UPC Runtime Design Utilizing Portals-4

Contents

List of Tables

Chapter 1

Authors and Revision Information

Authors: Gary Funck <gary@intrepid.com> Nenad Vukicevic <nenad@intrepid.com>

Intrepid Technology, Inc.

http://www.intrepid.com

http://www.gccupc.org

Revision: 1.2 (2012/10/18)

Chapter 2

Introduction

This document describes the design of a UPC runtime, which implements UPC (Unified Parallel C) semantics, and that makes use of the functions and operations provided by the Portals-4 API (Application Programming Interface). The Portals-4 operations are used primarily to implement access to remote computing nodes across a high-speed network.

Note
Portals-4 will simply be referred to as *Portals* throughout the remainder of this document.

Chapter 3

Overview

This design specification is organized into a series of topics that are arranged in an order that makes it easier to follow the core design discussions. This order of presentation necessarily introduces various low-level functions first that are needed by the following design discussions.

Chapter 4

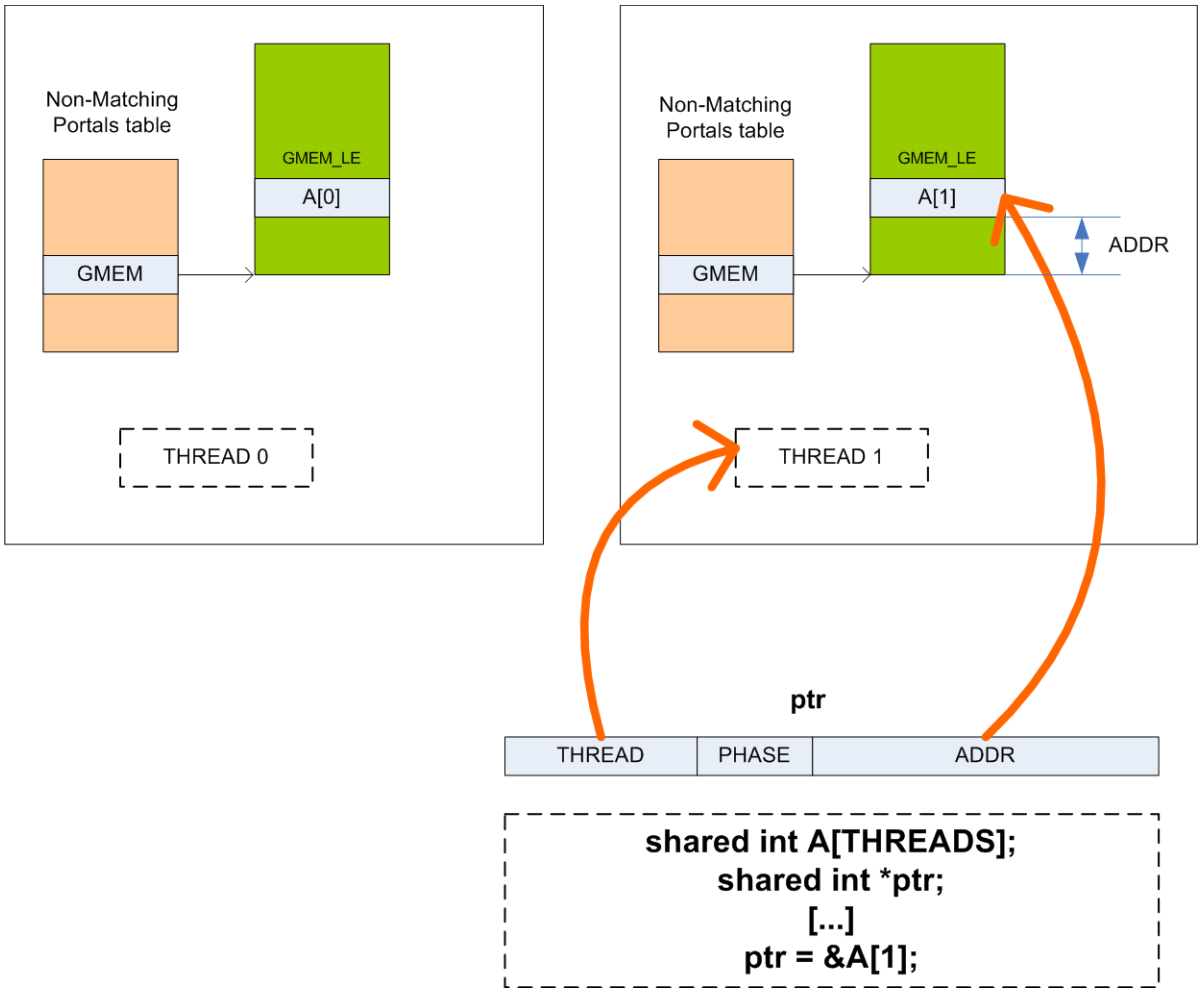
UPC Concepts

In UPC, all shared data items are located in a memory area called the *global segment*. Each UPC thread (operating system process) has its own global segment. This global segment contains the data values for all program-defined "UPC" variables that have been declared as *shared* qualified. The global segment will also contain all dynamically allocated UPC shared data.

For example,

```
shared double x;
shared int A[100*THREADS];
shared int *ptr;
```

Above, *x* is a double precision floating point scalar, *A* is an array with 100 elements allocated on each UPC thread, and *ptr* is pointer to a location in the global shared memory. The UPC specification defines *x* (and all shared scalars) as being located in the global segment of thread 0. Note that *ptr* itself is not located in shared memory, but rather it points to a location in shared memory. The (thread, offset) global memory address used by the UPC runtime to identify locations in shared memory can be derived directly from a UPC pointer-to-shared value.


Non-Matching
Portals table
GMEM_LE
A[0]
GMEM
THREAD 0
Non-Matching
Portals table
GMEM_LE
A[1]
ADDR
GMEM
THREAD 1
ptr
THREAD
PHASE
ADDR
shared int A[THREADS];
shared int *ptr;
[...]
ptr = &A[1];

Chapter 5

The UPC Shared Data Consistency Model

The *consistency model* describes the expected behavior demonstrated by a conforming UPC program when it makes accesses to variables or data that are *shared-qualified*. The formal definition of the UPC consistency model is defined on pages 11-12 and 101-103 of UPC Language Specification v1.2 [upc_lang_spec]. In this section, the UPC consistency model will be described.

5.1 Shared Memory Accesses

The following table lists the categories of UPC shared references that may be subject to differing constraints in the UPC shared memory consistency model.

Table 5.1: Categories of UPC Shared References

ID	Description
SW	strict shared memory write
SR	strict shared memory read
RW	relaxed shared memory write
RR	relaxed shared memory read
LW	local memory write
LR	local memory read

There are two restrictions related to UPC memory operations:

1. Strict accesses always appear (to all threads) to have executed in program order with respect to other strict accesses:
   - all relaxed accesses must complete before any strict access
   - all strict accesses must complete before any other strict or relaxed access
2. Any sequence of relaxed accesses issued by a given thread in an execution may appear to be arbitrarily reordered relative to program order by the implementation. The only exception to this rule 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.

Program execution and the relationship between strict and relaxed operations are described in Chapter 5.1.2.3 of [upc_lang_spec].

5.2 UPC Synchronization Operations

The memory consistency semantics of the synchronization operations are defined in terms of equivalent accesses to a unique shared variable that does not appear elsewhere in the program.

1. *upc_fence* is equivalent to a null SW followed by a null SR.
2. *upc_notify* is equivalent to a null SW before notification.
3. *upc_wait* is equivalent to a null SR after the statement completes.
4. *upc_lock* or a successful *upc_lock_attempt* is equivalent to a null SR immediately before return.
5. *upc_unlock* is equivalent to a null SW upon entry.

The memory consistency semantics of the synchronization operations are described in Chapter B.3.1 of [upc_lang_spec].

5.3 Non-Collective UPC Standard Library Calls

For non-collective functions in the UPC standard library (e.g. *upc_memget* and *upc_memput*) any implied data accesses to shared objects behave as a set of RR's and RW's 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.

Note
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.

5.4 UPC Run-time Memory Consistency Checks

The table below describes checks that need to be performed before any new shared memory operation can be initiated.

Table 5.2: Description of table columns in UPC Consistency Model Checks Table

OPERATION	Requested operation
ENTRY/EXIT	Wait condition applies before or after operation
SAME LOC	There is an outstanding operation to the same location as the access under consideration (same thread and address)
RW/RR/SW/SR	Outstanding operations that must complete before executing the current request

Table 5.3: Runtime Checks Dictated by the UPC Shared Memory Consistency Modal

OPERATION	ENTRY/EXIT	SAME LOC	RW	RR	SW	SR
RW	ENTRY	NO			x	x
		YES	x	x	x	x
RR	ENTRY	NO			x	x
		YES	x		x	x
SW	ENTRY		x	x	x	x
SR	ENTRY		x	x	x	x
upc_fence	ENTRY		x	x	x	x
upc_notify	ENTRY		x	x	x	x
upc_wait	EXIT		x	x	x	x
upc_lock	EXIT		x	x	x	x
upc_lock_attempt	EXIT (on success)		x	x	x	x
upc_unlock	ENTRY		x	x	x	x
upc_memget	ENTRY				x	x
upc_memput	ENTRY				x	x
upc_memcpy	ENTRY				x	x

The following pseudo code describes checks made for all outstanding requests that must be made before the current request can proceed.

The following abbreviations are used:

```
op_mode        (RW, RR, SW, SR)
L              target location (thread/address)
ANY_LOC        do not check the location address
```

```
upc_check_conflicts (op_mode, L)
{
  switch op_mode
    {
    case RR:
      complete_any_outstanding_strict_op ();
      complete_relaxed_ops (RW, L);
    case RW:
      complete_any_outstanding_strict_op ();
      complete_relaxed_ops (RR | RW, L);
    case SR | SW:
      complete_any_outstanding_strict_op ();
      complete_relaxed_ops (RR | RW, ANY_LOC);
    }
}
```

Above, the *complete_any_outstanding_strict_op* procedure will wait for an outstanding SR or SW operation that has been previously initiated. At most one strict operation can be outstanding. As a simplification, a UPC runtime by design might always issue strict operations synchronously (that is: the runtime would wait for the strict operation to complete, immediately after the operation has been initiated). In that case, the call to *complete_any_outstanding_strict_op* is unnecessary.

The following pseudo code describes the shared memory conflict checks that are made for outstanding synchronization operations and library calls before they are allowed to proceed. The UPC run-time implementation of these functions must perform the conflict check at the places indicated in the previous table (entry or exit).

```
upc_operations_check_conflicts (op)
{
```

```
  switch (op)
    {
    case UPC_FENCE:
      upc_check_conflicts (SW, ANY_LOC);
      upc_check_conflicts (SR, ANY_LOC);
    case UPC_NOTIFY:
      upc_check_conflicts (SW, ANY_LOC);
    case UPC_WAIT:
      /* performed upon exit */
      upc_check_conflicts (SR, ANY_LOC);
    case UPC_LOCK:
      /* performed upon exit */
      upc_check_conflicts (SR, ANY_LOC);
    case UPC_LOCK_ATTEMPT:
      /* performed upon exit, if successful */
      upc_check_conflicts (SR, ANY_LOC);
    case UPC_UNLOCK:
      upc_check_conflicts (SW, ANY_LOC);
    case UPC_MEMGET:
    case UPC_MEMPUT:
    case UPC_MEMCPY:
    case UPC_MEMSET:
      upc_check_conflicts (SW, ANY_LOC);
      upc_check_conflicts (SR, ANY_LOC);
    }
}
```

5.5 Network Ordering Effects on Memory Consistency

The checks for RR's and RW's to the same location can be eliminated if the network provides guarantees of read/write request ordering for requests issued to the same thread.

Note
A proposed update to the Portals specification will provide a guarantee for request ordering of transfers that are not greater than a specified length. This maximum length is likely to be on the order of eight (8) bytes.

For relaxed reads and writes where the transfer length is not greater than this configuration defined limit, the checks for accesses to a specific shared location can be eliminated.

Chapter 6

UPC Language Defined Operators

The UPC language defined operators are parsed by the GCC/UPC compiler front-end, and no calls to the UPC runtime will be made by the generated code. In most common usages of these operators, the result is a compile-time known integer constant and no code is generated. These operators will will require no change when the compiler and runtime are ported to a host running Portals. They are listed here for completeness.

6.1 The sizeof operator

```
sizeof unary-expression
sizeof ( type-name )
```

The *sizeof* operator when applied to *shared-qualified* types generally behaves as described in the C99 language specification. One difference is that 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.

6.2 The upc_localsizeof operator

```
upc_localsizeof unary-expression
upc_localsizeof ( type-name )
```

The *upc_localsizeof* operator shall apply only to *shared-qualified* expressions or *shared-qualified* types. All constraints on the *sizeof* operator also apply to this operator. The *upc_localsizeof* operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a *shared-qualified* type. It returns the same value on all threads; the value is an upper bound of the size allocated with affinity to any single thread and may include an unspecified amount of padding. The result of *upc_localsizeof* is an integer constant. The type of the result is *size_t*. If the operand is an expression, that expression is not evaluated.

6.3 The upc_blocksizeof operator

```
upc_blocksizeof unary-expression
upc_blocksizeof ( type-name )
```

The *upc_blocksizeof* operator shall apply only to *shared-qualified* expressions or *shared-qualified* types. All constraints on the *sizeof* operator also apply to this operator. The *upc_blocksizeof* operator returns the block size of the operand, which may be a shared object or a *shared-qualified* type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of *upc_blocksizeof* is an integer constant. If the operand of *upc_blocksizeof* has indefinite block size, the value of *upc_blocksizeof* is 0. The type of the result is *size_t*. If the operand is an expression, that expression is not evaluated.

6.4 The upc_elemsizeof operator

```
upc_elemsizeof unary-expression
upc_elemsizeof ( type-name )
```

The *upc_elemsizeof* operator shall apply only to *shared-qualified* expressions or shared-qualified types. All constraints on the *sizeof* operator also apply to this operator. The *upc_elemsizeof* operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For non-array objects, *upc_elemsizeof* returns the same value as *sizeof*. The result of *upc_elemsizeof* is an integer constant. The type of the result is *size_t*.

Chapter 7

UPC Language Defined Synchronization Statements

This section describes the semantics of the UPC language defined synchronization statements. This description of their UPC language required behavior is derived from the UPC Language Specification (version 1.2).

The GCC/UPC front-end compiles the UPC language defined synchronization statements into calls to the UPC runtime library barrier functions.

7.1 upc_notify

```
upc_notify;
upc_notify expression;
```

If the optional *expression* is given, it shall have type *int*. Any collective operations issued between a *upc_notify* and a *upc_wait* statement are prohibited. *upc_notify* is a collective operation. A null strict access is implied before a *upc_notify* statement. The null strict access will occur after the evaluation of expression, if present.

Each thread shall execute an alternating sequence of *upc_notify* and *upc_wait* statements, starting with a *upc_notify* and ending with a *upc_wait* statement.

After a thread executes *upc_notify* the next collective operation it executes must be a *upc_wait*. A synchronization phase consists of the execution of all statements between the completion of one *upc_wait* and the start of the next.

This 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.

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.

7.2 upc_wait

```
upc_wait;
upc_wait expression;
```

If the optional *expression* is given, it shall have type *int*. Any collective operations issued between a *upc_notify* and a *upc_wait* statement are prohibited. A null strict access is implied after a *upc_wait* statement. *upc_wait* is a collective operation.

Each thread shall execute an alternating sequence of *upc_notify* and *upc_wait* statements, starting with a *upc_notify* and ending with a *upc_wait* statement.

After a thread executes *upc_notify* the next collective operation it executes must be a *upc_wait*. A synchronization phase consists of the execution of all statements between the completion of one *upc_wait* and the start of the next.

This 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.

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. *upc_wait* and *upc_notify* are collective operations. 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.

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

7.3 upc_barrier

```
upc_barrier;
upc_barrier expression;
```

If the optional *expression* is given, it shall have type *int*. The barrier operations at thread startup and termination have a value of *expression* which is not in the range of the type *int*.

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

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

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

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

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

7.4 upc_fence

```
upc_fence;
```

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.

One implementation of *upc_fence* may be achieved by a null strict access:

```
{static shared strict int x; x = 0; x;}
```

Chapter 8

UPC Language Defined *upc_forall* Statement

```
upc_forall (expression-opt; expression-opt; expression-opt; affinity-opt)
  statement
upc_forall (declaration expression-opt; expression-opt; affinity-opt)
  statement

where:
  affinity: expression | 'continue'
```

upc_forall is a collective operation in which, for each execution of the loop body, the controlling expression and affinity expression are single-valued. The *affinity* field specifies the executions of the loop body which are to be performed by a thread.

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

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*.

When *affinity* is *continue* or not specified, each loop body of the *upc_forall* statement is performed by every thread.

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*.

Every thread evaluates the first three clauses of a *upc_forall* statement in accordance with the semantics of the corresponding C99 language definition clauses for the for statement. Every thread evaluates the fourth clause of every iteration.

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

Note that *single-valued* implies that all threads agree on the total number of iterations, their sequence, and which threads execute each iteration. This semantic implies that side effects on the same thread have defined behavior, just like in the *for* statement.

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.

Chapter 9

Portals Resources Used by the UPC Runtime

The following describes the Portals resources that will be used by the UPC runtime.

Table 9.1: Portals Resources Used by the UPC Runtime

Portals Resource	Type	Purpose
GUPCR_PTL_PTE_GMEM	PTE	Used to access a UPC thread's shared memory
GUPCR_PTL_PTE_LOCK	PTE	Used by the UPC lock implementation
GUPCR_PTL_PTE_BARRIER_DOWN	PTE	Used to signal during the "down" phase of a UPC barrier
GUPCR_PTL_PTE_BARRIER_UP	PTE	Used to signal during "up phase" of a UPC barrier
GUPCR_PTL_PTE_SHUTDOWN	PTE	Used to accept remote shutdown request
GUPCR_PTL_PTE_COLL	PTE	Used to implement UPC collective operations
upcr_gmem_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_GMEM
upcr_lock_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_LOCK
upcr_lock_le_eq	EQ	Event queue associated with GUPCR_PTL_PTE_LOCK
upcr_lock_le_ct	CT	Counting event associated with upcr_lock_le
upcr_wait_down_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_BARRIER_DOWN
upcr_wait_down_le_eq	EQ	Event queue associated with upcr_wait_down_le
upcr_wait_down_le_ct	CT	Counting event associated with upcr_wait_down_le
upcr_wait_up_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_BARRIER_UP
upcr_wait_up_le_eq	EQ	Event queue associated with upcr_wait_up_le
upcr_wait_up_le_ct	CT	Counting event associated with upcr_wait_up_le
upcr_shutdown_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_SHUTDOWN
upcr_shutdown_le_ct	CT	Counting event associated with upcr_shutdown_le
upcr_coll_le	LE	Portals List Entry (LE) associated with GUPCR_PTL_PTE_COLL
upcr_coll_le_eq	EQ	Event queue associated with upcr_coll_le
upcr_coll_le_ct	CT	Counting event associated with upcr_coll_le

Chapter 10

UPC Runtime Configuration-defined Limits and Constants

The definitions below describe limits and constants that may vary from one configuration to another. Although specific values are listed, they are intended only to show the typical range of the values.

```
/* Useful constants */
#define KB 1024
#define MB (KB * KB)
#define GB (MB * KB)

/* Configuration-defined limits */
#define UPCR_GMEM_MAX_SAFE_PUT_SIZE 1K
#define UPCR_GMEM_MAX_PORTALS_GET_SIZE 1*GB
#define UPCR_GMEM_MAX_PORTALS_PUT_SIZE 1*GB
#define UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE 256*KB
#define UPCR_MAX_ORDERED_SIZE ptl_ni_limits.max_ordered_size

/* Portals table indexes */
#define GUPCR_PTL_PTE_GMEM      10
#define GUPCR_PTL_PTE_LOCK      11
#define GUPCR_PTL_PTE_BARRIER_DOWN 12
#define GUPCR_PTL_PTE_BARRIER_UP   13
#define GUPCR_PTL_PTE_SHUTDOWN  14
#define GUPCR_PTL_PTE_COLL 15

#define PTL_NO_MATCH_BITS ((ptl_match_bits_t) 0)
#define PTL_NULL_USER_PTR ((void *) 0)
#define PTL_NULL_HDR_DATA ((ptl_hdr_data_t) 0)

/* UPCR_BARRIER_ID_MAX is a value that is guaranteed to be greater then
   the maximum allowed barrier ID.  UPCR_BARRIER_ID_MAX is used by
   the barrier wait logic to ensure that the PTL_MIN atomic operations
   will determine the minimum barrier ID that is in use during
   the current synchronization phase. */
#define UPCR_BARRIER_ID_MAX MAX_LONG_INT

/* UPC lock queue link block signal values */
#define UPCR_LOCK_NO_SIGNAL 0
#define UPCR_LOCK_SIGNAL 1
```

Chapter 11

UPC Runtime Data Structures

```
/* Global memory address type */
typedef struct upcr_gmem_addr_struct
  {
    ptl_process_t thread;
    ptl_size_t    offset;
  } upcr_gmem_addr_t;

/* List of MD mappings */
typedef struct upcr_gmem_md_list_struct *upcr_gmem_md_list_p;
typedef struct upcr_gmem_md_list_struct
  {
    upcr_gmem_md_list_p next;
    upcr_gmem_md_list_p prev;
    void *start;
    ptl_size_t length;
    ptl_handle_md_t md_handle;
  } upcr_gmem_md_list_entry_t;

/* Track the information required to access global
   memory in a given direction (get/put) using non-blocking
   counting forms of 'get' and 'put'.  */
typedef struct upcr_gmem_xfer_info_struct
  {
    ptl_size_t num_pending;
    ptl_size_t num_completed;
    unsigned int md_options;
    ptl_handle_eq_t eq_handle;
    ptl_handle_ct_t ct_handle;
    upcr_gmem_md_list_p md_list;
  } upcr_gmem_xfer_info_t;
typedef upcr_gmem_xfer_info_t *upcr_gmem_xfer_info_p;

/* UPC dynamic allocation related data structures.
   These are described here as they would appear
   in a UPC program.  UPC is chosen because it more
   clearly expresses the nature of the data structures.
   The shared heap manager might not be implemented in UPC.
   In that case, the following structure definitions
   might use different syntax and refer to different
   types, but the fields will have similar meanings
   and usage.  */
typedef struct upcr_heap_node_struct
  {
    shared struct upcr_heap_node_struct *next;
```

```
    shared struct upcr_heap_node_struct *prev;
    size_t size;
    int is_global;
  } upcr_heap_node_t;
typedef shared upcr_heap_node_t *upcr_heap_node_p;
typedef struct upcr_heap_struct
  {
    upcr_lock_t lock;
    upcr_heap_node_p head;
  } upcr_heap_t;
typedef shared upcr_heap_t *upcr_heap_p;

/* UPC lock related data structures.
   These data structures are shown as they
   might appear written in UPC.
   The UPC runtime may choose to reference shared data
   using a different internal representation
   for efficiency reasons.  */

/* A lock structure is allocated in the shared memory
   with affinity to the calling thread or thread 0
   if a collective function is being called. */
typedef struct upcr_lock {
  /* pointer to the last waiting thread */
  shared void *last;
  /* lock owner's pointer to its wait queue link.  */
  shared void *owner_link;
} upcr_lock_t;

/* UPC lock waiting queue link */
typedef struct upcr_lock_link {
  /* pointer to the next waiting thread */
  shared void *next;
  /* used by predecessor to signal ownership of the lock. */
  int signal;
} upcr_lock_link_t;
```

Chapter 12

UPC Runtime Global Variables

```
/* The portals NIC */
ptl_handle_ni_t upcr_nic;

/* The current UPC thread */
ptl_process_t upcr_my_thread;

/* Track event-counted get/put operations */
upcr_gmem_xfer_info_t upcr_gmem_gets;
upcr_gmem_xfer_info_t upcr_gmem_puts;

/* Put "bounce buffer" */
typedef char upcr_gmem_put_bounce_buffer_t[UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE];
upcr_gmem_put_bounce_buffer_t upcr_gmem_put_bounce_buffer;
size_t upcr_gmem_num_put_bounce_buffer_bytes_used;
ptl_handle_md_t upcr_gmem_put_bounce_buffer_md_handle;

/* Flag that indicates whether the previous put operation
   was a strict put operation.  */
int upcr_pending_strict_put;

/* UPC dynamic memory allocation global variables.
   These are described here as they would appear
   in a UPC program.  UPC is chosen because it more
   clearly expresses the nature of the data structures.
   The shared heap manager might not be implemented in UPC.
   In that case, the following variables
   might use different syntax and refer to different
   types, but they will perform similar functions.  */
shared upc_heap_p upcr_global_heap;
shared upc_heap_p shared upcr_local_heap[THREADS];

/* Variables used in the implementation of upc_notify
   and upc_wait.
   upcr_barrier_active: flag indicate that a thread has
                        executed a 'upc_notify' state.
   upcr_barrier_id:     barrier ID passed to 'upc_notify'.
                        This will be matched against the barrier ID
                        of a subsequent 'upc_wait' call.  */
bool upcr_barrier_active;
long int upcr_barrier_id;

/* Variables used in the implementation of UPC locks. */
ptl_le_t upcr_lock_le;
ptl_handle_eq_t upcr_lock_eq;
```

```
ptl_handle_ct_t upcr_lock_le_ct;
int counter_lock_ct;
```

Chapter 13

UPC Runtime Shared Memory Access Functions

The UPC runtime library accesses shared memory via a collection of routines which are called the *upcr_gmem* functions. These functions implement a distributed global address space, where each location is addressed by a *(thread, offset)* pair. The *thread* identifies a logical thread of execution; in UPC this thread of execution is mapped to an operating system-defined process. The *offset* is a byte offset into the designated thread's contribution to the distributed global memory.

In UPC, the programmer designates which data items will be located in a given thread's contribution to global memory. Some other languages, such as *Titanium* (a Java dialect) permit most objects in a given thread's address space to be shared.

When a UPC program begins execution, the UPC runtime will create a series of operating system processes which implement the semantics of each UPC thread. The UPC runtime will also allocate a global segment for each thread. This global segment contains the shared data for program-declared shared variables as well as an area reserved for dynamically allocated globally shared data. The global memory segment is further divided into potentially two additional areas, for UPC-defined *global* and *local* shared memory allocations. In UPC, each thread contributes shared data to a globally allocated object and for local allocations shared data is allocated only in the calling thread's global segment.

Given that all UPC shared data is collected into a single per-thread global segment, the UPC runtime (and the *upcr_gmem* functions) need to be able to address only a single memory region in a target process in order to transfer memory to/from all the shared memory that a given UPC thread has allocated either statically via a shared variable declaration, or dynamically via calls the UPC-defined shared memory allocation functions.

Reading memory from another thread's shared segment is known as a *get* operation, and writing is called a *put* operation. The thread that performs the *get* or *put* operation is known as the *initiator*. The thread that has affinity to the shared data that is being transferred is called the *target* thread.

The global memory access layer of the UPC runtime defines two classes of get and put operations:

- non-blocking operations with *completion counts* where several operations can be initiated without waiting for completion. When the initiator needs to wait for completion, it waits for the count of completed operations to reach the number of operations that have been initiated. *Get* operation completions are counted separately from *put* operations.
- non-blocking operations with *synchronization handles* where operations can be initiated, and the initiator can wait selectively for the completion of a given operation by supplying the *handle* that was returned by global memory access routines when the operation was initiated.

UPC can be implemented using only synchronous *get* and *put* operations. However, for some UPC programs, performance can be improved by overlapping unrelated computations with the transfer of shared data over the network. In that scenario the UPC compiler will generate code that pre-fetches data before it is needed, and then calls a synchronization routine at the point the shared data value is needed. For these compiler-defined pre-fetch operations, completion count synchronization is both sufficient and likely has better performance. In addition to pre-fetches, the UPC-defined memory consistency model permits certain put operations to be overlapped with other program computation. Here again, transfer completion counts are sufficient to provide the required synchronization.

Synchronization handles are both more general, and likely will require more overhead than synchronization using completion counts. A UPC compiler/runtime is likely not to use synchronization handles, unless that is the only synchronization method

provided by the global memory access routines. Synchronization handles might be used by a runtime or library developer, or an advanced applications programmer.

Note
Although this document describes the interfaces for generalized non-blocking memory shared memory access functions that return handles, their design is not described in detail in this document because those more general non-blocking operations (with handles) are not needed by the UPC runtime described in this document.

The basic *get* and *put* operations are defined as asynchronous with event count completion. Thus, if the caller requires synchronous semantics, each get and put operation must be followed by a call to one of the synchronization functions described in the following insert.

The basic one-sided global memory *get* and *put* operations (with completion count semantics) will be implemented using a single Portals Table Entry (PTE) on the target Associated with `GUPCR_PTL_PTE_GMEM` will be a single persistent Portals List Entry (LE) named `upcr_gmem_le`. `upcr_gmem_le` describes the target thread's global shared memory segment.

On the initiator node, Portals counting events will be used to track the completion of *get* and *put* operations. In order to track *gets* and *puts*, separate Portals Memory Descriptors (MD's) will be used. To gain addressability to various regions of the initiator's local memory, MD's will be created on an as needed basis.

Note
A better performing implementation will use knowledge of the initiating thread's address space layout in order to minimize the need to create additional MD's.

Note
A proposed change to the Portals specification is being considered, which will support the mapping of a process's address space (inclusive of "holes") by a single memory descriptor (MD). If this capability is approved, then the UPC shared memory access functions will need to define only two memory descriptors: one to track *get* operations and another to track *put* operations.

The *upcr_gmem_put* and *upcr_gmem_copy* functions guarantee that the caller can immediately re-use the source argument's memory upon return from the function. The motivation for providing this guarantee is shown in the following UPC example.

```
shared x, y, z;
volatile int k = 1;
[...]
x = k;
k = k + 1;
y = k;
k = k + 1;
z = k;
```

This example might lead to compiled code that is equivalent to that shown below.

```
shared x, y, z;
volatile int k = 1;
[...]
upcr_gmem_put (&x, &k, sizeof (int));
k = k + 1;
upcr_gmem_put (&y, &k, sizeof (int));
k = k + 1;
upcr_gmem_put (&z, &k, sizeof (int));
upcr_gmem_sync_puts ();
```

Above, the values of *x*, *y*, and *z* should be 1, 2, and 3 respectively. Since *k* is the source argument to each call to *upcr_gmem_put*, and those put operations might proceed asynchronously, the possibility exists that the value of *k* is modified by subsequent *k = k + 1;* assignments before enough time has passed for the previous value of *k* to have been transmitted (or copied by the NIC). To prevent this race condition, *upcr_gmem_put* will copy its input argument, as long as the size of the argument is moderate. If the length of the argument is above a configuration defined limit, then the input argument is not copied; the operation will be executed synchronously.

An alternative design might require that the caller of *upcr_gmem_put* is responsible for making a copy of the input argument. This approach does not work well because the lifetime of the input argument may exceed the current procedural scope. This is shown in the example below.

```
put_and_return (shared int *dest, int src)
{
  int src_copy = src;
  upcr_gmem_put (dest, &src_copy, sizeof (int));
}
```

Upon return from the call to *put_and_return* the put operation might still be pending, but the *src_copy* variable has gone out-of-scope.

13.1 UPC Runtime Shared Memory Access Utility Functions

13.1.1 upcr_gmem_shared_offset_to_local_addr

```
void *upcr_gmem_shared_offset_to_local_addr (ptl_size_t offset);
```

upcr_gmem_shared_offset_to_local_addr converts an offset into the calling thread's global data segment into the address of the associated local memory.

The GCC/UPC runtime allocates the shared data segment when the runtime is initialized. The address of the local data addressed by the shared offset is simply the offset added to the local base address of the global segment.

13.1.2 upcr_gmem_map_addr_to_md

```
ptl_handle_md_t
upcr_gmem_map_addr_to_md (upcr_gmem_xfer_info_p xfer_info,
                          void *addr, ptl_size_t n);
```

upcr_gmem_map_addr_to_md will return a Memory Descriptor that will cover at least the memory area beginning at *start* spanning *n* bytes.

```
upcr_gmem_md_list_p
upcr_gmem_map_addr_to_md (upcr_gmem_xfer_info_p xfer_info,
                          void *addr, ptl_size_t n)
{
  upcr_gmem_md_list_p m;
  ptl_md_t md;
  char *last_addr_plus_1 = (char *)addr + n;
  for (m = xfer_info->md_list; m != NULL; m = m->next)
    {
      if (addr >= m->start
          && (last_addr_plus_1 - (char *)m->start) <= m->length)
      return m;
    }
  /* Create a new MD list entry and add it to the end of
     the chain, for transfers in a particular direction (get/put).  */
  m = (upcr_gmem_md_list_p) calloc (sizeof (md_list_entry_t), 1);
```

```
  m->prev = xfer_info->md_list.prev;
  m->next = xfer_info->md_list;
  xfer_info->md_list.prev = m;;
  m->start = addr;
  m->length = n;
  md.options = xfer_info->md_options;
  md.start = addr;
  md.length = n;
  /* All transfers in a given direction (get/put)
     share the same event queue, for processing failure
     notices, and the same counter event which records
     completion events.  */
  md.eq_handle = xfer_info->eq_handle;
  md.ct_handle = xfer_info->ct_handle;
  status = PtlMDBind (upcr_nic, &md, &m->md_handle);
  if (status != PTL_OK)
    upcr_fatal_portals_error (status);
  return m;
}
```

Note
In the code listing above, *xfer_info→md_options* will be initialized to *PTL_MD_EVENT_SUCCESS_DISABLE | PTL_MD_EVENT_CT_ACK*.

Note
A proposed change to Portals that would support binding the entire address space to a Memory Descriptor (MD) may remove the need to manage a collection of MD bindings; in that case *upcr_gmem_map_addr_to_md* would no longer be needed.

13.2 upcr_gmem_get

```
void upcr_gmem_get (void *dest, upcr_gmem_addr_t src, size_t n);
```

Copy the shared data at *src* to the local memory area at *dest*. The number of bytes to copy is given by *n*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory area designated by *src* is located in the global memory region that has affinity to the calling thread, this operation will be implemented as a local memory-to-memory copy. Upon return, the initiated get count may be incremented by one (1).

```
void upcr_gmem_get (void *dest, upcr_gmem_addr_t src, size_t n)
{
  if (src.thread == upcr_my_thread)
    {
      memcpy (dest, upcr_gmem_shared_offset_to_local_addr (src.offset), n);
    }
  else
    {
      char *dest_addr = (char *)dest;
      size_t n_rem = n;
      while (n_rem > 0)
        {
          size_t n_xfer;
          upcr_gmem_md_list_p m;
          int status;
          ptl_size_t local_offset;
          n_xfer = MIN (n_rem, UPCR_GMEM_MAX_PORTALS_GET_SIZE);
          ++upcr_gmem_gets.num_pending;
```

```
          m = upcr_gmem_map_addr_to_md (
                              &upcr_gmem_gets, dest_addr, n_xfer);
          local_offset = dest_addr - (char *)m->start;
          status = PtlGet (m->md_handle, local_offset, n_xfer,
                           src.thread, GUPCR_PTL_PTE_GMEM, PTL_NO_MATCH_BITS,
                           src.offset, PTL_NULL_USER_PTR);
          if (status != PTL_OK)
            upcr_fatal_portals_error (status);
          n_rem -= n_xfer;
          dest_addr += n_xfer;
        }
    }
  return;
}
```

13.3 upcr_gmem_put

```
void upcr_gmem_put (upcr_gmem_addr_t dest, void *src, size_t n);
```

Copy the local memory area at *src* to the global memory area at *dest*. The number of bytes to copy is given by *n*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory area designated by *dest* is located in the global memory region that has affinity to the calling thread, this operation will be implemented as a local memory-to-memory copy. Upon return, the memory area specified by the *src* argument can safely be re-used and the initiated put count will be incremented by one (1).

```
void upcr_gmem_put (upcr_gmem_addr_t dest, void *src, size_t n)
{
  if (dest.thread == upcr_my_thread)
    {
      memcpy (upcr_gmem_shared_offset_to_local_addr (dest.offset), src, n);
    }
  else
    {
      /* Large puts must be synchronous, to ensure that it is
         safe to re-use the source buffer upon return.  */
      bool must_sync = (n > UPCR_GMEM_MAX_SAFE_PUT_SIZE);
      char *src_addr = (char *)src;
      size_t n_rem = n;
      while (n_rem > 0)
        {
          size_t n_xfer;
          upcr_gmem_md_list_p m;
          ptl_handle_md_t md_handle;
          int status;
          ptl_size_t local_offset;
          n_xfer = MIN (n_rem, UPCR_GMEM_MAX_PORTALS_PUT_SIZE);
          if (must_sync)
            {
              m = upcr_gmem_map_addr_to_md (&upcr_gmem_puts,
                                            src_addr, n_xfer);
              local_offset = src_addr - (char *)m->start;
              md_handle = m->md_handle;
            }
          else
            {
              char *bounce_buf;
              /* If this transfer will overflow the bounce buffer,
                 then first wait for all outstanding puts to complete.  */
              if ((upcr_gmem_num_put_bounce_buffer_bytes_used + n_xfer)
```

```
                  > UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE)
                upcr_gmem_sync_puts ();
              bounce_buf = &upcr_gmem_put_bounce_buffer
                           [upcr_gmem_num_put_bounce_buffer_bytes_used];
              memcpy (bounce_buf, src_addr, n_xfer);
              md_handle = upcr_gmem_put_bounce_buffer_md_handle;
              local_offset = bounce_buf - upcr_gmem_put_bounce_buffer;
              upcr_gmem_num_put_bounce_buffer_bytes_used += n_xfer;
            }
          ++upcr_gmem_puts.num_pending;
          status = PtlPut (md_handle, local_offset, n_xfer,
                           PTL_ACK_REQ, dest.thread,
                           GUPCR_PTL_PTE_GMEM, PTL_NO_MATCH_BITS, dest.offset
                           PTL_NULL_USER_PTR, PTL_NULL_HDR_DATA);
          if (status != PTL_OK)
            upcr_fatal_portals_error (status);
          n_rem -= n_xfer;
          src_addr += n_xfer;
        }
      if (must_sync)
        upcr_gmem_sync_puts ();
    }
  return;
}
```

13.4 upcr_gmem_copy

```
void upcr_gmem_copy (upcr_gmem_addr_t dest, upcr_gmem_addr_t src, size_t n);
```

Copy the global memory area at *src* to the global memory area at *dest*. The number of bytes to copy is given by *count*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory areas designated by *dest* and *src* are located in the global memory region that has affinity to the calling thread, this operation may be implemented as a local memory-to-memory copy. Upon return, the memory area designated by the *src* argument can be safely re-used and both the initiated put count may be incremented by one (1).

```
void upcr_gmem_copy (upcr_gmem_addr_t dest, upcr_gmem_addr_t src, size_t n)
{
  if (dest.thread == upcr_my_thread && src.thread == upcr_my_thread)
    {
      memcpy (upcr_gmem_shared_offset_to_local_addr (dest.offset)
              upcr_gmem_shared_offset_to_local_addr (src.offset) , n);
    }
  else
    {
      size_t n_rem = n;
      while (n_rem > 0)
        {
          size_t n_xfer;
          char *bounce_buf;
          ptl_handle_md_t md_handle;
          ptl_size_t local_offset;
          int status;
          /* Use the entire put "bounce buffer" if the transfer
             count is sufficiently large.  */
          n_xfer = MIN (n_rem, UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE);
          if ((upcr_gmem_num_put_bounce_buffer_bytes_used + n_xfer)
              > UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE)
            upcr_gmem_sync_puts ();
          bounce_buf = &upcr_gmem_put_bounce_buffer
```

```
                        [upcr_gmem_num_put_bounce_buffer_bytes_used];
          upcr_gmem_num_put_bounce_buffer_bytes_used += n_xfer;
          /* Read the source data into the bounce buffer */
          upcr_gmem_get (bounce_buf, src, n_xfer);
          upcr_gmem_sync_gets ();
          md_handle = upcr_gmem_put_bounce_buffer_md_handle;
          local_offset = bounce_buf - upcr_gmem_put_bounce_buffer;
          ++upcr_gmem_puts.num_pending;
          status = PtlPut (md_handle, local_offset, n_xfer,
                           PTL_ACK_REQ, dest.thread,
                           GUPCR_PTL_PTE_GMEM, PTL_NO_MATCH_BITS, dest.offset
                           PTL_NULL_USER_PTR, PTL_NULL_HDR_DATA);
          if (status != PTL_OK)
            upcr_fatal_portals_error (status);
          n_rem -= n_xfer;
          src_addr += n_xfer;
        }
    }
  return;
}
```

13.5 upcr_gmem_set

```
void upcr_gmem_set (upcr_gmem_addr_t dest, int c, size_t n);
```

Fill the global memory area at *dest* with *n* bytes of the value given by the *c* argument. There is no address alignment restriction for the destination. If the shared memory area designated by *dest* is located in the global memory region that has affinity to the calling thread, this operation will be implemented as a local memory transfer. Upon return, the initiated put count may be incremented by one (1).

```
void upcr_gmem_set (upcr_gmem_addr_t dest, int c, size_t n)
{
  if (dest.thread == upcr_my_thread)
    {
      memset (upcr_gmem_shared_offset_to_local_addr (dest.offset), c, n);
    }
  else
    {
      size_t n_rem = n;
      bool already_filled = false;
      while (n_rem > 0)
        {
          size_t n_xfer;
          char *bounce_buf;
          ptl_handle_md_t md_handle;
          ptl_size_t local_offset;
          int status;
          /* Use the entire put "bounce buffer" if the transfer
             count is sufficiently large.  */
          n_xfer = MIN (n_rem, UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE);
          if ((upcr_gmem_num_put_bounce_buffer_bytes_used + n_xfer)
              > UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE)
            upcr_gmem_sync_puts ();
          bounce_buf = &upcr_gmem_put_bounce_buffer
                        [upcr_gmem_num_put_bounce_buffer_bytes_used];
          upcr_gmem_num_put_bounce_buffer_bytes_used += n_xfer;
          /* Fill the bounce buffer, if we haven't already  */
          if (!already_filled)
            {
```

```
              memset (bounce_buf, c, n_xfer);
              already_filled = (bounce_buf == upcr_gmem_put_bounce_buffer
                                && n_xfer == UPCR_GMEM_PUT_BOUNCE_BUFFER_SIZE);
            }
          md_handle = upcr_gmem_put_bounce_buffer_md_handle;
          local_offset = bounce_buf - upcr_gmem_put_bounce_buffer;
          ++upcr_gmem_puts.num_pending;
          status = PtlPut (md_handle, local_offset, n_xfer,
                           PTL_ACK_REQ, dest.thread,
                           GUPCR_PTL_PTE_GMEM, PTL_NO_MATCH_BITS, dest.offset,
                           PTL_NULL_USER_PTR, PTL_NULL_HDR_DATA);
          if (status != PTL_OK)
            upcr_fatal_portals_error (status);
          n_rem -= n_xfer;
          src_addr += n_xfer;
        }
    }
  return;
}
```

13.6 upcr_gmem_sync

```
void upcr_gmem_sync (void);
```

Wait for all asynchronous counting *get* and *put* operations to complete. All operations are considered complete when the completed get count is equal to the initiated get count and the completed put count is equal to the initiated put count.

```
void upcr_gmem_sync (void)
{
  if (upcr_gmem_gets.num_pending > 0)
    upcr_gmem_sync_gets ();
  if (upcr_gmem_puts.num_pending > 0)
    upcr_gmem_sync_puts ();
}
```

13.7 upcr_gmem_sync_gets

```
void upcr_gmem_sync_gets (void);
```

Wait for all asynchronous counting get operations to complete. All get operations are considered complete when the completed get count is equal to the initiated get count.

```
void upcr_gmem_sync_gets (void)
{
  if (upcr_gmem_gets.num_pending > 0)
    {
       int status;
       ptl_size_t num_initiated = upcr_gmem_gets.num_completed
                                  + upcr_gmem_gets.num_pending;
       ptl_ct_event_t ct;
       status = PtlCTWait (upcr_gmem_gets.ct_handle, num_initiated, &ct);
       if (status != PTL_OK)
         upcr_fatal_portals_error (status);
       upcr_gmem_gets.num_pending = 0;
       upcr_gmem_gets.num_completed = num_initiated;
       if (ct.fail > 0)
```

```
          upcr_gmem_process_fail_events ();
    }
}
```

13.8 upcr_gmem_sync_puts

```
void upcr_gmem_sync_puts (void);
```

Wait for all asynchronous counting *put* operations to complete. All put operations are considered complete when the completed put count is equal to the initiated put count.

```
void upcr_gmem_sync_puts (void)
{
  if (upcr_gmem_puts.num_pending > 0)
    {
       int status;
       ptl_size_t num_initiated = upcr_gmem_puts.num_completed
                                  + upcr_gmem_puts.num_pending;
       ptl_ct_event_t ct;
       status = PtlCTWait (upcr_gmem_puts.ct_handle, num_initiated, &ct);
       if (status != PTL_OK)
         upcr_fatal_portals_error (status);
       upcr_gmem_puts.num_pending = 0;
       upcr_gmem_puts.num_completed = num_initiated;
       upcr_pending_strict_put = 0;
       upcr_gmem_num_put_bounce_buffer_bytes_used = 0;
       if (ct.fail > 0)
         upcr_gmem_process_fail_events ();
    }
}
```

Chapter 14

GCC/UPC Compiler-Runtime Interface

The GCC/UPC compiler generates code that calls various UPC runtime library routines to implement the semantics of global shared memory accesses and barriers. These runtime library routines are called in the following situations:

- the compiled UPC program accesses global UPC shared memory via references to variables that have been *shared* qualified, or by indirection through a UPC pointer-to-shared value.
- the compiled UPC program executes UPC language statements such as *upc_barrier* or *upc_wait*.

The get/put functions called by compiler generated code do not have explicit synchronization semantics. At present, GCC/UPC (version 4.5.1.2 released in November, 2010) will not generate code that attempts to pre-fetch *shared relaxed* values for example. Therefore, the current runtime API does not define shared memory access synchronization functions.

In the design described in this section the *get* routines will be defined as synchronous (blocking) and the *put* routines will be asynchronous (non-blocking) to the degree permitted by the UPC shared memory consistency model.

As described in the UPC consistency model section, relaxed operations can be executed asynchronously with respect to each other as long as there are no pairs of read and write operations (issued in any order on the same thread) that do not conflict on the same memory location. Further, a single strict *put* operation can be initiated without waiting for its completion, though any subsequent shared memory operation must first wait for the strict *put* operation to complete.

If the UPC runtime does implement the relaxed/strict puts as non-blocking operations (described below) then the UPC runtime library routines that implement *put* operations must ensure that the source data value is first copied to a location ("bounce buffer") that will not be modified during the course of the data transmission.

14.1 Memory Fences

Apart from their UPC language-defined consistency behavior, the *get* and *put* functions must ensure correct local processor cache consistency, and must provide hints to the compiler that particular operations have an implied "read fence" or "write fence" (that prevents gets/puts from being moved either above or below the fence). The interaction between these fence operations and UPC constructs are described in [upc_smp_impl].

The discussion below, will refer to these fence functions. In this example, the memory fence implementation for an *x86_64* target platform is shown.

```
#if defined (__x86_64__)
#define UPCR_WRITE_FENCE() asm __volatile__ ("mfence":::"memory")
#define UPCR_READ_FENCE() asm __volatile__ ("":::"memory")
#else
#error no memory fence operations defined for this architecture
#endif
/* memory barrier */
#define UPCR_FENCE() { UPCR_READ_FENCE (); UPCR_WRITE_FENCE (); }
```

14.2 UPC Shared Access Routine Naming Conventions

The UPC compiler will generate calls to UPC runtime library functions that perform gets/puts to UPC shared memory. These shared memory access functions follow a naming convention that agrees with the naming conventions used by the GNU C compiler (gcc) when compiled code must access runtime library functions in order to perform the desired operations. These runtime library routines have names of the form <*op*><*type*><*nargs*>. The table below lists the UPC library routine operation (<*op*>) names and their purpose.

Table 14.1: UPC Runtime Library Memory Access Function Prefix

Operation Name Prefix	**Purpose**
__get	Relaxed Read from UPC shared memory
__gets	Strict Read from UPC shared memory
__getblk	Relaxed Read from a block of UPC shared memory
__getsblk	Strict Read from a block of UPC shared memory
__put	Relaxed Write to UPC shared memory
__puts	Strict Write to UPC shared memory
__putblk	Relaxed Write to a block of UPC shared memory
__putsblk	Strict Write to a block of UPC shared memory
__copyblk	Relaxed Copy of a block of UPC shared memory from one shared memory location to another
__copysblk	Strict Copy of a block of UPC shared memory

The type of the argument passed to/from the UPC compiler-defined memory access routines is described by a sequence of two letters. These two letter sequences are listed in the table below.

Table 14.2: UPC Runtime Library Memory Access Operand Type Codes

Type Code	**Type Description**
qi	signed byte
hi	signed half word (16 bits)
si	signed word (32 bits)
di	signed double word (64 bits)
ti	signed terra word (128 bits)
sf	single float (32 bits)
df	double float (64 bits)
tf	terra float (128 bits)
xf	extended float (96 bits)

14.3 Shared *Relaxed Get* Access Routines

For UPC-defined *relaxed shared* memory reads (gets), the GCC/UPC compiler generates calls to one/more of the following functions.

```
u_intQI_t __getqi2 (upcr_shared_ptr_t src);
u_intHI_t __gethi2 (upcr_shared_ptr_t src);
u_intSI_t __getsi2 (upcr_shared_ptr_t src);
u_intDI_t __getdi2 (upcr_shared_ptr_t src);
u_intTI_t __getti2 (upcr_shared_ptr_t src);
```

```
float __getsf2 (upcr_shared_ptr_t src);
double __getdf2 (upcr_shared_ptr_t src);
long double __gettf2 (upcr_shared_ptr_t src);
long double __getxf2 (upcr_shared_ptr_t src);
void __getblk3 (void *dest, upcr_shared_ptr_t src, size_t n);
```

The relaxed shared memory read (get) functions will call *upcr_gmem_get* in order to transfer shared memory to local memory. Below, *getsi2* is shown (it reads a 4-byte value from shared memory).

```
u_intSI_t __getsi2 (upcr_shared_ptr_t p)
{
  upcr_gmem_addr_t addr = upcr_sptr_to_gmem_addr (p);
  u_intSI_t result;
  if (upcr_pending_strict_put)
    upcr_gmem_sync_puts ();
  upcr_gmem_get (&result, addr, sizeof (result));
  upcr_gmem_sync_gets ();
  return result;
}
```

Note
The logic in *upcr_gmem_get* that deals with checking for the length of the source operand is unneeded by the various runtime *get* functions, because of the limited and known size of the source argument. Also, given that the current UPC compiler runtime API supports only blocking gets, a simplified form of *upcr_gmem_get* may be defined when the UPC runtime is implemented, or Portals might be called directly.

Note
A proposed change to Portals that would support binding the entire address space to a Memory Descriptor (MD) would lead to further simplifications.

14.4 Shared *Relaxed Put* Access Routines

For UPC-defined "relaxed shared memory writes (puts)", the GCC/UPC compiler generates calls to one/more of the following functions.

```
void __putqi2 (upcr_shared_ptr_t dest, u_intQI_t v);
void __puthi2 (upcr_shared_ptr_t dest, u_intHI_t v);
void __putsi2 (upcr_shared_ptr_t dest, u_intSI_t v);
void __putdi2 (upcr_shared_ptr_t dest, u_intDI_t v);
void __putti2 (upcr_shared_ptr_t dest, u_intTI_t v);
void __putsf2 (upcr_shared_ptr_t dest, float v);
void __putdf2 (upcr_shared_ptr_t dest, double v);
void __puttf2 (upcr_shared_ptr_t dest, long double v);
void __putxf2 (upcr_shared_ptr_t dest, long double v);
void __putblk3 (upcr_shared_ptr_t src, void *dest, size_t n);
void __copyblk3 (upcr_shared_ptr_t dest, upcr_shared_ptr_t src, size_t n);
```

The relaxed shared memory write (put) functions will call *upcr_gmem_put* in order to transfer local memory to shared memory. Below, *putsi2* and *copyblk3* are shown.

```
void
__putsi2 (upcr_shared_ptr_t p, u_intSI_t v)
{
  upcr_gmem_addr_t addr = upcr_sptr_to_gmem_addr (p);
  if (sizeof (v) <= UPCR_MAX_ORDERED_SIZE)
```

```
    {
      if (upcr_pending_strict_put)
        upcr_gmem_sync_puts ();
      /* Ordered puts can proceed in parallel.  */
      upcr_gmem_put (addr, &v, sizeof (v));
    }
  else
    {
      /* Wait for any outstanding ordered puts */
      upcr_gmem_sync_puts ();
      upcr_gmem_put (addr, &v, sizeof (v));
      /* This put is unordered, we have to execute
         it as a blocking put.  */
      upcr_gmem_sync_puts ();
    }
}

void __copyblk3 (upcr_shared_ptr_t dest, upcr_shared_ptr_t src, size_t n)
{
  upcr_gmem_addr_t dest_addr = upcr_sptr_to_gmem_addr (dest);
  upcr_gmem_addr_t src_addr = upcr_sptr_to_gmem_addr (src);
  if (n <= UPCR_MAX_ORDERED_SIZE)
    {
      if (upcr_pending_strict_put)
        upcr_gmem_sync_puts ();
      /* Ordered copies can proceed in parallel.  */
      upcr_gmem_copy (dest_addr, src_addr, n);
    }
  else
    {
      /* Wait for any outstanding ordered puts */
      upcr_gmem_sync_puts ();
      upcr_gmem_copy (dest_addr, src_addr, n);
      /* This copy is unordered, we have to execute
         it as a blocking put.  */
      upcr_gmem_sync_puts ();
    }
}
```

Note
The logic in *upcr_gmem_put* that deals with checking for the length of the source operand is unneeded by the various runtime *put* functions, because of the limited and known size of the source argument. Thus, a simplified form of *upcr_gmem_put* and may be defined when the UPC runtime is implemented, or Portals might be called directly.

Note
A proposed change to Portals that would support binding the entire address space to a Memory Descriptor (MD) would lead to further simplifications of *upcr_gmem_copy*.

14.5 Shared *Strict Get* Access Routines

For UPC-defined *strict shared* memory reads (gets), the GCC/UPC compiler generates calls to one/more of the following functions.

```
u_intQI_t __getsqi2 (upcr_shared_ptr_t src);
u_intHI_t __getshi2 (upcr_shared_ptr_t src);
```

```
u_intSI_t __getssi2 (upcr_shared_ptr_t src);
u_intDI_t __getsdi2 (upcr_shared_ptr_t src);
u_intTI_t __getsti2 (upcr_shared_ptr_t src);
float __getssf2 (upcr_shared_ptr_t src);
double __getsdf2 (upcr_shared_ptr_t src);
long double __getstf2 (upcr_shared_ptr_t src);
long double __getsxf2 (upcr_shared_ptr_t src);
void __getsblk3 (void *dest, upcr_shared_ptr_t src, size_t n);
```

The strict shared memory read (get) functions will call *upcr_gmem_get* in order to transfer shared memory to local memory. Below, *getssi2* is shown (it reads a 4-byte value from shared memory.

```
u_intSI_t __getssi2 (upcr_shared_ptr_t p)
{
  upcr_gmem_addr_t addr = upcr_sptr_to_gmem_addr (p);
  u_intSI_t result;
  /* wait for all outstanding gets/puts */
  upcr_gmem_sync ();
  UPCR_FENCE ();
  upcr_gmem_get (&result, addr, sizeof (result));
  upcr_gmem_sync_gets ();
  UPCR_READ_FENCE ();
  return results;
}
```

Note
The logic in *upcr_gmem_get* that deals with checking for the length of the source operand is unneeded by the various runtime *get* functions, because of the limited and known size of the source argument. Also, given that the current UPC compiler runtime API supports only blocking gets, a simplified form of *upcr_gmem_get* may be defined when the UPC runtime is implemented, or Portals might be called directly.

Note
A proposed change to Portals that would support binding the entire address space to a Memory Descriptor (MD) would lead to further simplifications.

14.6 Shared *Strict Put* Access Routines

For UPC-defined *strict shared* memory writes (puts), the GCC/UPC compiler generates calls to one/more of the following functions.

```
void __putsqi2 (upcr_shared_ptr_t dest, u_intQI_t v);
void __putshi2 (upcr_shared_ptr_t dest, u_intHI_t v);
void __putssi2 (upcr_shared_ptr_t dest, u_intSI_t v);
void __putsdi2 (upcr_shared_ptr_t dest, u_intDI_t v);
void __putsti2 (upcr_shared_ptr_t dest, u_intTI_t v);
void __putssf2 (upcr_shared_ptr_t dest, float v);
void __putsdf2 (upcr_shared_ptr_t dest, double v);
void __putstf2 (upcr_shared_ptr_t dest, long double v);
void __putsxf2 (upcr_shared_ptr_t dest, long double v);
void __putsblk3 (upcr_shared_ptr_t dest, void *src, size_t n);
void __copysblk3 (upcr_shared_ptr_t dest, upcr_shared_ptr_t src, size_t n);
```

The strict shared memory write (put) functions will call *upcr_gmem_put* in order to transfer local memory to shared memory.

Below, *getsi2* and *copysblk3* are shown.

```
void
__putssi2 (upcr_shared_ptr_t p, u_intSI_t v)
{
  upcr_gmem_addr_t addr = upcr_sptr_to_gmem_addr (p);
  /* wait for all outstanding gets/puts */
  upcr_gmem_sync ();
  UPCR_WRITE_FENCE ();
  upcr_gmem_put (addr, &v, sizeof (v));
  UPCR_FENCE ();
  /* A single strict put is allowed to be
     issued as a non-blocking put.  */
  upcr_pending_strict_put = 1;
}
```

```
void __copysblk3 (upcr_shared_ptr_t dest, upcr_shared_ptr_t src, size_t n)
{
  upcr_gmem_addr_t dest_addr = upcr_sptr_to_gmem_addr (dest);
  upcr_gmem_addr_t src_addr = upcr_sptr_to_gmem_addr (src);
  /* wait for all outstanding gets/puts */
  upcr_gmem_sync ();
  UPCR_WRITE_FENCE ();
  upcr_gmem_copy (dest_addr, src_addr, n);
  UPCR_FENCE ();
  upcr_pending_strict_put = 1;
}
```

Note
The logic in *upcr_gmem_put* that deals with checking for the length of the source operand is unneeded by the various runtime *put* functions, because of the limited and known size of the source argument. Thus, a simplified form of *upcr_gmem_put* and may be defined when the UPC runtime is implemented, or Portals might be called directly.

Note
A proposed change to Portals that would support binding the entire address space to a Memory Descriptor (MD) would lead to further simplifications of *upcr_gmem_copy*.

14.7 UPC Runtime Support for Barriers

The GCC/UPC compiler generates calls to the following UPC runtime functions to implement the UPC synchronization statements described earlier.

```
void __upc_notify (int barrier_id);
void __upc_wait (int barrier_id);
void __upc_barrier (int barrier_id);
void __upc_fence ();
```

The UPC runtime barrier implementation uses an "all reduce" algorithm as outlined in the paper *Enabling Flexible Collective Communication Offload with Triggered Operations* by Keith Underwood et al. January, 2007 [triggered_ops]. Portals *atomic* and *triggered atomic* operations are used to propagate and verify that all UPC threads have entered the same synchronization phase with matching barrier ID's.

For the purposes of implementing UPC barriers, all UPC threads in a given job are organized as a tree. Thread 0 is the *root thread* (at the top of the tree). Other threads represent either an *inner thread* (which has at least one child), or a *leaf thread* (which has no children).

A UPC barrier is implemented in two phases: a *notify phase* and a *wait phase*. The UPC barrier implementation supports a split phase barrier, where a thread completes its wait state once all threads have entered the notify state of the barrier.

In the notify phase, all threads agree on the minimal barrier ID among themselves by using the Portals PTL_MIN atomic function in order to propagate this value to the root node (thread 0). In the wait phase, the agreed on barrier ID is pushed down the tree to all threads, which in turn check this value against their own barrier ID. An error is raised if there is a mismatch.

The Portals split phase barrier implementation uses Portals triggered functions to accomplish barrier ID transfer among various threads in the tree. This also guarantees that communication of the barrier ID down the tree proceeds without the thread's involvement.

The *all reduce* algorithm uses Portals *counting events*, *triggered atomic* operations, and *triggered put* operations. Two non-matching Portals Table Entries (PTE's) are used to implement the barrier:

1. `GUPCR_PTL_PTE_BARRIER_UP` is used to implement the notify phase of the barrier.
2. `GUPCR_PTL_PTE_BARRIER_DOWN` is used to implement the wait phase of the barrier.

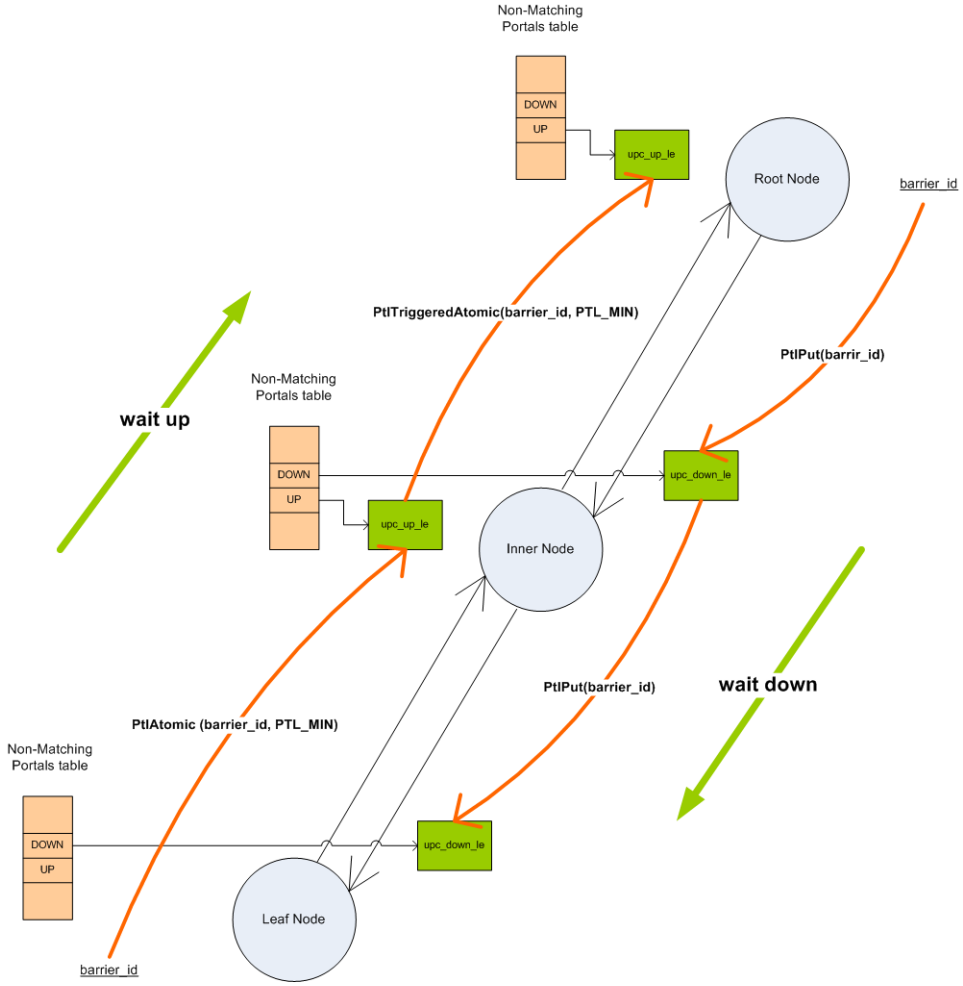
The *notify phase* does the following:

1. Setup triggered function for up the tree barrier ID traversal.
2. Setup triggered functions for down the tree barrier ID traversal.
3. Use atomic PTL_MIN function to agree on a minimal value of the barrier ID among itself and its children.

The *wait phase* does the following:

1. Receive the derived minimal agreed on barrier ID from the parent
2. Re-initialize the Portals event counters used for triggered functions
3. Verify the derived minimal agreed on barrier ID against its own ID

The following figure illustrates how the UPC runtime interacts with Portals to efficiently implement the barrier statement. The "wait up" path is associated with the notify part of the barrier, while "wait down" is associated with the wait part.



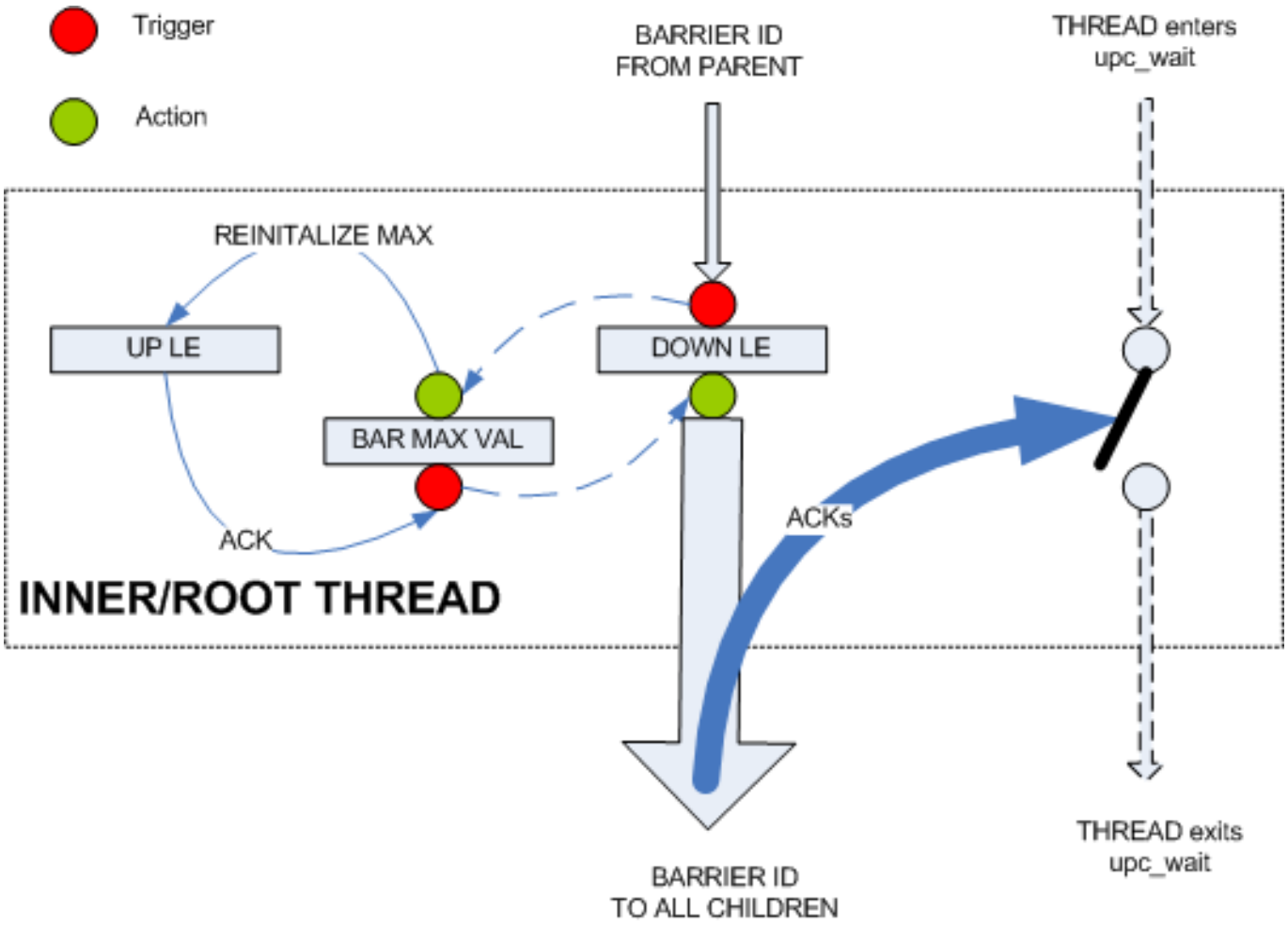
14.7.1 __upc_notify

The *__upc_notify* function records the current barrier ID and prepares the necessary support for barrier ID traversal using the thread's up and down PTEs. Than, it initiates the notify process by using the Portals atomic PTL_MIN function to send the thread's barrier ID to the parent.

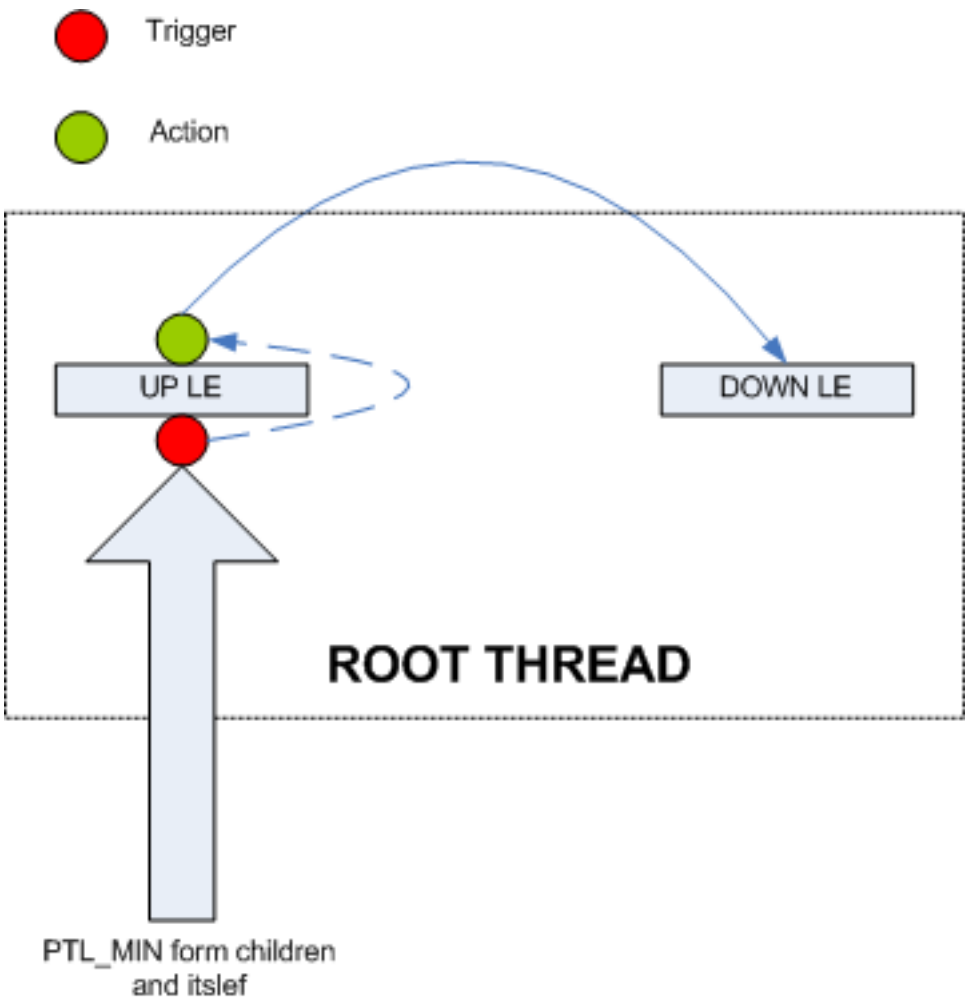
```
void
__upc_notify (int barrier_id)
{
  if (upcr_barrier_active)
    upcr_error ("already in barrier synchronization phase");
  upcr_barrier_id = barrier_id;
  upcr_barrier_active = TRUE;

  if (LEAF_THREAD)
    {
      /* Atomic PTL_MIN Put of barrier ID to BARRIER UP PTE of the parent. */
    }
  else
    {
      if (ROOT_THREAD)
        {
          /* Set up the Triggered PtlPut from the UP LE into
             the DOWN LE once derived minimal agreed on barrier ID
             arrives from all threads. */
        }
```

```
        /* Set up a Triggered Put from the buffer holding
           the MAX barrier ID to the UP LE of the thread once
           the barrier ID arrives in the DOWN LE. This
           step ensures that UP LE is properly setup for the next
           call to the barrier synchronization. */
        /* Set up the Triggered Puts from the DOWN LE buffer to all
           children once the previous step completes. Derived minimal
           agreed on barrier ID is being passed to the children. */
      if (INNER_THREAD)
        {
          /* Set up a Trigger Atomic PTL_MIN Put from the UP LE buffer
             into the UP LE of the parent once all children and the thread
             itself completed their Atomic PTL_MIN Puts to the
             UP LE of this thread. */
        }
        /* Send Atomic PTL_MIN Put of the barrier ID to its own
           UP LE. This step calculates the minimal barrier ID among
           the thread and its children. */
    }
}
```

The following diagram describes all the triggered operations on the inner thread (and partially the root thread).



The root thread has an additional triggered operation as presented in this diagram:



14.7.2 __upc_wait

In the wait phase, all the threads simply wait to receive the minimal agreed on barrier ID from their parent.

```
void
__upc_wait (barrier_id)
{
  if (!upcr_barrier_active)
    upcr_error ("program error");
  if (upcr_barrier_id != barrier_id)
    upcr_error ("program error");

  if (!LEAF_NODE)
    {
      /* Wait for the children to receive the barrier ID. This guaranties
         that all the triggered functions completed. */
    }
  else
    {
      /* Wait for barrier ID (received_barrier_id) to arrive in the
         DOWN LE buffer */
    }
  /* Re-initialize various counting events */

  if (received_barrier_id != barrier_id)
    upcr_error ("program error");
```

```
  upcr_barrier_active = FALSE;
}
```

14.7.3 __upc_barrier

The *__upc_barrier* function is a simple combination of *__upc_notify* and *__upc_wait*.

```
void
__upc_barrier (barrier_id)
{
  __upc_notify (barrier_id);
  __upc_wait (barrier_id);
}
```

14.7.4 __upc_fence

The *__upc_fence* function must complete all outstanding memory operations.

```
void __upc_fence ()
{
  UPCR_WRITE_FENCE ();
  upcr_gmem_sync ();
  UPCR_READ_FENCE ();
}
```

Chapter 15

UPC Library Functions

The UPC library functions are organized into the following groups:

- Bulk Shared Memory Copy Functions
- Dynamic Shared Memory Allocation
- Lock Functions
- Miscellaneous Functions

15.1 Bulk Shared Memory Copy Functions

The UPC Language Specification [upc_lang_spec] defines several library routines that copy shared memory to/from local memory, as well as between two shared memory areas. These routines can be considered to be generalizations of the non user-visible routines called by compiler-generated code (described in previous sections).

15.1.1 upc_memcpy

```
void upc_memcpy (upcr_shared_ptr_t dest, upcr_shared_ptr_t src, size_t n);
```

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.

The *upc_memcpy* function is implemented in the UPC runtime by a call to *upcr_gmem_copy*.

15.1.2 upc_memget

```
void upc_memget (void *dest, upcr_shared_ptr_t src, size_t n);
```

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

The *upc_memget* function is implemented in the UPC runtime by a call to *upcr_gmem_get*.

15.1.3 upc_memput

```
void upc_memput (upcr_shared_ptr_t dest, const void *src, size_t n);
```

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

The *upc_memput* function is implemented in the UPC runtime by a call to *upcr_gmem_put*.

15.1.4 upc_memset

```
void upc_memset (upcr_shared_ptr_t dest, int c, size_t n);
```

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*.

The *upc_memset* function is implemented in the UPC runtime by a call to *upcr_gmem_set*.

15.2 Dynamic Shared Memory Allocation

The UPC dynamic memory allocator obtains memory from the area of the global shared segment that is above the region reserved for declared UPC shared variables. This dynamic allocation memory space is further split between "global" allocations, where space is allocated across all threads and "local" allocations where space is allocated only from the shared data segment associated with the calling thread.

Thus, the global segment of each thread is divided into three parts:

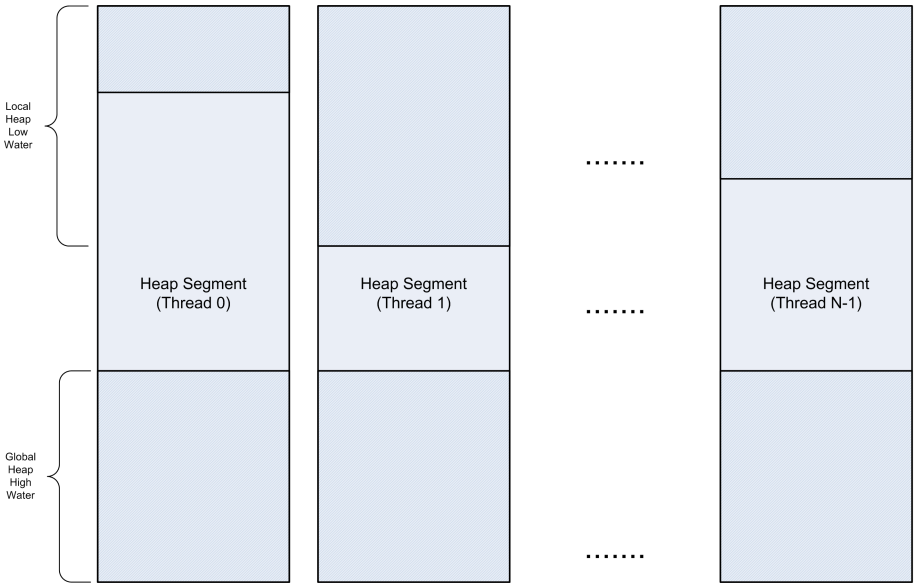
- The *program declared UPC shared variables* reside at the base of the global segment.
- The *global heap* allocations begin just after the region reserved for the program declared UPC shared variables.
- The *local heap* allocations begin at the top (highest address) of the global segment. The local heap grows downward.

Per the UPC language definition, the following constraints apply to the dynamic shared memory allocation functions:

- Any UPC thread can perform a global shared memory allocation. (Note that *upc_global_alloc* is not a collective operation.)
- Only the calling UPC thread can perform a local shared memory allocation.
- Any UPC thread can free the memory space allocated by a prior call to any of the dynamic shared memory allocation functions, independent of whether that memory space was allocated by a call to one of the global allocation functions, (*upc_global_alloc* or *upc_all_alloc*) or to a call to the *upc_alloc* local allocation function.

In order to ensure that local allocations do not overrun global allocations (and vice versa), the UPC dynamic memory allocator must maintain a global allocator *high water mark*, and a local allocator *low water mark*. When the allocator needs additional available shared space, it must ensure that the local allocator low water mark does not cross below the global allocator high water mark. To perform this check:

1. The allocator must consult a centralized record of both the high water mark and low water mark values. These two values will be stored in the global shared segment of thread 0, at a location known to the UPC runtime.
2. In order to ensure that the two values (the high and low water marks) are consistent, they must be queried and updated atomically. Given that the size of the global and local allocation regions might exceed the range of a 32-bit value, the high and low water mark values will hold only the top 32 bits of their actual value. These shifted values will be packed into a single 64-bit word so that this word can be queried and updated using a Portals defined atomic compare-and-swap operation. For example, if the range of the high and low water marks were constrained to 48 bits, then the low-order 16-bits would be assumed as zero and the resulting value would be shifted right 16 bits. The dynamic memory allocator would therefore ensure that when the heap regions are extended that they always grow in even increments of 2^{16} (65536 decimal).



Note
Maintaining the high and low water marks in a central location may create contention, for example, when the UPC program begins execution and all threads try to allocate dynamic memory at the same time. This can be avoided by providing an option for the runtime to statically partition the shared memory region reserved for dynamic allocation.

For global allocations, a central record of the state of the allocated and freed areas must be maintained. This centralized state of the global heap will be maintained in the global segment of thread 0. Further, when a given thread calls one of the UPC defined global allocation functions (*upc_global_alloc* or its collective form *upc_all_alloc*) or the *upc_free* function, a global heap data structure lock must first be acquired (to ensure that this data structure can be queried and updated without the risk of data corruption that might result from simultaneous updates).

For local allocations, a per-thread record of the state of the allocated and freed areas must be maintained. This per-thread record of the state of the local heap will be maintained in the global segment of the thread that calls *upc_alloc*. Further, when a given thread calls the local allocation routine, *upc_alloc*, or it calls *upc_free* to free previously allocated space that was originally allocated from a local heap, the calling thread must first acquire a lock associated the local heap that is the target of the called dynamic shared memory function.

Allocating memory from a heap requires searching a free list; freeing memory requires that adjacent entries in the free list are coalesced into a single free list entry. Since the free lists that are being operated on do not necessarily have affinity to the calling thread, it is important to choose a heap management algorithm that minimizes the number of potentially remote accesses required to search and update a heap free list.

The UPC runtime will utilize a *binary buddy algorithm* to search and update the heap free lists used to implement shared memory allocation and free operations. The binary buddy algorithm requires a relatively small and bounded number of memory accesses in order to search and coalesce the free list, which makes it a suitable choice.

15.2.1 UPC Runtime Broadcast Utility Functions

The broadcast utility functions are internal functions that are called by the UPC runtime when it is necessary to propagate (broadcast) a value to all other threads. Typically, this broadcast is used when implementing UPC library functions that are called as a collective, and are defined such that the same return value is passed back as a result of the library call on each thread. For example, upc_all_alloc and upc_all_lock_alloc will call these broadcast functions.

The broadcast support functions will use an algorithm that is a variant of the that used to implement <<*upc_wait,upc_wait*>>. The "up phase" of the broadcast procedure simply is used to indicate that each thread is ready to receive the broadcasted value from thread 0. The value is propagated in the "down phase".

Note
These internal broadcast utility functions are described here because they are first referred to by *upc_all_alloc*. They are not technically part of the implementation of the UPC dynamic shared memory allocator.

15.2.1.1 upcr_broadcast_get

```
void upcr_broadcast_get (void *value, size_t nbytes);
```

upcr_broadcast_get waits for the broadcasted value to arrive and then copies that value into the location given by *value* argument, for *nbytes* bytes.

Before returning, *upcr_broadcast_get* writes *value* into the broadcast bounce buffer of each of the children of this thread. The broadcast bounce buffer is in shared memory at a location known to each thread.

15.2.1.2 upcr_broadcast_put

```
void upcr_broadcast_put (void *value, size_t nbytes);
```

upcr_broadcast_put must be called only from thread 0. Thread 0 will issue a put into the broadcast bounce buffer of each of its children. The broadcast bounce buffer is in shared memory at a location known to each thread.

15.2.2 UPC Runtime Shared Memory Dynamic Allocation Utility Functions

The UPC runtime shared memory dynamic allocation utility functions are internal functions that are called to support the implementation of UPC's dynamic shared memory allocation library functions (for example, upc_global_alloc).

15.2.2.1 upcr_heap_alloc

```
shared void *upcr_heap_alloc (upcr_heap_p heap,
                              bool is_global,
                              shared void *(*heap_extend_func)
                                           (size_t nbytes),
                              size_t alloc_size);
```

upcr_heap_alloc is called by the various library functions that implement UPC dynamic shared memory allocation. The *heap* parameter is a pointer to a heap data structure used to manage a heap. If *is_global* is *true*, then the current allocation request is a global allocation (across all threads). The size of the allocation request is given by *alloc_size*.

If there is not enough free space to satisfy the allocation request, then the *heap_extend* callback procedure will be called; it will try to advance the relevant water mark by enough to satisfy the request. Typically, *heap_extend* is called to extend a heap by a fairly large "chunk" size. This minimizes the number of times that the heap extension procedure is called and consequently the number of times that the heap water marks have to be queried and updated.

The heap allocator proceeds as follows.

1. Lock the heap data structure using a global lock similar to those described in the Lock Functions section.
2. Attempt to allocate the object. If successful, return the newly allocated object.
3. If insufficient free space, call the 'heap_extend" function. If it does not enough additional storage to satisfy the allocation request, then return return a NULL pointer-to-shared indicating that the allocation request failed.
4. Set the *is_global* field in the newly allocated heap entry's header to the value specified by the *is_global* argument.
5. Unlock the heap data structure and return the pointer to the newly allocated storage.

As described in Dynamic Shared Memory Allocation, the heap allocator will employ a heap allocation algorithm that limits the number of shared memory accesses required to access and update the heap's free list. A *buddy algorithm* is the current candidate for use by the UPC heap allocator.

15.2.2.2 upcr_heap_free

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

upcr_heap_free is called to free previously dynamically allocated shared memory.

The freeing of dynamically allocated UPC shared memory proceeds as follows:

1. Decrement *ptr* so that it points to the heap entry header.
2. Query the *is_global* field in the heap entry. If set, then this storage was allocated globally. This flag is used to select either the global heap or the calling threads local heap data structure. The selected heap object will be accessed in the following steps.
3. Lock the heap object.
4. Return the allocated storage space to the heap object's free list.
5. Unlock the heap object, and return.

15.2.2.3 upcr_heap_global_extend

```
shared void *upcr_heap_global_extend (size_t nbytes);
```

upcr_heap_global_extend is a callback function called from *upcr_heap_alloc* to allocate an additional "chunk" of storage to the global heap when the heap allocator cannot find enough free space to satisfy the allocation request. If successful, *upcr_heap_global_extend* will return a pointer to the newly allocated space. The local copies of the heap low water and high water marks will have been updated to reflect the new heap extension.

The heap extension procedure is called by the heap allocator to allocate fairly large "chunks", which may be perhaps 4 megabytes or larger. The use of large chunks to extend a heap will lower the number of times that the heap extension procedure is called. Consequently, updates to the water marks should be a rare occurrence.

Since the water marks advance monotonically, it is safe for each thread to (atomically) read the current value of the water marks to see if the water marks have been already moved enough (to satisfy the current request) by another thread. After a thread has advanced the water marks, the new water mark levels may be enough to satisfy requests made by other threads; this will limit the need to update the global record of the water marks.

The low water and high water marks are encoded into a single 64 bit word, which is updated as follows.

1. Issue an atomic read to obtain the current low water/high water mark values.
2. If the new water marks are sufficient to satisfy the current heap allocation request, then return the base of the newly extended heap area.
3. Otherwise, using this newly fetched water mark value calculate a new tentative high water mark value by advancing the global heap high water mark by the amount requested.
4. Use the Portals compare-and-swap operation to compare the current water mark value to the value read, and if it still matches, then write the new value, and return the new heap base with the local copy of the water marks updated.
5. Otherwise, if the compare-and-swap fails, it means that some other thread has already updated the water marks. Go to step 1 to retry the heap extension procedure with the updated water marks.

15.2.2.4 upcr_heap_local_extend

```
shared void *upcr_heap_local_extend (size_t nbytes);
```

upcr_heap_local_extend is a callback function called from *upcr_heap_alloc* to allocate an additional "chunk" of storage to the local heap when the heap allocator cannot find enough free space to satisfy the allocation request. If successful, *upcr_heap_local_extend* will return a pointer to the newly allocated space. The local copies of the heap low water and high water marks will have been updated to reflect the new heap extension.

The heap extension procedure is called by the heap allocator to allocate fairly large "chunks", which may be perhaps 4 megabytes or larger. The use of large chunks to extend a heap will lower the number of times that the heap extension procedure is called. Consequently, updates to the water marks should be a rare occurrence.

Since the water marks advance monotonically, it is safe for each thread to (atomically) read the current value of the water marks to see if the water marks have been already moved enough (to satisfy the current request) by another thread. After a thread has advanced the water marks, the new water mark levels may be enough to satisfy requests made by other threads; this will limit the need to update the global record of the water marks.

The low water and high water marks are encoded into a single 64 bit word, with is updated as follows.

1. Issue an atomic read to obtain the current low water/high water mark values.
2. If the new water marks are sufficient to satisfy the current heap allocation request, then return the base of the newly extended heap area.
3. Otherwise, using this newly fetched water mark value calculate a new tentative high water mark value by decrementing the local heap low water mark by the amount requested.
4. Use the Portals compare-and-swap operation to compare the current water mark value to the value read, and if it still matches, then write the new value, and return the new heap base with the local copy of the water marks updated.
5. Otherwise, if the compare-and-swap fails, it means that some other thread has already updated the water marks. Go to step 1 to retry the heap extension procedure with the updated water marks.

15.2.3 upc_global_alloc

```
shared void *upc_global_alloc (size_t nblocks, size_t nbytes);
```

The upc_global_alloc function allocates shared space compatible with the declaration: shared [nbytes] char[nblocks * nbytes]. 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.

```
shared void *upc_global_alloc (size_t nblocks, size_t nbytes)
{
  size_t request_size = ROUND(nblocks, THREADS) * nbytes;
  size_t alloc_size = request_size / THREADS;
  shared void *mem = upcr_heap_alloc (upcr_global_heap,
                                      true /* is_global */,
                                      upcr_heap_global_extend,
                                      alloc_size);
  return mem;
}
```

Above, upcr_heap_alloc implements the heap allocator. upcr_heap_global_extend is a function that attempts to grow the global heap region by increasing the global heap high water mark. The upcr_global_heap variable holds a pointer to the heap object associated with the global heap.

15.2.4 upc_all_alloc

```
shared void *upc_all_alloc (size_t nblocks, size_t nbytes);
```

upc_all_alloc is a collective function with single-valued arguments. The *upc_all_alloc* function allocates shared space compatible with the following declaration: shared [nbytes] char[nblocks * nbytes]; 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.

```
shared void *upc_all_alloc (size_t nblocks, size_t nbytes)
{
  shared void *mem;
  /* Thread 0 allocates the space and broadcasts the
     pointer to the allocated storage space to all
     other threads.  */
  if (MYTHREAD == 0)
    {
      mem = upc_global_alloc (nblocks, nbytes);
      upcr_broadcast_put (&mem, sizeof (mem));
    }
  else
    {
      upcr_broadcast_get (&mem, sizeof (mem));
    }
  return mem;
}
```

Above, upcr_broadcast_put is called by thread 0 to propagate the indicated value to all other threads. These other threads call upcr_broadcast_get to retrieve this broadcasted value.

15.2.5 upc_alloc

```
shared void *upc_alloc (size_t nbytes);
```

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

```
shared void *upc_alloc (size_t nbytes)
{
  shared void *mem = upcr_heap_alloc (upcr_local_heap[MYTHREAD],
                                      false /* !is_global */,
                                      upcr_heap_local_extend,
                                      nbytes);
  return mem;
}
```

Above, upcr_heap_alloc implements the heap allocator and upcr_heap_local_extend is a function that attempts to grow the local heap region by decreasing the local heap low water mark, if necessary.

The *upcr_local_heap[MYTHREAD]* expression uses the current thread id (*MYTHREAD*) as an index into an array of pointers to heap objects (upcr_local_heap). The selected heap object is used by the heap manager to implement local heap allocations for the current thread.

15.2.6 upc_free

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

The *upc_free* function frees the dynamically allocated shared storage pointed to by *ptr*. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the *upc_alloc*, *upc_global_alloc*, *upc_all_alloc*, or *upc_local_alloc*, function, or if the space has been de-allocated by a previous call, by any thread, to *upc_free*, the behavior is undefined.

```
void upc_free (shared void *ptr)
{
    const int thread = (int)upc_threadof (ptr);
    upcr_heap_p *heap_p;
    upcr_heap_node_p this_p;
    /* 'this_p' points to the heap object header
        that is located below the allocated storage area.  */
    this_p = (upc_heap_node_p)((shared [] char *)ptr - UPCR_HEAP_OVERHEAD);
    if (this_p->is_global)
      /* If a global allocation, use the global heap object.  */
      heap_p = (shared upc_heap_p *)&__upc_global_heap;
    else
      /* If a local allocation, use the global heap object.  */
      heap_p = (shared upc_heap_p *)&__upc_local_heap[thread];
    /* free the object using the selected global/local heap free list.  */
    upcr_heap_free (heap_p, this_p);
}
```

Above, either the global heap, or a local heap object is selected based upon the value of the *is_global* flag in the allocated heap object's header structure. The upcr_heap_free function will perform the de-allocation, returning the allocated storage to the specified heap's free list.

15.3 Lock Functions

The UPC lock functions use *MCS locks* as described in the Mellor-Crummey and Scott paper: *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. ACM Transaction on Computer Systems, February 1991 [mcs_locks].

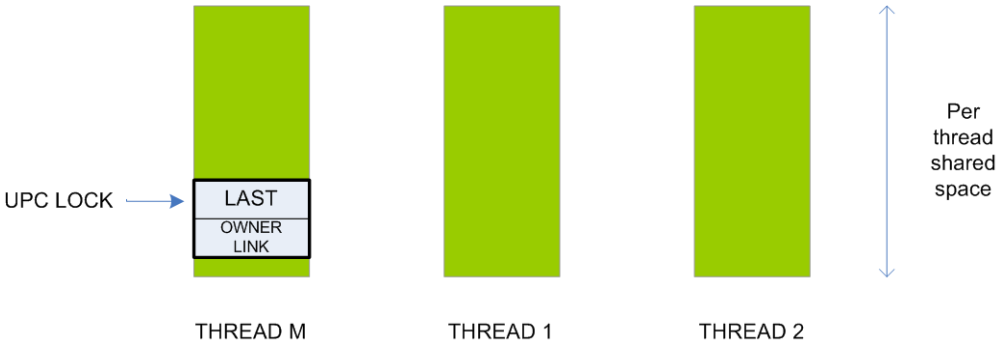
A lock is a simple data structure that lives in the shared memory space. A pointer is used to point to the last thread on the waiting list. A lock is available if this pointer is NULL. Portals atomic operations are used to determine:

- if the lock is available (atomic SWAP)
- if the lock can be released (atomic CSWAP)

The Portals implementation of UPC locks has the following characteristics:

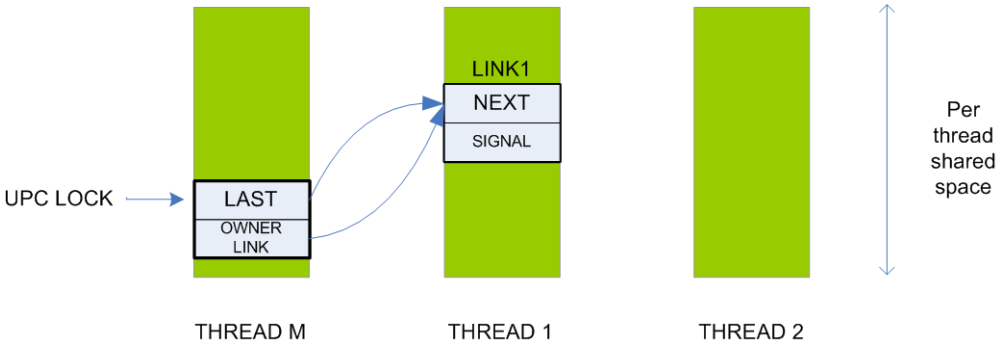
- The lock object has affinity to the thread that creates the lock. If the collective function, *upc_all_lock_alloc*, is called, then the lock object will have affinity to thread 0.
- A thread's lock waiting queue link object has affinity to the waiting thread.
- Portals atomic functions (SWAP and CSWAP) are used to guarantee fair access and FIFO ordering for all waiting threads.
- A special Portals Table Entry (PTE) is used to provide for signaling threads taken off the waiting list.

The following set of pictures demonstrate threads queuing while waiting on the lock:



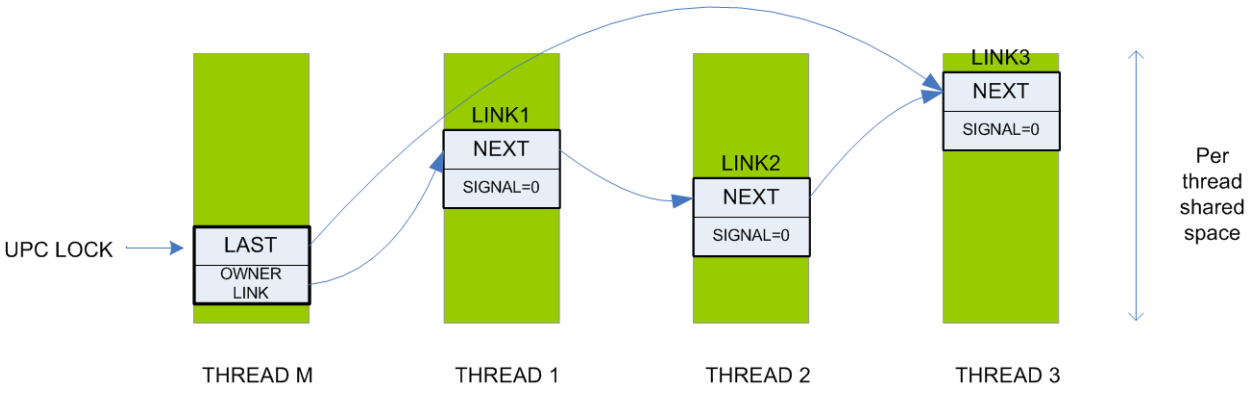
THREAD M CREATES A LOCK

1. Lock object is allocated in thread M's shared space
2. LAST/OWNER LINK fields are set to NULL indicating that the lock is free



THREAD 1 ACQUIRES THE LOCK

1. Thread 1 allocates its link structure in its shared space - LINK1. Both the NEXT and SIGNAL fields of LINK1 are initialized to NULL.
2. Thread 1 tries to acquire the lock by performing a Portals-4 SWAP atomic operation on the LAST field of the lock. A pointer to LINK1 is written into LAST; the current value of LAST is returned.
3. Since the lock is free a NULL is returned, Thread 1 is the owner of the lock. The OWNER LINK field of the lock is initialized with a pointer to LINK1.



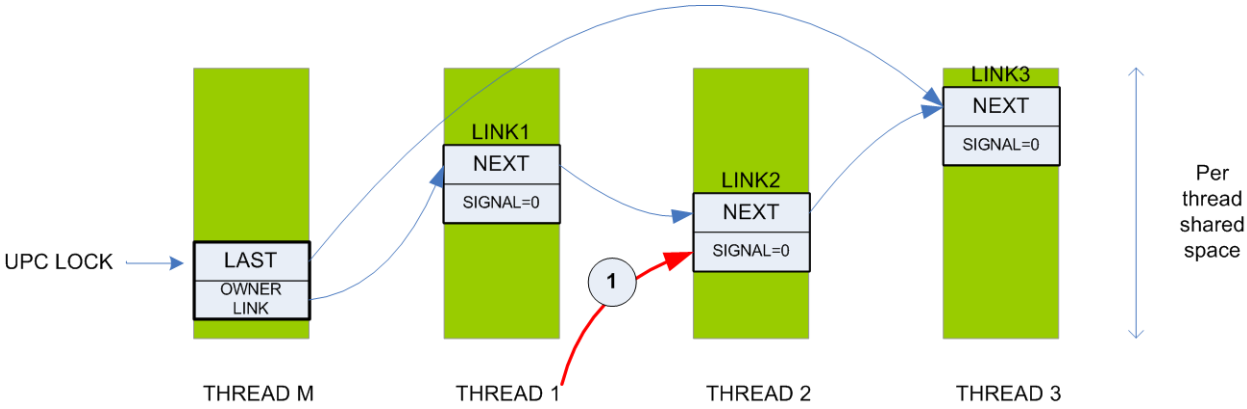
THREAD 2 ATTEMPTS TO ACQUIRE THE LOCK

1. Thread 2 allocates LINK2 from its own shared space.
2. A pointer to LINK1 is returned as the result of the swap operation.

3. Thread 2 inserts itself onto the waiting list by writing a pointer to LINK2 into the NEXT field of LINK1.
4. Thread 2 must wait for Thread 1 to pass ownership of the lock. Thread 2 waits for a value of 1 to be written into the SIGNAL field of LINK2.

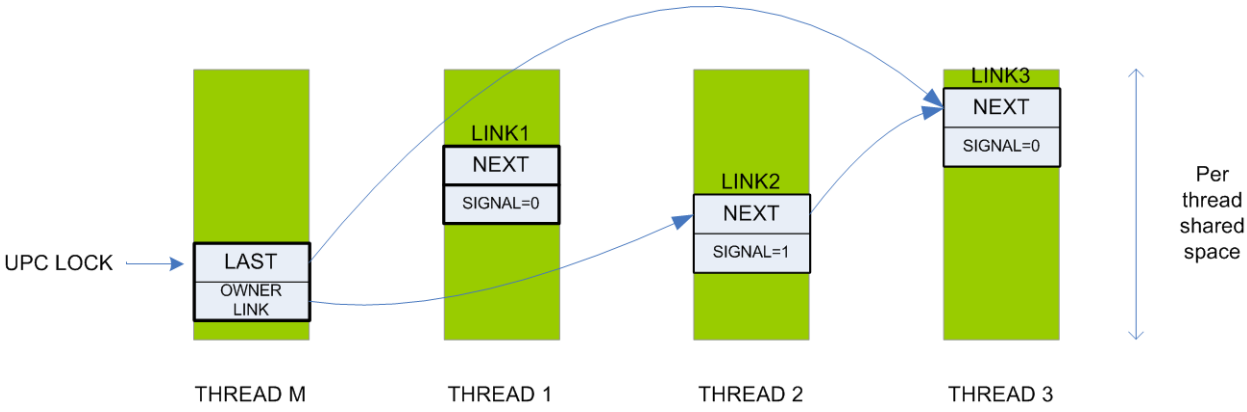
THREAD 3 ATTEMPTS TO ACQUIRE THE LOCK

1. A pointer to LINK3 is written into the NEXT field of LINK2.



THREAD 1 RELEASES THE LOCK

1. Thread 1 attempts to release the lock by performing a Portals compare and swap (CSWAP) atomic operation on the LAST field of the lock. If the value of the field points to LINK1 (there are no other threads waiting on the lock), a NULL is written into it and the lock is released.
2. A Compare and Swap operation returns a pointer to LINK3 and the lock wait list must be checked.
3. The NEXT field of LINK1 is checked for the first waiting thread. It is set to point to LINK2 (if not, then wait for it to be set).
4. Thread 1 writes a value of 1 (any value different then zero) into the SIGNAL field of LINK2.



THREAD 2 OWNS THE LOCK

1. Thread 2 receives a signal from Thread 1 and takes ownership of the lock.

15.3.1 UPC Runtime Lock Utility Functions

The UPCR lock utility functions are designed to atomically transfer data to/from a thread's shared memory space, as well as to pass ownership of the lock to the next thread on the waiting list. For this purpose, the thread's shared address space is accessible through a special Non-Matching Portals table entry, `GUPCR_PTL_PTE_LOCK`, in order to separate Portals counting events for writes to the lock data structure from events caused by regular UPC shared memory accesses.

15.3.1.1 upcr_lock_put

```
void
upcr_lock_put (shared void *dst, void *val, int size);
```

This function performs a Portals put operation on the lock's Portals table entry (GUPCR_PTL_PTE_LOCK). This separate Portals table entry is used to make it possible to count only Portals put operations on the *signal* or *next* words of a UPC lock wait list entry.

This function is used to *signal* the remote thread that: - ownership of the lock is passed to a remote thread if the remote thread is the next thread on the waiting list - a pointer to the calling thread's local control block has been appended to the lock's waiting list

15.3.1.2 upcr_lock_swap

```
/* Execute atomic fetch and store remote operation. Value
   "val" is written into remote location and the
    old location value is returned to the caller. */
void
upcr_lock_swap (shared void *dst, void *val, void *old, int size);
```

A Portals *swap atomic* operation is used when the acquiring thread must atomically determine if the lock is available. A pointer to the thread's local lock waiting list link is atomically written into the lock's *last* field, and the current value of the *last* field is returned. If NULL, the acquiring thread is the new owner, otherwise it must insert itself onto the waiting list.

15.3.1.3 upcr_lock_cswap

```
/* Execute an atomic compare and swap operation.  The value
   pointed to by 'val' is written into the remote location pointed by
  'dst' only if value in 'dst' is identical to 'cmp'. Return
   the value of TRUE indicates successful operation. */
int
upcr_lock_cswap (shared void *dst, void *cmp, void *val, int size);
```

A Portals compare and swap atomic operation is used during the lock release phase when the owner of the lock must atomically determine if there are threads waiting on the lock. This is accomplished by using the Portals CSWAP atomic operation, where a NULL pointer is written into the lock *last* field only if the same field contains the pointer to the owner's local lock link structure.

15.3.2 upc_global_lock_alloc

```
upc_lock_t *upc_global_lock_alloc ();
```

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

```
upc_lock_t *
upc_global_lock_alloc ()
{
  shared upcr_lock_t *lock;
  /* Allocate space for the lock from shared memory with
     affinity to the calling thread.  */
  lock = upc_alloc (sizeof (lock));
  lock->link = NULL;
  lock->owner_link = NULL;
  return (upc_lock_t *)lock;
}
```

15.3.3 upc_all_lock_alloc

```
upc_lock_t *upc_all_lock_alloc ();
```

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

```
upc_lock_t *
upc_all_lock_alloc ()
{
 shared upcr_lock_t *lock;
 /* Allocate space for the lock from the shared memory of
    thread 0 and broadcast its address. */
 if (MYTHREAD == 0)
   {
     lock = upc_alloc (sizeof (lock));
     lock->link = NULL;
     lock->owner_link = NULL;
     upcr_broadcast_put (&lock, sizeof (lock));
   }
 else
   {
     upcr_broadcast_get (&lock, sizeof (lock));
   }
 return (upc_lock_t *)lock;
}
```

Above, upcr_broadcast_put is called by thread 0 to propagate the newly allocated lock pointer to all other threads. These other threads call upcr_broadcast_get to retrieve this broadcasted value.

15.3.4 upc_lock

```
void upc_lock (upc_lock_t *ptr);
```

The *upc_lock* function sets the state of the lock pointed to by *ptr* to *locked*. If the lock is already in a *locked* state due to the calling thread setting it to the *locked* state, the result is undefined. If the lock is already in a *locked* state, then the calling thread waits for some other thread to set the state to *unlocked*. Once the lock is in the state *unlocked*, a single calling thread sets the state to *locked* and the function returns. A null strict access is implied after a call to *upc_lock*.

```
void
upc_lock (upc_lock_t *ptr)
{
  upcr_lock_link_t *link;
  shared void *old_link;
  upcr_lock_t *lock = (upcr_lock_t *) ptr;
  /* Allocate space for the lock waiting queue link.
     It will have affinity to the calling thread. */
  link = upc_alloc (sizeof (link));
  /* Atomically set the lock value to point to the
     calling thread's link queue object and
     return the previous value of the lock link. */
  upcr_lock_swap (lock->last, &link->next, &old_link,
                  sizeof (link->next));
  if (old_link != NULL)
    {
      /* We have to wait. Clear the ownership signal field
         and insert our pointer into the predecessor's link. */
      link->signal = UPCR_LOCK_NO_SIGNAL;
      upcr_lock_put (old_link, &link->next, sizeof(link->next));
      /* At this point the thread has to wait until the lock is
```

```
          is released. Process counting events one by one until
          the value of the signal word changes. */
       do
         {
           PtlCTWait (upcr_lock_le_ct, ++current_lock_ct, ...);
         } while (link->signal == UPCR_LOCK_NO_SIGNAL);
     }
  lock->owner_link = link;
}
```

15.3.5 upc_lock_attempt

```
int upc_lock_attempt (upc_lock_t *ptr);
```

The *upc_lock_attempt* function attempts to set the state of the lock pointed to by *ptr* to *locked*. If the lock is already in the *locked* state due to the calling thread setting it to the *locked* state, the result is undefined. If the lock is already in the *locked* state, the function returns 0. If the lock is in the state *unlocked*, a single calling thread sets the state to *locked* and the function returns 1. A null strict access is implied after a call to *upc_lock_attempt* that returns 1.

```
int
upc_lock_attempt (upc_lock_t *ptr)
{
  upcr_lock_link_t *link;
  shared void *old_link;
  upcr_lock_t *lock = (upcr_lock_t *) lock;
  /* Allocate space for the lock waiting queue with affinity
     to the calling thread. */
  link = upc_alloc (sizeof (link));
  /* Atomically set the lock value to the link entry and
     return the previous value of the lock ONLY if the value
     of the lock is already NULL. */
  compare_ok = upcr_lock_cswap (&lock->last, NULL, &link, sizeof(link));
  if (!compare_ok)
    {
      upc_free (link);
      return FALSE;
    }
  lock->owner_link = link;
  return TRUE;
}
```

15.3.6 upc_unlock

```
void upc_unlock (upc_lock_t *ptr);
```

The *upc_unlock* function sets the state of the lock pointed to by *ptr* to *unlocked*. Unless the lock is in *locked* state and the calling thread is the locking thread, the result is undefined. A null strict access is implied before a call to *upc_unlock*.

```
void
upc_unlock (upc_lock_t *ptr)
{
  upcr_lock_t *lock = (upcr_lock_t *)ptr;
  upcr_lock_link_t *link = lock->owner_link;
  int signal = UPCR_LOCK_SIGNAL;
  bool compare_ok;

  /* Try to release the lock: write NULL into lock->last
```

```
     if it contains a pointer to our own link block. If it fails then
     some other thread is on the waiting list. */
  lock->owner_link = NULL;
  compare_ok = upcr_lock_cswap (&lock->last, &link, NULL, sizeof(link));
  if (!compare_ok)
    {
      /* Pass ownership to the next waiting thread. Process
         counting events one by one until 'next' link is set.  */
      while (link->next == NULL)
        {
          PtlCTWait (upcr_lock_le_ct, ++current_lock_ct, ...);
        }
      /* Signal the waiting thread that it now owns the lock.  */
      upcr_lock_put (&signal, sizeof (int), link->next->signal);
    }
  upc_free (link);
}
```

15.3.7 upc_lock_free

```
void upc_lock_free(upc_lock_t *ptr);
```

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 de-allocated by a previous call to *upc_lock_free* the behavior is undefined.

```
void
upc_lock_free (upc_lock_t *ptr)
{
  upc_free (ptr);
}
```

15.4 Miscellaneous Functions

15.4.1 upc_global_exit

```
void upc_global_exit (int status);
```

upc_global_exit pushes all I/O, releases all storage, and terminates the execution for all active threads.

The *upc_global_exit* function presents some implementation difficulties because in order to implement this function according to the specification, all threads are required to run clean up code before exiting.

There are several design alternatives:

- The calling thread directs the job manager to send a `SIGTERM` signal to all other threads (processes) in the application. The UPC runtime catches the signal, performs all necessary clean up actions and then exits. The job manager ensures that the initial process exits with the indicated exit code.
- A PTE is reserved and an LE is set up in each thread that will participate in a triggered put operation which will broadcast the exit value to a known location. The runtime installs a timer signal handler that periodically checks for changes in this known global exit code communication area. If a change is detected, the clean up handler is executed and the thread (process) exits.
- Active Messages, if implemented, can be used to ensure that the clean up handler is invoked asynchronously to the compute thread.

We have adopted a design that uses elements from the design alternatives described above. This design is described in the following paragraphs.

Each UPC thread (process) creates a helper (shutdown) pthread with the sole purpose of waiting for receipt of a remote request to shutdown, as a result of the other thread issuing a call to upc_global_exit.

This pthread uses a special PTE/LE (GUPCR_PTL_PTE_SHUTDOWN) to receive a global exit code from another UPC thread. A simple PtlPut of the exit code issued to the shutdown PTE on some other UPC thread triggers exit of the receiving thread.

The following steps are taken to initialize, wait, and signal the UPC global exit:

1. Each thread initializes a PTE/LE to receive an exit code that was passed in as the argument to upc_global_exit().
2. Each thread creates a helper pthread - gupcr_shutdown_pthread() that waits on the shutdown LE's counting event (one count only).
    a. The main UPC thread installs a signal handler for the shutdown signal (e.g. SIGUSR2) that is used by the shutdown pthread to signal a need for global exit.
    b. Remote shutdown takes the following steps:
        i. A UPC thread executing a call to upc_global_exit() sends the exit code to all other UPC threads by using the shutdown PTE.
        ii. The pthread associated with each UPC thread receives the exit code and returns from the counting event Portals wait call.
        iii. The receiving pthread sends shutdown signal to main UPC thread and then calls pthread_exit().
        iv. The main UPC thread receives the signal, which invokes the signal handler.
        v. The signal handler waits for the shutdown pthread to exit and then calls exit() with the code received from the thread that sent the shutdown request.

Note
The gupcr_exit() function is registered with atexit() and will be executed when exit() is called.

Note
Upon regular exit, the main UPC thread disables the shutdown signal, and terminates the shutdown pthread by writing a dummy value using its own shutdown PTE.

15.5 Pointer-to-shared Manipulation Functions

Although these routines are specified as part of the UPC language specification, they likely will not need to be re-implemented when the runtime is modified for operation with Portals. They are listed here for completeness.

15.5.1 upc_threadof

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

The *upc_threadof* function returns the index of the thread that has affinity to the shared object pointed to by *ptr*. If *ptr* is a null pointer-to-shared, the function returns 0.

15.5.2 upc_phaseof

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

The *upc_phaseof* function returns the phase component of the pointer-to-shared argument. If *ptr* is a null pointer-to-shared, the function returns 0.

15.5.3 upc_resetphase

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

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

15.5.4 upc_addrfield

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

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

15.5.5 upc_affinitysize

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

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*. 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. In the case of a statically allocated shared object with declaration: shared [b] t d[s]; the *totalsize* argument shall be *s * sizeof (t)* and the *nbytes* argument shall be *b * sizeof (t)*. If the block size is indefinite, *nbytes* shall be 0. *threadid* shall be a value in *0..(THREADS-1)*.

Chapter 16

UPC Collectives Library

UPC collective operations allow UPC threads to collectively operate on multiple data streams and let them gather, scatter, and exchange data.

Although there are an extensive number of *collectives* defined by the UPC (Chapter 7.3 of the UPC Language Specification v1.2 [upc_lang_spec]), many of the reduce functions vary only in the data types that they operate on. For that class of functions, a unique two-letter suffix is appended to the function name, indicating the data type of the arguments.

The following UPC collective functions are defined by the UPC Language Specification:

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

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

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

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

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

void
```

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

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

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

A collective function *upc_all_sort()* is not part of the UPC Language Specification v1.2. However, it is part of the UPC Collective Operations Specifications V1.0 ([collectives_spec]), and it is also part of the current GCC/UPC run-time library implementation.

```
void
upc_all_sort (
         shared void *A,
         size_t elem_size,
         size_t nelems,
         size_t blk_size,
         int (*func) (shared void *, shared void *),
         upc_flag_t sync_mode);
```

The following functions are part of the GCC/UPC collectives implementation:

```
void
upc_coll_init ();

void
upc_coll_err (
        shared void *dst,
        shared const void *src,
        shared const int *perm,
        size_t nbytes,
        upc_flag_t sync_mode,
        size_t blk_size,
        size_t nelems,
        upc_op_t op,
        upc_flag_t upc_coll_op);
```

The UPC language specification also defines an integral data type *upc_flag_t* (defined in <upc.h>) which is used to control the data synchronization semantics of certain collective UPC library functions. It can take the following values:

```
#define UPC_IN_NOSYNC           1
#define UPC_IN_MYSYNC           2
#define UPC_IN_ALLSYNC          0
```

```
#define UPC_OUT_NOSYNC          4
#define UPC_OUT_MYSYNC          8
#define UPC_OUT_ALLSYNC         0
```

The upc_flag_t *IN* values regulate access to shared data upon entry to a collective function, while *OUT* values regulate access upon exit from a collective function.

The following table describes all possible values of *upc_flag_t*:

Table 16.1: Memory Semantics of Collective Library Functions

Value	**Description**
UPC_IN_NOSYNC	Function may begin to read or write data when the first thread has entered the collective function call
UPC_IN_MYSYNC	Function may begin to read or write only data which has affinity to threads that have entered the collective function call
UPC_IN_ALLSYNC	Function may begin to read or write data only after all threads have entered the function call
UPC_OUT_NOSYNC	Function may read and write data until the last thread has returned from the collective function call
UPC_OUT_MYSYNC	Function call may return in a thread only after all reads and writes of data with affinity to the thread are complete
UPC_OUT_ALLSYNC	Function call may return only after all reads and writes of data are complete

16.1 UPC Collectives - Supported Operations

The UPC collective functions define functions that are collectively executed by all threads. For *all-reduce* collective functions the following collective operations are defined:

```
#define UPC_ADD                 0
#define UPC_MULT                1
#define UPC_AND                 2
#define UPC_OR                  3
#define UPC_XOR                 4
#define UPC_LOGAND              5
#define UPC_LOGOR               6
#define UPC_MIN                 7
#define UPC_MAX                 8
#define UPC_FUNC                9
#define UPC_NONCOMM_FUNC        10
```

The following table lists the data types supported by the UPC collective functions. The corresponding MPI data types are also listed.

Table 16.2: Data Types Supported by UPC Collectives

Identifier	**Type**	**Size**
C	signed char	8
UC	unsigned char	8
S	signed short	16
US	unsigned short	16
I	signed int	32
UI	unsigned int	32
L	signed long	64
UL	unsigned long	64
F	float	32
D	double	64
LD	long double	128

16.2 UPC Collectives - Correspondence to MPI and Portals

The following table describes the correspondence between UPC and MPI collective operations. Also listed are Portals atomic operations that provide support for the corresponding collectives library function.

Table 16.3: UPC and MPI All-Reduce Operations Comparison

UPC Operation	**MPI Operation**	**Portals Atomic Operation**
UPC_ADD	MPI_SUM	PTL_SUM
UPC_MULT	MPI_PROD	PTL_PROD
UPC_AND	MPI_BAND	PTL_BAND
UPC_OR	MPI_BOR	PTL_BOR
UPC_XOR	MPI_BXOR	PTL_BXOR
UPC_LOGAND	MPI_LAND	PTL_LAND
UPC_LOGOR	MPI_LAND	PTL_LOR
UPC_MIN	MPI_MIN	PTL_MIN
UPC_MAX	MPI_MAX	PTL_MAX
UPC_FUNC	N/A	
UPC_NONCOMM_FUNC	N/A	

16.3 UPC Collectives - Correspondence to MPI and Portals Data Types

The following table describes the correspondence between UPC, MPI, and Portals supported atomic operation data types.

Table 16.4: UPC and MPI All-Reduce Data Type Comparison

UPC Data Type	**MPI Data Type**	**Portals Atomic Data Type**
C	MPI_CHAR	PTL_CHAR
UC	MPI_UNSIGNED_CHAR	PTL_UCHAR
S	MPI_SHORT	PTL_SHORT
US	MPI_UNSIGNED_SHORT	PTL_USHORT
I	MPI_INT	PTL_INT
UI	MPI_UNSIGNED_INT	PTL_UINT
L	MPI_LONG	PTL_LONG
UL	MPI_UNSIGNED_LONG	PTL_ULONG
F	MPI_FLOAT	PTL_FLOAT
D	MPI_DOUBLE	PTL_DOUBLE
LD	MPI_LONG_DOUBLE	PTL_LONG_DOUBLE

16.4 upc_all_broadcast

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

For the purposes of implementing the *upc_all_broadcast*, all UPC threads in a given job are organized as a tree. This is similar to the tree used for implementation of the UPC barrier, but the notable difference is that the thread with affinity to the data pointed

by the source pointer to shared is in the root of the tree. Data pointed to by the source pointer-to-shared is passed down the tree, with each thread being responsible of passing data to its children.

A special Portals PTE for collectives is used to *signal* a thread that parent wrote data into the thread's local memory.

```
void
upc_all_broadcast (
         shared void *dst,
         shared const void *src,
         size_t nbytes,
         upc_flag_t sync_mode)
{
  /* Optional barrier on enter. */

  if (ROOT THREAD)
    {
      /* Copy source data into this thread's portion
         of the destination area. Use local memory copy
         as source and destination have affinity to this
         thread. */
    }
  else
    {
      /* Wait for parent to write data into destination
         area with affinity to this thread. */
    }
  if (! LEAF THREAD)
    {
      /* Write received data to all thread's children and
         wait for transfer completion. */
    }

  /* Optional barrier on exit. */
}
```

16.5 upc_all_scatter

The *upc_all_scatter* function copies the *I-th* block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the *I-th* thread. MPI's implementation of MPI_Scatter function might provide an example for optimal UPC implementation.

A portable UPC implementation is provided in the reference implementation of the UPC collective function library. It is listed below.

```
void
upc_all_scatter (
         shared void *dst,
         shared const void *src,
         size_t nbytes,
         upc_flag_t sync_mode)
{
  upc_memcpy ((shared char *) dst + MYTHREAD,
              (shared char *) src + nbytes * MYTHREAD * THREADS, nbytes);
}
```

16.6 upc_all_gather

The *upc_all_gather* function is the opposite operation from *upc_all_scatter*. Smaller blocks of memory are combined into larger blocks. MPI's implementation of the *MPI_Gather* function should provide guidance for an efficient UPC implementation.

A portable UPC implementation is provided in the reference implementation of the UPC collective function library. It is listed below.

```
void
upc_all_gather (
         shared void *dst,
         shared const void *src,
         size_t nbytes,
         upc_flag_t sync_mode)
{
  if ((int) upc_threadof ((shared void *) dst) == MYTHREAD)
    {
      for (i = 0; i < THREADS; ++i)
        {
          upc_memcpy ((shared char *) dst + nbytes * i * THREADS,
                      (shared char *) src + i, nbytes);
        }
    }
  upc_barrier;
}
```

16.7 upc_all_gather_all

The *upc_all_gather_all* function copies a block of memory from one shared memory area with affinity to the *I-th* thread to the *I-th* block of shared memory on each thread. MPI's implementation of the *MPI_Gather* function should provide guidance on an efficient UPC implementation.

A portable UPC implementation is provided in the reference implementation of the UPC collective function library. It is listed below.

```
void
upc_all_gather_all (
         shared void *dst,
         shared const void *src,
         size_t nbytes,
         upc_flag_t sync_mode)
{
  for (i = 0; i < THREADS; i++)
    {
      upc_memcpy ((shared char *) dst + i * nbytes * THREADS + MYTHREAD,
                  (shared char *) src + i, nbytes);
    }
}
```

16.8 upc_all_exchange

The *upc_all_exchange* function copies the *I-th* block of memory from a shared memory area that has affinity to thread *J* to the *J-th* block of a shared memory area that has affinity to thread *I*.

A portable UPC implementation is provided in the reference implementation of the UPC collective function library. It is listed below.

```
void
upc_all_exchange (
         shared void *dst,
         shared const void *src,
                 size_t nbytes,
```

```
                  upc_flag_t sync_mode)
{
  for (i = 0; i < THREADS; i++)
    {
      upc_memcpy ((shared char *) dst + i * nbytes * THREADS + MYTHREAD,
                  (shared char *) src + i, nbytes);
    }
}
```

16.9 upc_all_permute

The *upc_all_permute* function copies a block of memory from a shared memory area that has affinity to the *I-th* thread to a block of a shared memory that has affinity to thread *perm[i]*.

A portable UPC implementation is provided in the reference implementation of the UPC collective function library. It is listed below.

```
void
upc_all_permute (
         shared void *dst,
         shared const void *src,
         shared const int *perm,
         size_t nbytes,
         upc_flag_t sync_mode);
{
  upc_memcpy ((shared char *) dst + perm[MYTHREAD],
              (shared char *) src + MYTHREAD, nbytes);
}
```

16.10 upc_all_reduceT

The algorithm used for *all reduce* UPC collectives is similar to the one already described in the implementation of UPC barriers. UPC collectives over the Portals takes advantage of Portals atomic and triggered atomic functions.

All of the data types supported by the UPC collectives are supported by the Portals. However, some of the operations are not supported and special handling must be implemented.

- For operations and data types that have equivalent in Portals, it should be possible to use Portals atomic or triggered atomic operations when propagating results both up and down the thread tree hierarchy. Portals atomic operations should be applicable in this situation, because no additional computation is needed at each node in tree, other than the computations performed by Portals.
- UPC collectives operations UPC_FUNC and UPC_NONCOMM_FUNC have no equivalent in Portals atomic operations. They must be performed by each thread that has children in the tree.
- UPC collectives operations UPC_LOGAND and UPC_LOGOR on the floating point types are not supported by the Portals implementation. They must be implemented the same way as the UPC_FUNC operation.

Also, the *upc_all_reduceT* implementation over Portals must take into the account that not all of the threads participate in the reduce operation. Only threads starting from the source pointer to shared and up to the number of specified elements are participating.

The pseudo code in the following example assumes that all UPC collectives operations are supported by the Portals.

```
void
upc_all_reduceT (
         shared void *dst,
         shared const void *src,
         upc_op_t op,
         size_t nelems,
         size_t blk_size,
         upc_flag_t sync_mode);
{
  /* Optional barrier on enter. */

  n_locals = ...     /* Calculate number of elements local to this thread based
                        on the source pointer to shared, number elements to
                        reduce, and the block size. */
  local_result = ... /* Reduce all elements that are local to this thread. */

  if (n_locals)
    {
      /* This thread has to participate in the reduce operation among
         threads. */

      /* Create thread tree structure for threads participating in the
         reduce function. Provide a hint for the root thread (destination
         thread might not be participating in the reduce). */

      /* Store this thread's local_result into the area that is going
         to be used for Portals atomic functions. */

      if (! LEAF_THREAD)
        {
          /* Send signal to all children that parent is ready for the reduce
             operation. This is accomplished by using a Portals put operation
             into the children's PTEs. */

          /* Wait for children to perform reduce. */

          if (!ROOT THREAD)
            {
              /* Wait for parent to be ready. */

              /* Perform Portals atomic operation in to the parent's PTE. */

              /* Wait for operation completion. */
            }

        }
       else /* LEAF THREAD */
        {
          /* Wait for parent to be ready. */

          /* Perform Portals atomic operation in to the parent's PTE. */

          /* Wait for operation completion. */
        }

       if (ROOT THREAD)
        {
          /* Make reduction result available for the caller. */
        }
    }
```

```
  /* Optional barrier on exit. */
}
```

16.11 upc_all_prefix_reduceT

MPI's implementation of the *MPI_reduce* function should provide on an efficient implementation of this collectives library function.

16.12 upc_all_sort

The *upc_all_sort* function takes a shared array with elements of specific size and sorts them in place in ascending order using the specified function to compare elements.

The UPC collective reference implementation provides a basic implementation. It is too lengthy to be listed here.

16.13 upc_coll_init

```
void upc_coll_init ();
```

The *upc_coll_init* function must be called by all thread before calling any of the functions from the collective library.

This function will be used to initialize necessary storage space for collectives implementation on each thread:

- signal - Area used for parent to inform children that they are ready for the Portals atomic operations.
- value - Value for the Portals atomic operations.

16.14 upc_coll_err

```
void upc_coll_err ();
```

The *upc_coll_err* function checks that the arguments passed to a UPC collectives library function are single-valued. It is checks that all threads have called the same UPC Collectives library function.

The UPC collectives reference implementation provides an implementation of this function. It is too lengthy to be listed here.

Chapter 17

Active Messages

One well known UPC runtime uses Active Messages (see [gasnet_spec]) extensively to implement various UPC constructs, and as a fall back when the underlying network layer does not directly provide support for one-sided communication.

For the UPC runtime design described in this document, there is no demonstrated need for Active Messages, and therefore no design for Active Messages is presented here.

If Active Messages were implemented using Portals, it is likely that event queues, LE's with remotely managed offsets, and events signalled by *PTL_EVENT_PUT* will provide most of the required support for Active Messages.

Chapter 18

Generalized Non-blocking Get and Put Operations

The *get* and *put* operations in this section have *nh* in their name to indicate their more general non-blocking nature. These functions use *handles* to synchronize and manage *get* and *put* operations. The basic *get* and *put* operations described earlier are also non-blocking, but use a simpler event-counting interface to synchronize event completions.

These more general non-blocking *get* and *put* operations described in this section refer to an opaque type: *upcr_gmem_handle_t*. Although this type is not defined to be a pointer, its value is constrained to fit into a pointer sized container.

The UPC runtime design described in this document does not demonstrate the need for the more general form of non-blocking *get* and *put* functions. Therefore no design is presented. Their call interface specifications are provided for completeness, in the event that there is a decision to implement them at a future time.

If these more general non-blocking *get* and *put* functions were implemented using Portals, it is likely that event queues associated with local *get* and *put* memory descriptors will be used. Other elements of the design already presented for basic gets/puts will likely be applicable, except that the "bounce buffer" and event synchronization logic will be more complex.

18.1 upcr_gmem_get_nh

```
upcr_gmem_handle_t
upcr_gmem_get_nh (void *dest, upcr_gmem_addr_t src, size_t n);
```

Copy the shared data at *src* to the local memory area at *dest*. The number of bytes to copy is given by *n*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory area designated by *src* is located in the global memory region that has affinity to the calling thread, this operation will be implemented as a local memory-to-memory copy. An object of opaque type *upcr_gmem_handle_t* is returned. This "handle" can be used to wait for completion of this 'get operation.

18.2 upcr_gmem_put_nh

```
upcr_gmem_handle_t
upcr_gmem_put_nh (upcr_gmem_addr_t dest, void *src, size_t n);
```

Copy the local memory area at *src* to the global memory area at *dest*. The number of bytes to copy is given by *n*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory area designated by *dest* is located in the global memory region that has affinity to the calling thread, this operation may be implemented as a local memory-to-memory copy. If the byte count is less than or equal to a configuration-defined value, *UPCR_GMEM_MAX_SAFE_PUT_SIZE*, the memory area at *src* can be safely re-used without the need to wait for completion of this operation. An object of opaque type *upcr_gmem_handle_t* is returned. This "handle" can be used to wait for completion of this put operation.

18.3 upcr_gmem_copy_nh

```
upcr_gmem_handle_t
upcr_gmem_copy_nh (upcr_gmem_addr_t dest, upcr_gmem_addr_t src, size_t n);
```

Copy the global memory area at *src* to the global memory area at *dest*. The number of bytes to copy is given by *n*. There is no address alignment restriction for either the source or destination. The memory areas should not overlap. If the shared memory areas designated by *dest* and *src* are located in the global memory region that has affinity to the calling thread, this operation may be implemented as a local memory-to-memory copy. If the byte count is less than or equal to a configuration-defined value, *UPCR_GMEM_MAX_SAFE_PUT_SIZE*, the memory area at *src* can be safely re-used without the need to wait for completion of this operation. An object of opaque type *upcr_gmem_handle_t* is returned. This "handle" can be used to wait for completion of this copy operation.

18.4 upcr_gmem_set_nh

```
upcr_gmem_handle_t
upcr_gmem_set_nh (upcr_gmem_addr_t dest, int c, size_t n);
```

Fill the global memory area at *dest* with *n* bytes with the value given by the *c* argument. There is no address alignment restriction for the destination. If the shared memory area designated by *dest* is located in the global memory region that has affinity to the calling thread, this operation may be implemented as a local memory access. An object of opaque type *upcr_gmem_handle_t* is returned. This "handle" can be used to wait for completion of this *set* operation.

18.5 upcr_gmem_wait_all_nh

```
void upcr_gmem_wait_all_nh (upcr_gmem_handle_t *which, size_t n);
```

Wait for all non-blocking get or put operations identified by the list of handles given by the *which* argument. The number of entries in this list of handles is given by the value of the *n* argument.

18.6 upcr_gmem_wait_any_nh

```
void upcr_gmem_wait_any_nh (upcr_gmem_handle_t *which, size_t n);
```

Wait for the completion of any non-blocking get or put operation identified by the list of handles given by the *which* argument. The number of entries in this list of handles is given by the value of the *n* argument.

Chapter 19

Conclusions

This report describes the design of a UPC runtime, which utilizes the Portals 4 API when transmitting data and control messages between the local node and other nodes in a computing cluster.

Key design elements are summarized below.

1. Non-Matching PTEs allow for efficient UPC shared data addressing
2. Counting events allow for fast non-blocking UPC get/put operations
3. Triggered operations allow for efficient implementation of UPC barriers
4. Mapping a thread's shared space through a separate PTE is used for "signaling". This signaling capability provides for an efficient implementation of UPC locks
5. Portals atomic operations provide a good basis for implementing UPC collectives

Chapter 20

References

20.1 Bibliography

[1] [collectives_spec] Steve Seidel et al. UPC Collective Operations Specifications v1.0. CCS-TR-02-159, March, 2002.

[2] [gasnet_spec] Dan Bonachea. GASNet Specification. Version 1.4, June 28th, 2003.

[3] [mcs_locks] Mellor-Crummey and Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM Transaction on Computer Systems, February 1991.

[4] [portals4] Underwood et al. Portals 4 Specification. Sandia Technical Report. January, 2011.

[5] [triggered_ops] Keith Underwood et al. Enabling Flexible Collective Communication Offload with Triggered Operations. January, 20007.

[6] [upc_lang_spec] William Carlson et al. UPC Language Specifications (V1.2). May 31, 2005.

[7] [upc_smp_impl] Bonachea et al. Implementing the UPC memory consistency model for shared-memory architectures. http://upc.gwu.edu/~upc/upcworkshop04/bonachea-consis-0904.pdf.