
George Mason University
4400 University Drive
Fairfax, VA 22030-4444
tarek@gmu.edu

William W. Carlson
Jesse M. Draper
IDA Center for Computing Sciences
17100 Science Drive
Bowie, MD 20715
{wwc, jdraper}@super.org

February 25, 2001

Acknowledgments

Many scientists have contributed to the ideas and concepts behind these specifications. They are too many to mention here, but we would like to cite the contributions of David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren who have contributed to the initial UPC language concepts and specifications. We also would like to acknowledge the role of the participants in the first UPC workshop, held in May 2000 in Bowie, Maryland, and in which the specifications of this version were discussed. In particular we would like to acknowledge the support and participation of Compaq, Cray, HP, Sun, and CSC. We would like also to acknowledge the abundant input of Kevin Harris and Sébastien Chauvin and the efforts of Lauren Smith. Finally, the efforts of Brian Wibecan and Greg Fischer were invaluable in bringing these specifications to the final (version 1.0) state.

Table of contents

Introduction

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

2 This version of UPC, denoted version 1.0, has been 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.

1 Scope

1 This document focuses only on the UPC specifications that extend ANSI C to an explicit parallel C based on the distributed shared memory model. All ANSI C specifications as per ISO/SEC 9899 [ISO/SEC00] are considered a part of these UPC specifications, and therefore will not be addressed in this document.

2 Small parts of ANSI C [ISO/SEC00] may be repeated for self-containment and clarity of a subsequent UPC extension definition.

2 Normative references

1 The following document and its identified normative references in addition to these documents constitute provisions of these UPC specifications. This will not apply to subsequent revisions of this document.

2 ISO/SEC 9899: 1999(E), Programming languages - C [ISO/SEC00]

3 The section numbering of this document is consistent with the previous document [ISO/SEC00]. The correspondence between the subsection of this document and the previous document, however, is given in Appendix A.

4 In the beginning of each UPC specifications subsection, the corresponding ANSI-C [ISO/SEC00] subsection will be noted.

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

3.1.1

1 **local access**
<execution-time action> to read or modify, by a given thread, the value of an object in either the private space of that thread, or in the shared address locations that have affinity to that thread.

3.1.2

1 **private access**
<execution-time action> to read or modify the value of an object in the private address space of a given thread by that thread.

3.1.3

1 **remote access**
<execution-time action> to read or modify, by a given thread, the value of an object whose address is in the shared address space portion which has affinity to the other threads.

3.2

1 **affinity**
a logical association of a portion of the shared address space with a given thread.

3.3

1 **shared object**
A shared object is an object that resides in the shared address space.

3.4

1 **shared pointer**
A shared pointer is a pointer to a shared object.

3.5

1 **thread**
a program task in execution with access not only to a private memory space, but also to a shared memory space which can be accessed by other threads.

4 Conformance

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

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

5 Environment

5.1 Conceptual Models

5.1.1 Translation environment

5.1.1.1 Threads environment

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/SEC00: Sec. 5.1.2]

2 Each thread has local data on which it operates and which are logically divided into a private portion and a shared one. All operations on the private portion of the data are exactly as described in [ISO/SEC00].

3 Each thread may access shared data that have affinity to any thread; the semantics of these accesses are described herein.

4 Except for implied barriers at program startup and termination, there is no implicit synchronization among the threads.

5 Some library calls may imply synchronization among threads. These will be explicitly noted.

5.1.2.1 Program startup

1 In the execution environment of a UPC program, derived from the hosted environment as defined in ANSI C [ISO/SEC00], each thread calls the UPC program’s main () function.

5.1.2.2 Program termination

1 A program is terminated by the termination of all the threads (there is an implied barrier at program end) or a call to the function `upc_global_exit ()`.

2 Thread termination follows the ANSI C definition of program termination in [ISO/SEC00: 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).

5.1.2.3 Program execution

1 Unless declared objects or references are qualified as strict, there is no change to the ANSI C execution model as applied to an individual thread. This implies that translators are free to reorder and/or ignore operations (including shared operations) as long as the restrictions found in [ISO/SEC00: Sec. 5.1.2.3] are observed.

2 A further restriction applies to strict references. For each strict reference, the restrictions found in [ISO/SEC00: Sec. 5.1.2.3] must be observed with respect to all threads if that reference is eliminated (or reordered with respect to all other shared references in its thread).

3 Equally, the behavior of strict shared references can be defined as follows. Label each shared access S(i,j) or R(i,j), where S represents a strict shared access (read or write), R represents a relaxed shared access (read or write), i is the thread number making the access, j is an integer which monotonically increases as the evaluation of the program (in the abstract machine) proceeds from startup through termination. The "abstract order" is a partial ordering of all accesses by all threads such that an access x(a,b) occurs before y(c,d) in the ordering if a==c and b < d. The "actual order(k)" for thread k is another partial order in which x(a,b) occurs before y(c,d) if thread k observes the x access before it observes the y access. A thread observes all accesses present in the abstract order which effect either the data written to files by it or its input and output dynamics as described in [ISO/SEC00: Sect 5.1.2.3]. The least requirements on a conforming implementation are that:

- x(a,b) must "occur before" y(c,d) in actual order(e) if a == c and a == e and b < d

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

UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in [ISO/SEC00: Sec. 5.1.2.3].

6 Language

6.1 Notations

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

{ $expression_{opt}$ **}**

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 Predefined identifiers

1 This subsection provides the UPC parallel extensions of section 6.4.2.2 in [ISO/SEC00].

6.2.1 THREADS

1 **THREADS** is an integer equal to the number of independent computational units, i.e. threads. Under the "dynamic THREADS" translation environment, **THREADS** is a non-modifiable lvalue of type int. Under the "static THREADS" translation environment, **THREADS** is a constant.

6.2.2 MYTHREAD

1 **MYTHREAD** is defined at execution time; it specifies the unique thread index. The range of possible values is **0..THREADS-1**.

6.2.3 UPC_MAX_BLOCK_SIZE

1 **UPC_MAX_BLOCK_SIZE** is a predefined integer constant value. It indicates the maximum value allowed in a layout qualifier for shared data. It may be defined in upc.h, or it may be defined by the compiler.

6.3 Expressions

1 This subsection provides the UPC parallel extensions of section 6.5 in [ISO/SEC00].

6.3.1 The upc_localsizeof operator

upc_localsizeof *unary-expression*

upc_localsizeof (*type-name*)

Constraints

1 The **upc_localsizeof** operator shall apply only to shared objects or shared-qualified types.

Semantics

1 The **upc_localsizeof** operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a shared-qualified type. If the block size is indefinite and the operand is an expression, it returns zero on all threads which do not have affinity to the object. Otherwise it returns the same value on all threads; the value is the maximum of the size of objects with affinity to any one thread.

2 The type of the result is size_t.

6.3.2 The upc_blocksizeof operator

upc_blocksizeof *unary-expression*

upc_blocksizeof (*type-name*)

Constraints

1 The **upc_blocksizeof** operator shall apply only to shared objects or shared-qualified types.

Semantics

1 The **upc_blocksizeof** operator returns the block size of the operand, which may be a shared object or a shared-qualified type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of **upc_blocksizeof** is a compile-time constant.

2 If the operator of **upc_blocksizeof** has indefinite block size, the value of **upc_blocksizeof** is 0.

3 The type of the result is **size_t.**

Forward references: **indefinite block size** (6.4.2).

6.3.3 The `upc_elemsizeof` operator

`upc_elemsizeof` *unary-expression*

`upc_elemsizeof` (*type-name*)

Constraints

1 The **`upc_elemsizeof`** operator shall apply only to shared objects or shared-qualified types.

Semantics

1 The **`upc_elemsizeof`** operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For non-array objects, **`upc_elemsizeof`** returns the same value as sizeof.

2 The type of the result is **size_t.**

6.3.4 Shared pointer arithmetic

1 When an expression that has integer type is added to or subtracted from a shared pointer, the result has the type of the shared pointer operand. If the shared pointer 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 shared pointer arithmetic is identical to that described for normal C pointers in [ISO/SEC00 sec. 6.5.6], except that the thread of the new pointer shall be the same as that of the original pointer. If the shared array has a definite block size, then the following example describes pointer arithmetic:

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

```
p1 = p + i;
```

2 After this assignment the following equations must hold in any UPC implementation. In each case the / operator indicates truncating integer division and the % operator returns a nonnegative value less than its right hand side:

```
upc_phaseof (p1) == (upc_phaseof(p) + i) % B
upc_threadof (p1) == (upc_threadof(p) + (upc_phaseof(p) +
i)/B) % THREADS
```

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

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

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

4 For all S1 and S2 that point to two distinct elements of the same shared array object which have affinity to the same thread:

S1 and P1 shall point to the same object.
S2 and P2 shall point to the same object.

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

If S1 < S2 then **`upc_addrfield(S1)`** shall be **`< upc_addrfield(S2)`** otherwise **`upc_addrfield(S1)`** shall be > **`upc_addrfield(S2)`**

Forward references: **`upc_threadof`** (7.3.5), **`upc_phaseof`** (7.3.6), **`upc_addrfield`** (7.3.7).

6.3.5 Cast and Assignment Expressions

Constraints

1 A shared type qualifier shall not appear in a type cast of an object that is not shared-qualified; i.e., private pointers cannot be cast to shared.

Semantics

1 A cast or assignment from one shared pointer to another in which either the type size or the block size differs results in a pointer with a zero phase, unless one of the types is "shared void *", the generic shared pointer.

2 If a pointer with a shared-qualified type is cast to a pointer whose type is not shared-qualified, and the affinity of the shared data is not to the current thread, the result is undefined.

6.4 Declarations

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

Constraints

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

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

Syntax

1 The following is the declaration definition as per [ISO/SEC00] section 6.7, repeated here for self-containment and clarity of the subsequent UPC extension specifications.

2 *declaration:*

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

3 *declaration-specifiers:*

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

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

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

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

4 *init-declarator-list:*

init-declarator

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

5 *init-declarator:*

declarator

declarator **=** *initializer*

Forward references: strict and relaxed type qualifiers (6.4.2).

6.4.1 Type qualifiers

1 This subsection provides the UPC parallel extensions of section 6.7.3 in [ISO/SEC00].

Syntax

1 *type-qualifier*:

const

restrict

volatile

shared-type-qualifier

reference-type-qualifier

6.4.2 The shared and reference type qualifiers

Syntax

1 *shared-type-qualifier*:

shared *layout-qualifier*$_{opt}$

2 *reference-type-qualifier*:

relaxed

strict

3 *layout-qualifier*:

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

[*]

Constraints

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

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

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

Semantics

1 An object that has shared-qualified type shall exist in shared memory space and not in private space. Any thread may reference a shared object. Shared objects are placed in memory based on an affinity to a particular thread.

2 An object that has strict-qualified type behaves as described in section 5.1.2.3 of this document.

3 An object that has relaxed-qualified type behaves as if it were not strict-qualified.

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

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

6 The block size is a part of the type compatibility.

7 A **`shared void`*** pointer is assignment compatible with any shared pointer type.

8 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.

9 EXAMPLE 1: declaration of a shared scalar

```
strict shared int y;
```

strict shared is the type qualifier

10 EXAMPLE 2: automatic storage duration

```
void foo (void) {
shared int x;  /* a shared automatic variable is not allowed  */
shared int* y; /* a pointer to shared is allowed  */
int * shared z;       /*a shared automatic variable is not allowed*/
… }
```

11 EXAMPLE 3: inside a structure

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

Forward references: shared array (6.4.3.2), pointer declarator (6.4.3.1).

6.4.3 Declarators

Syntax

1 The following is the declarator definition as per [ISO/SEC00] section 6.7.5, repeated here for self-containment and clarity of the subsequent UPC extension specifications.

2 *declarator:*

pointer$_{opt}$ *direct-declarator*

3 *direct-declarator:*

identifier

(*declarator* **)**

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**

direct-declarator **[** **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list*$_{opt}$ ***]**

direct-declarator **(** *parameter-type-list* **)**

direct-declarator **(** *identifier-list*$_{opt}$ **)**

4 *pointer:*

***** *type-qualifier-list*$_{opt}$

***** *type-qualifier-list*$_{opt}$ *pointer*

5 *type-qualifier-list:*

type-qualifier

type-qualifier-list type-qualifier

Constraints

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

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

3 **`shared`** shall not appear in a declarator which has automatic storage duration, unless it qualifies a pointer type.

Semantics

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

2 Inside a structure, no data can be declared as shared; only pointers that point to shared objects can have the shared qualifier.

6.4.3.1 Pointer declarators

1 This subsection provides the UPC parallel extensions of section 6.7.5.1 in [ISO/SEC00].

Constraints

1 The cast of a shared pointer to a private pointer by a thread not having affinity with the referenced object has an undefined result.

Semantics

1 A shared reference which is cast to non-shared will lose all qualities pertaining to being shared.

2 Shared objects with affinity to a given thread can be accessed by either shared pointers or private pointers of that thread.

3 EXAMPLE 1:

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

```
if (upc_threadof(q) == MYTHREAD) p = (int *) q;
                                 /* is allowed */
```

6.4.3.2 Array declarators

1 This subsection provides the UPC parallel extensions of section 6.7.5.2 in [ISO/SEC00].

Constraints

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

Semantics

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

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

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

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

4 EXAMPLE 2: declarations legal only in static translation environment:

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

5 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.

6 EXAMPLE 4:

```
typedef int S[10];

shared [3] S T[3*THREADS];
```

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

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

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.

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

All elements referenced by subscripting or otherwise dereferencing p have affinity to the same thread. That thread may be any thread; it does not have to be thread 0.

6.5 Statements and blocks

1 This subsection provides the UPC parallel extensions of section 6.8 in [ISO/SEC00].

Syntax

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

6.5.1 Barrier Statements

Syntax

1 *synchronization-statement*:

upc_notify *expression*$_{opt}$**;**

upc_wait *expression*$_{opt}$**;**

upc_barrier *expression*$_{opt}$**;**

upc_fence;

Constraints

1 *expression* shall be an *integer expression*.

2 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. A synchronization phase consists of the execution of all statements between one **upc_notify** and the next.

Semantics

1 A **upc_wait** statement does not complete until all threads have completed the **upc_notify** statement which begins the synchronization phase. Note that this implies that all threads are in the same synchronization phase as they complete the **upc_wait** statement.

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

3 One implementation of **upc_fence** (or "remote memory barrier") may be achieved by a *null* strict reference: **{static shared strict int x; x = x; }.** The construct acts as a fence for the shared references occurring before or after it.

4 A null strict reference is implied before a **upc_notify** statement and after a **upc_wait** statement.

5 The **upc_wait** statement will generate a runtime error if the value of its expression (if given) does not equal the value of the expression (if given) by the **upc_notify** statement which starts the synchronization phase.

6 The **upc_wait** statement will generate a runtime error if the value of its expression (if given) differs from any expression (if given) on the **upc_wait** and **upc_notify** statements issued by any thread in the current synchronization phase.

7 The **upc_barrier** statement is equivalent to the compound statement:

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

8 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; }
```

9 Between the **upc_notify** and the **upc_wait** statements, references to shared data shall be permitted, but they have no synchronization relationship to the **upc_notify** and **upc_wait** statements.

6.5.2 Iteration statements

1 This subsection provides the UPC parallel extensions of section 6.8.5 in [ISO/SEC00].

Syntax

1 *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*

affinity:

expression$_{opt}$
`continue`

Constraints:

1 The *expression* for affinity shall be a pointer to a shared object or an integer expression.

Semantics:

1 The affinity field specifies to each thread which iterations of the loop body of the **`upc_forall`** statement it executes.

2 When *affinity* is a reference to shared memory space, 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*`)`**.**

3 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 pmod(affinity, THREADS), where pmod(a,b) is evaluated as (a>=0)?(a%b):(((a%b)+b)%b)

4 When "affinity" is **`continue`**, the loop body of the **`upc_forall`** statement is executed for every iteration on every thread.

5 When no affinity is specified, the execution behavior of the **`upc_forall`** statement is the same as it would be if the affinity were **`continue`**.

6 If the loop body of a **`upc_forall`** statement contains one or more **`upc_forall`** statements, either directly or through one or more function calls, the construct is called a "nested upc_forall" statement. In a "nested upc_forall", the outermost **`upc_forall`** statement that has an affinity expression which is not **`continue`** is called the "controlling upc_forall" statement. All **`upc_forall`** statements which are not

"controlling" in a "nested upc_forall" behave as if their affinity expressions were **`continue`**.

7 Unless all threads enter the beginning of the **`upc_forall`** statement during the same synchronization phase, the behavior is undefined.

8 If any iteration of a **`upc_forall`** statement (loop body or control expressions) produces a side-effect needed by another iteration of the same **`upc_forall`** statement, the behavior is undefined.

9 If a thread terminates or if it executes a **`upc_barrier`**, **`upc_notify`**, **`upc_wait`** or return statement within the dynamic scope of a **`upc_forall`** statement, the result is undefined. If a thread branches outside a **`upc_forall`** statement, the result is undefined.

10 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 were **`upc_threadof(&a[j])`** equals the value of **`MYTHREAD`**.

6.6 Preprocessing directives

1 This subsection provides the UPC parallel extensions of section 6.10 in [ISO/SEC00].

6.6.1 UPC pragmas

Semantics

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

```
#pragma upc strict
#pragma upc relaxed
```

These pragmas control the default behavior of code which follows. Under a strict default, all accesses to shared objects that are not qualified as relaxed are in strict mode. Under a relaxed default, all accesses to shared objects that are not qualified as strict are in relaxed mode.

2 These directives do not affect shared objects that are explicitly qualified as either strict or relaxed.

3 Unless these directives are used, shared references and objects which are neither strict qualified nor relaxed qualified behave in an implementation defined manner which is either strict default or relaxed default. Users wishing portable programs are strongly encouraged to specify default behavior either by using these directives or by including upc_strict.h or upc_relaxed.h.

4 The pragmas shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When they are outside external declarations, they apply until another such pragma or the end of the translation unit. When inside a compound statement, they apply until the end of the compound statement; at the end of the compound statement the state of the pragmas is restored to that preceding the compound statement. If these pragmas are used in any other context, their behavior is undefined.

7 Library

7.1 Standard headers

1 This subsection provides the UPC parallel extensions of section 7.1.2 in [ISO/SEC00].

2 The standard headers are

<upc_strict.h> **<upc_relaxed.h>** **<upc.h>**

3 upc_strict.h shall contain at least:

```
#pragma upc strict

#include <upc.h>
```

4 upc_relaxed.h shall contain at least:

```
#pragma upc relaxed

#include <upc.h>
```

5 upc.h shall contain at least:

```
#define barrier upc_barrier
#define barrier_notify upc_notify
#define barrier_wait upc_wait
#define forall upc_forall
#define fence upc_fence
```

7.2 General utilities

1 This subsection provides the UPC parallel extensions of section 7.20 in [ISO/SEC00].

7.2.1 Termination of all threads

Synopsis

```
upc_global_exit(int status)
```

Description

1 **`upc_global_exit`** () will flush all I/O, release all memory, and terminate the execution for all active threads.

7.3 Memory allocation functions

7.3.1 The **`upc_global_alloc`** function

Synopsis

1
```
#include <upc.h>
shared void *upc_global_alloc(size_t nblocks, size_t
nbytes);
nblocks : number of blocks
nbytes : block size
```

Description

1 Allocates a contiguous shared memory space blocked as if the following declaration were used:

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

2 Intended to be called by one thread; no synchronization with other threads is implied. If called by multiple threads, all threads which make the call get different allocations.

7.3.2 The **upc_all_alloc** function

Synopsis

1
```
#include <upc.h>
shared void *upc_all_alloc(size_t nblocks, size_t
nbytes);
nblocks : number of blocks
nbytes : block size
```

Description

1 **upc_all_alloc** is a collective function, with implied synchronization before all threads execute the function call.

2 **upc_all_alloc** allocates memory with a layout as if the following declaration were used:

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

3 The **upc_all_alloc** function returns the same pointer value on all threads.

4 The dynamic lifetime of an allocated object extends from the time any thread completes the call to **upc_all_alloc** until all threads have deallocated the object.

7.3.3 The **upc_local_alloc** function

Synopsis

1
```
#include <upc.h>
shared [] void *upc_local_alloc(size_t nblocks, size_t
nbytes);
nblocks : number of blocks
nbytes : block size
```

Description

1 Returns a pointer to **`nblocks * nbytes`** bytes of shared memory space with affinity to the calling thread and with type:

```
shared[] void *.
```

2 **`upc_local_alloc`** implies no synchronization with other threads.

3 **`upc_local_alloc`** is similar to **`malloc`**() except that it returns a shared pointer value. It is not a collective operation.

Constraints

1 The return value of the allocation functions shall be cast to a shared pointer of the correct block size.

7.3.4 The `upc_free` function

Synopsis

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

Description

1 The **`upc_free`** function frees the dynamically allocated shared memory 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_local_alloc`**, **`upc_global_alloc`**, or **`upc_all_alloc`** function, or if the space has been deallocated by a previous call to **`upc_free`**, the behavior is undefined.

7.3.5 The `upc_threadof` function

Synopsis

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

Description

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

7.3.6 The `upc_phaseof` function

Synopsis

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

Description

1 The `upc_phaseof` function returns the phase field of the shared pointer argument.

7.3.7 The `upc_addrfield` function

Synopsis

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

Description

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

7.4 Locks

7.4.1 Type

1 The type declared is

```
upc_lock_t
```

2 The type **`upc_lock_t`** is an opaque UPC type. Variables of type **`upc_lock_t`** are meant, therefore, to be manipulated through pointers only.

7.4.2 The `upc_lock_init` function

Synopsis

1

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

Description

1 Initializes the lock pointed to by **`ptr`**. After the **`upc_lock_init`** function is completed, the first thread calling the **`upc_lock`** function will succeed in obtaining this lock.

2 Intended to be called by one thread; no synchronization with other threads is implied.

7.4.3 The `upc_global_lock_alloc` function

Synopsis

1

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

Description

1 The **`upc_global_lock_alloc`** function dynamically allocates a lock and returns a pointer to it.

2 The lock pointed to is initialized in the same way **`upc_lock_init(upc_lock_t *ptr)`** would have done it.

3 Intended to be called by one thread; no synchronization with other threads is implied.

7.4.4 The `upc_all_lock_alloc` function

Synopsis

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

Description

1 The **`upc_all_lock_alloc`** function dynamically allocates a lock and returns a pointer to it. The **`upc_all_lock_alloc`** function is a collective function, with implied synchronization before all threads execute the function call. All threads receive the same pointer value.

2 The lock pointed to is initialized in the same way **`upc_lock_init(upc_lock_t *ptr)`** would have done it.

7.4.5 The `upc_lock` function

Synopsis

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

Description

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

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

7.4.6 The `upc_lock_attempt` function

Synopsis

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

Description

1 The `upc_lock_attempt` function tries to lock a shared variable, of type `upc_lock_t`, pointed to by the pointer given as argument.

2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0.

7.4.7 The `upc_unlock` function

Synopsis

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

Description

1 The `upc_unlock` function frees the lock and does not return any value.

7.5 Shared String Handling

7.5.1 The `upc_memcpy` function

Synopsis

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

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

Description

1 The **upc_memcpy** function copies a block of memory from one shared memory area to another shared memory area. The number of bytes copied is n. If copying takes place between objects that overlap, the behavior is undefined.

2 The **upc_memcpy** function treats the **dst** and **src** pointers as if each of them pointed to a shared memory space on a single thread and therefore had type:

shared [] char[n]

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

7.5.2 The **upc_memget** function

Synopsis

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

Description

1 The **upc_memget** function copies a block of memory from a shared memory area to a private memory area on the calling thread. The number of bytes copied is **n.** If copying takes place between objects that overlap, the behavior is undefined.

2 The **upc_memget** function treats the **src** pointer as if it pointed to a shared memory space on a single thread and therefore had type:

shared [] char[n]

The effect is equivalent to copying the entire contents from one shared array with this type (the **src** array) to a local array (the **dst** array) declared with the type **char[n]**.

7.5.3 The **upc_memput** function

Synopsis

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

Description

1 The **upc_memput** function copies a block of memory from the calling thread's private memory area to a shared memory area. The number of bytes copied is n. If copying takes place between objects that overlap, the behavior is undefined.

2 The **upc_memput** is equivalent to copying the entire contents from a local array (the **src** array) declared with the type

char[n]

to a shared array (the **dst** array) with the type

shared [] char[n]

7.5.4 The **upc_memset** function

Synopsis

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

Description

1 The **upc_memset** function copies the value of c, converted to an unsigned char, to a shared memory area. The number of bytes set is n.

2 The **upc_memset** function treats the **dst** pointer as if it pointed to a shared memory space on a single thread and therefore had type:

shared [] char[n]

The effect is equivalent to setting the entire contents of a shared array with this type (the dst array) to the value c.

References

[CARLSON99] 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/SEC00] ANSI. Programming Langauges-C. ISO/SEC 9899. May, 2000.

Appendix A: UPC versus ANSI C section numbering

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

Table A1. Mapping UPC subsection to ANSI C specifications subsections