IBM XL Unified Parallel C for AIX, V11.0 (Technology Preview)

IBM XL Unified Parallel C User's Guide

Version 11.0

IBM XL Unified Parallel C for AIX, V11.0 (Technology Preview)

IBM XL Unified Parallel C User's Guide

Version 11.0

Note

Before using this information and the product it supports, read the information in “Notices” on page 97.

© Copyright International Business Machines Corporation 2010.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Parallel programming and Unified Parallel C 1
Parallel programming 1
Partitioned global address space programming model 1
Unified Parallel C introduction 2

Chapter 2. Unified Parallel C programming model 3
Distributed shared memory programming 3
Data affinity and data distribution 3
Memory consistency 5
Synchronization mechanism 6

Chapter 3. Using the XL Unified Parallel C compiler 9
Compiler options 9
New compiler options 9
Modified compiler options 11
Unsupported compiler options 12
Compiler commands 13
Invoking the compiler 13
Compiling and running an example program . . . 15

Chapter 4. Unified Parallel C language 17
Predefined identifiers 17
Unary operators 17
The address operator & 17
The sizeof operator 18
The upc_blocksizeof operator 18
The upc_elemsizeof operator 19
The upc_localsizeof operator. 19
Data and pointers 19
Shared and private data 20
Blocking of shared arrays. 20
Shared and private pointers 22
Pointer-to-shared arithmetic 26
Cast and assignment expressions 32
Declarations 39
Type qualifiers 39
Declarators 41
Statements and blocks 42
Synchronization statements 42
Iteration statements. 46
Predefined macros and directives 47
Unified Parallel C directives 47
Predefined macros 48

Chapter 5. Unified Parallel C library functions. 49
Utility functions 49
Program termination 49
Dynamic memory allocation 50
Pointer-to-shared manipulation 56
Serialization 62
Memory transfer. 70
Collective functions. 75
Synchronization options 75
Relocalization functions 76
Computational functions 83

Chapter 6. Compiler optimization . . . 89
Shared object access optimizations. 89
Shared object access privatization 89
Shared object access coalescing 90
Shared object remote updating 91
Array idiom recognition 92
Parallel loop optimizations 93
Loop reshaping 93
Loop versioning 95

Notices 97
Trademarks and service marks 99

Index 101

© Copyright IBM Corp. 2010

Chapter 1. Parallel programming and Unified Parallel C

Unified Parallel C is a parallel programming language that follows the partitioned global address space (PGAS) programming model. To support parallel programming, Unified Parallel C adds parallelism to the concepts and features of the C language. The following sections give a general description of parallel programming, of the PGAS programming model, and an introduction to the Unified Parallel C programming language.

Parallel programming

Parallel programming is a computer programming method that uses simultaneous execution of instructions to achieve better performance than serial programming. You can often obtain faster results by dividing a large task into independent subtasks that can be performed concurrently. This programming method is often used to perform large and complex tasks efficiently.

Based on the underlying memory architectures, the parallel programming models can be classified as follows:

The shared memory programming model
: Multiple threads or processes work concurrently, and share the same memory resources. The entire shared memory space can be directly accessed by any thread or process.

The distributed memory programming model
: This programming model consists of multiple independent processing nodes with local memory modules. Each process executing the program has direct access to its local portion of the memory. A process that requires access to memory located on a different node can do so by issuing a communication call to the process running on that remote node.

The distributed shared memory programming model
: The global memory space is divided into shared memory and private memory spaces. Each thread or process can directly access the entire shared memory space in addition to its own private memory.

Partitioned global address space programming model

The partitioned global address (PGAS) programming model is an explicitly parallel programming model that divides the global shared address space into a number of logical partitions. Similar to the shared memory programming model, each thread can address the entire shared memory space. In addition, a thread has a logical association with the portion of the global shared address space that is physically located on the computational node where the thread is running.

The PGAS programming model is designed to combine the advantages of the shared memory programming model and the message passing programming model. In the message passing programming model, each task has direct access to only its local memory. To access share data, tasks communicate with each other by sending and receiving messages. Unified Parallel C introduces the concept affinity which refers to the physical association between shared memory and a particular thread. The PGAS programming model facilitates data locality exploitation for

© Copyright IBM Corp. 2010

performance improvement like in the message passing programming model. In addition, the PGAS programming model uses one-sided communication to reduce the cost of inter-thread communication.

Related reference

"Data affinity and data distribution" on page 3

Unified Parallel C introduction

Unified Parallel C is an explicitly parallel extension of C based on the PGAS programming model. It preserves the efficiency of the C language and supports effective programming on various computer architectures. By using Unified Parallel C, high performance scientific applications have access to the underlying hardware architecture and can efficiently minimize the time required to transfer shared data between threads.

Unified Parallel C has the following features:

- Explicitly parallel execution model. The execution model used by Unified Parallel C is called Single Program Multiple Data (SPMD). All threads in a Unified Parallel C program execute the same program concurrently. Synchronization between threads is explicitly controlled by the user.
- Separate shared and private address spaces. Unified Parallel C threads can access their private memory space and the entire global shared space. The global shared memory space is partitioned and each thread has a logical association with its local portion of shared memory.
- Synchronization primitives. Unified Parallel C makes no implicit assumptions about the interaction between threads. Therefore, it is the user's responsibility to control thread interaction explicitly with the synchronization primitives: barriers, locks, and fences.
- Memory consistency. Unified Parallel C supports two memory consistency models: `strict` and `relaxed`. Strict shared data accesses are implicitly synchronized while relaxed shared memory accesses are not. By default, every shared data access follows the relaxed memory consistency model.

The IBM® XL Unified Parallel C compiler is a conforming implementation of the latest XL Unified Parallel C language specification (version 1.2), supporting IBM System p® systems running the AIX®, or Linux® operating system.

In addition to extensive syntactic and semantics checks, the XL Unified Parallel C compiler incorporates the following advanced optimizations that are developed and tailored to reduce the communication cost of Unified Parallel C programs:

- Shared-object access privatization
- Shared-object access coalescing
- `upc_forall` loop optimizations
- Remote shared-object updates
- Unified Parallel C parallel loop optimizations

Chapter 2. Unified Parallel C programming model

This section provides a general overview of the distributed shared memory programming model used by Unified Parallel C.

Distributed shared memory programming

In the distributed shared memory programming model, the global address space is divided into shared and private memory spaces. The shared memory space is logically partitioned, and each partition has affinity to a particular thread. In addition to the local portion of the shared memory, each thread also maintains its own private memory space.

The difference between the shared memory space and the private memory space is that, each thread can access any part of the global shared memory space, but it can only access its own private memory space.

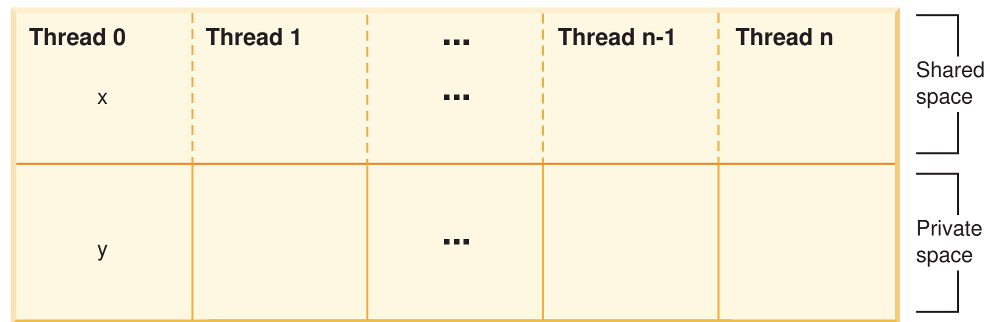


Figure 1. Unified Parallel C memory model

As illustrated in Figure 1, the shared variable *x* can be accessed by any thread, whereas the private variable *y* can only be accessed by thread 0. Although the variable *x* can be accessed from any thread, thread 0 can read or store a value in it much more efficiently than any other thread (x has affinity to thread 0). It is important to design your parallel algorithm in a way that minimizes accesses to shared data by threads that have no affinity to it.

Data affinity and data distribution

The following section describes the concept of data affinity to a thread, and introduces how data distribution works in Unified Parallel C.

Data affinity

Data affinity refers to the logical association between a portion of shared data and a given thread. In Figure 2 on page 4, each partition of shared memory space (partition i) has affinity to a particular thread (thread i). A well written Unified Parallel C program attempts to minimize communication between threads. An effective strategy for reducing unnecessary communication is to choose a data layout that maximizes accesses to shared data done by the thread with affinity to the data.

© Copyright IBM Corp. 2010

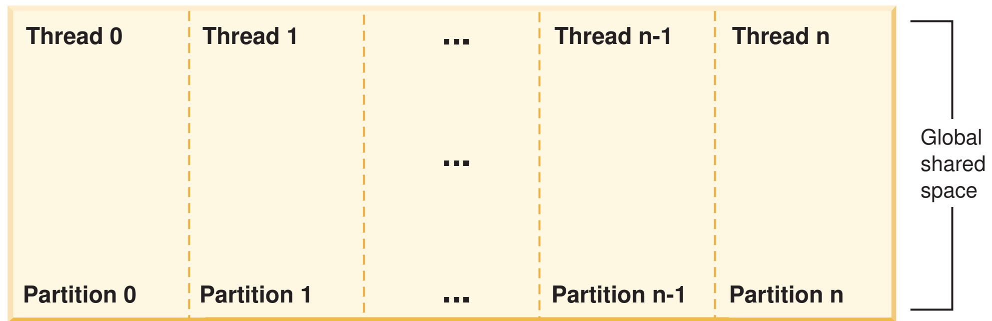


Figure 2. Data affinity

Data distribution

In the distributed shared memory programming model, data is distributed across multiple threads based on how it is declared. Unified Parallel C introduces the keyword `shared` as a type qualifier for declaring shared data, which is allocated in shared memory space. Declaration statements without the keyword `shared` declares private data, which is allocated in the private memory space of each thread.

When a private object is declared, Unified Parallel C allocates memory for it in the private memory space of each thread. For example, the declaration `int y` at file scope causes the allocation of the memory required to hold variable `y` in the private memory space of each thread. Each thread has its own private "copy" of the variable, and can only read or write its thread-local instance of `y`.

When a shared object is declared, Unified Parallel C allocates memory for it in shared memory space. For example, the declaration `shared int z` causes variable `z` to be allocated in the partition of shared memory space associated with thread 0. Although variable `z` has affinity to thread 0, it can be referenced and modified by all threads.

By default, a shared array is allocated in a round robin fashion one element at a time. For example, suppose that a program runs on 2 threads and contains the following shared array declaration:

```
shared int A[10];
```

Elements `A[0]`, `A[2]`, `A[4]`, `A[6]`, `A[8]` are allocated with affinity to thread 0, while array elements `A[1]`, `A[3]`, `A[5]`, `A[7]`, `A[9]` are allocated with affinity to thread 1.

A shared array can be allocated in blocks of consecutive elements by using a layout qualifier. The layout qualifier specifies the number of consecutive array elements to be allocated with affinity to the same thread. For example, suppose that a program runs on 2 threads and contains the following shared array declaration:

```
shared [2] int a[10];
```

This declaration has a layout qualifier which causes blocks of 2 consecutive elements to be allocated in a round robin fashion. Array elements `a[0]`, `a[1]`, `a[4]`, `a[5]`, `a[8]`, `a[9]` are allocated with affinity to thread 0; array elements `a[2]`, `a[3]`, `a[6]`, `a[7]` are allocated with affinity to thread 1.

Related reference

- Layout qualifiers
- "Blocking of shared arrays" on page 20
- "Shared and private data" on page 20

Memory consistency

In a multithread program, accesses to shared data issued by one thread might appear to other threads in a different order. For example, thread 0 might attempt to read a shared variable while another thread is writing a new value into it. Concurrent accesses to the same memory location might result in thread 0 reading the old value, a partially updated value, or the new value stored by the other thread.

Memory consistency modes define the order in which the results of write operations can be observed by read operations. By using the appropriate memory consistency mode, the user can decide when an update made to shared data by a thread is visible to other threads. Unified Parallel C provides the following two memory consistency modes:

strict
: In the strict mode, any change made to the shared data is immediately visible to other threads, and any operation on shared data begins only when the previous ones are complete. The compiler cannot change the sequence of independent shared access operations. Note that using the strict mode might increase the latency of program execution.

relaxed
: In the relaxed mode, threads can access shared data at any time, regardless of the changes made to it by other threads. An optimizing compiler is free to reorder the sequences of independent shared access operations.

The memory consistency mode can be set in the following program contexts:

- To set the strict or relaxed mode for the program scope, use the #include <upc_strict.h> or #include <upc_relaxed.h> directive.
- To set the strict or relaxed mode for a block scope, use the #pragma upc strict or #pragma upc relaxed directive.
- To set the strict or relaxed mode for a variable, use the reference type qualifier strict or relaxed.

Note: By default, shared variables follow the relaxed memory consistency semantics.

When a program contains conflicting memory consistency modes, the compiler uses the following precedence orders to determine the memory consistency mode that takes effect:

1. The mode set for a variable using strict or relaxed overrides the mode set for a block scope using #pragma upc strict or #pragma upc relaxed.
2. The mode set for a block scope using #pragma upc strict or #pragma upc relaxed overrides the mode set for the program scope using #include <upc_strict.h> or #include <upc_relaxed.h>.
3. The mode set for the program scope using #include <upc_strict.h> or #include <upc_relaxed.h> overrides the default memory consistency mode.

Synchronization mechanism

To manage thread interaction, Unified Parallel C provides three types of synchronization primitives: barriers, locks, and fences.

Barriers

A barrier is used to synchronize the executing threads at a given program point. It ensures that all threads reach a given program point before any of them proceeds further. Unified Parallel C has two types of barriers: the blocking barrier and the nonblocking barrier.

Figure 3 demonstrates the usage of the blocking barrier. Thread x and thread y are executed concurrently. Suppose that the barrier statement `upc_barrier` is at program point a, when thread x reaches program point a, it must stop and wait until thread y and other threads reach program point a. No thread can proceed further in the execution of the program until all threads reach the barrier.

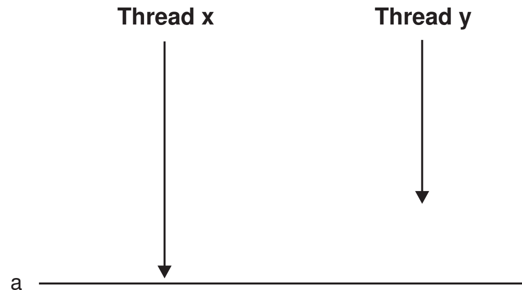


Figure 3. The blocking barrier

Figure 4 on page 7 demonstrates the usage of the nonblocking barrier. The nonblocking barrier, also referred to as the split-phase barrier, consists of the `upc_notify` and `upc_wait` statements. Thread x executes the `upc_notify` statement at program point a and notifies all other threads that it has reached program point a. Then thread x proceeds further, and stops when it executes the `upc_wait` statement at program point b. Thread x proceeds further from program point b only when thread y and other threads reach program point a and execute the `upc_notify` statement to report their presence.

From program point a to program point b, thread x can perform local computations that do not require synchronization with other threads. This has the effect of overlapping communication with local computation, partially hiding the latency of the communication.

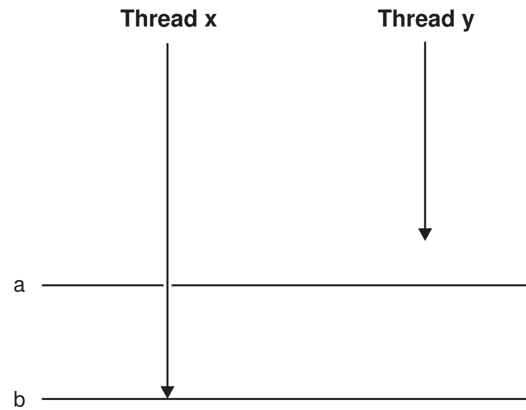


Figure 4. The nonblocking barrier

Locks

A lock ensures that shared data is accessed by only one thread at a time. It can be dynamically allocated and used to coordinate accesses to the critical part of the program.

To allocate locks, Unified Parallel C provides the following two functions:

upc_global_lock_alloc
This function allocates a lock dynamically. It is to be called by one thread. If the function is called by multiple threads, each calling thread gets a different allocation.

upc_all_lock_alloc
This function is a collective function that allocates the same lock to each thread.

To use locks, Unified Parallel C provides the following functions:

upc_lock
This function sets the state of a lock as locked. If the lock is already in a locked state, the calling thread has to wait until the lock is unlocked by the thread that locked it. If the lock is in an unlocked state, the calling thread locks it.

upc_lock_attempt
If the lock is in a locked state, this function returns `0`; if the lock is in an unlocked state, the calling thread locks it and the function returns 1.

upc_unlock
The locking thread can use this function to release the lock it had previously acquired.

To free a previously allocated lock, Unified Parallel C provides the following function:

upc_lock_free
This function is used to release the memory allocated for a lock.

Fences

A fence can be used to synchronize shared memory accesses locally. `upc_fence` ensures that the execution of any shared access issued after the fence begins only

after all shared accesses issued before the fence are complete.

Related reference

- "Blocking barriers" on page 42
- "Nonblocking barriers" on page 43
- "Fences" on page 45
- "Serialization" on page 62

Chapter 3. Using the XL Unified Parallel C compiler

The following sections provide information about using the XL Unified Parallel C compiler.

Compiler options

The XL Unified Parallel C compiler supports a subset of the XL C compiler options and adds some new Unified Parallel C specific options. It also modifies some existing XL C options.

The following sections list the XL Unified Parallel C compiler options that are different from the XL C compiler options. For the compiler options that are common between the two compilers, consult the *XL C Compiler Reference*.

Note: The XL Unified Parallel C compiler does not support any C++ compiler options.

New compiler options

This section describes the new option that the XL Unified Parallel C compiler introduces.

Table 1. New compiler options

Option name	Description
-qupc	Specifies the number of static threads or the number of nodes in a cluster for a Unified Parallel C program.

-qupc

Category

Optimization and tuning

Pragma equivalent

None

Purpose

Specifies the number of static threads or the number of nodes in the cluster used for the execution of the Unified Parallel C program.

Syntax

```
>>— -q—upc—=——┬threads┬——=——number————————————————————————————————><
              └dnodes─┘
```

Defaults

Not applicable

Parameters

threads
Specifies the number of static THREADS to be used in the execution of the Unified Parallel C program.

dnodes
Specifies the number of nodes in a cluster that are used for the execution of the Unified Parallel C program.

number
Represents the number of static threads or the number of nodes. It is an integer literal in the range of 1 and 65535.

Usage

The options **-qupc=threads** and **-qupc=dnodes** are used to specify the number of threads and nodes for the execution of a Unified Parallel C program at compile time. Specifying the two options assists the compiler with the locality analysis that is required to enable some optimizations. If you do not specify **-qupc=threads** or **-qupc=dnodes** when you compile a Unified Parallel C program, the compiler assumes that the program runs in the dynamic environment.

The Parallel Operating Environment (POE) command-line options or environment variables can be used to set the number of threads and nodes. For more information, consult the POE documentation.

If you specify **-qupc=dnodes=M** without specifying **-qupc=threads**, the compiler assumes *M* threads and *M* nodes for the generated program.

If you specify both **-qupc=threads=N** and **-qupc=dnodes=M**, ensure that `N ≥ M` and `N` must be a multiple of `M` (that is `N % M = 0`).

Predefined macros

None

Examples

To compile a `hello1.upc` program and set the generated program to use 4 static threads and run on 4 nodes in a cluster, enter the following command:

```
xlupc -qupc=threads=4 -qupc=dnodes=4 hello1.upc
```

In this example, the generated program runs 1 thread per node. The program must be run on 4 nodes.

To compile a `hello2.upc` program and set the generated program to use 8 static threads and run on 4 nodes in a cluster, enter the following command:

```
xlupc -qupc=theads=8 -qupc=dnodes=4 hello2.upc
```

In this example, the generated program runs 2 threads per node. The program must be run on 4 nodes.

Modified compiler options

This section describes the XL Unified Parallel C compiler options that are used differently from the same XL C compiler options.

Table 2. Modified compiler options

Option name	Description	Changes in XL Unified Parallel C compiler
-qalias	Indicates whether a program contains certain categories of aliasing or does not conform to C standard aliasing rules.	Supports only the following suboptions: • **ansi** • **noansi** The default suboption is **ansi**.
-qalign	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.	The following suboptions are not supported: • **mac68k** • **twobyte**
-qarch	Specifies the processor architecture for which the code (instructions) should be generated.	Supports only the suboption, **pwr7**
-qignprag	Instructs the compiler to ignore certain pragma statements.	The following suboptions are not supported: • **ibm** • **omp**
-qkeyword	Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.	The following suboptions are not supported: • **inline** • **restrict**
-qlanglvl	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or superset of a standard.	Supports only the following suboptions: • **stdc99** • **extc99** The default suboption is **extc99**.
-qlistfmt	Creates a report to assist with finding optimization opportunities.	The following suboptions are not supported: • **pdf** • **nopdf**
-O, -qoptimize	Specifies whether to optimize code during compilation and, if so, at which level.	The following optimization levels are not supported: • **-O4** • **-O5**
-qpath	Determines substitute path names for XL UPC executables such as the compiler, assembler, linker, and preprocessor.	The following suboptions are not supported: • **I** • **L**

Table 2. Modified compiler options (continued)

Option name	Description	Changes in XL Unified Parallel C compiler
-qsourcetype	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.	The following suboptions are not supported: • **assembler** • **assembler-with-cpp** Adds the new suboption, **upc**.
-qstrict_induction	Prevents the compiler from performing induction (loop counter) variable optimizations.	The default suboption is: • **-qstrict_induction** • **-qnostrict_induction** when **-O3** is in effect
-t	Applies the prefix specified by the **-B** option to the designated components.	The following suboptions are not supported: • **I** • **L**
-qtune	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.	Supports only the following suboptions: • **pwr7** • **auto** • **balanced**

Unsupported compiler options

This section describes the XL C compiler options that the XL Unified Parallel C compiler does not support.

- -q32, -q64
- -qaltivec
- -qcompact
- -qcpluscmt
- -qdatalocal
- -qdataimported
- -qdigraph
- -qdirectstorage
- -qdollar
- -qdpcl
- -e
- -f
- -qfdpr
- -qfunctrace
- -qgenproto
- -qhot
- -qipa
- -qlibmpi
- -qmacpstr
- -O4

- -O5
- -qoptdebug
- -qpascal
- -qpdf1, -qpdf2
- -qprolocal, -qprocimported, -qprocunknown
- -qproto
- -Q
- -qshowpdf
- -qsmallstack
- -qsmp
- -qspeculateabsolutes
- -qstatsym
- -qtabsize
- -qtrigraph
- -qupconv
- -qwarn64
- -qweakexp
- -qweaksymbol
- -qvecnvol

Compiler commands

This section provides information about invoking the XL Unified Parallel C compiler and setting the target execution environment for the program.

Invoking the compiler

This section provides information about invoking the XL Unified Parallel C compiler and compiling programs for different execution environments.

Invocation command

To compile a Unified Parallel C program, use the **xlupc** command to invoke the compiler. By default, a .c file is compiled as a Unified Parallel C program unless the option **-qsourcetype=c** is specified.

Note: If you want to mix .c files with .upc files in your application, .c files can be compiled and linked with **xlupc**.

Execution environments

The execution environment of a Unified Parallel C program can be static or dynamic. In the static environment, the number of threads for the target program is known at compile time. In the dynamic execution environment, neither the number of threads nor the number of nodes is known at compile time.

To set the number of threads for a program in the static environment, you can use the **-qupc=threads** option. For example, to compile the `test.upc` program that will run with 4 threads, enter the following command:

```
xlupc -o test1 -qupc=threads=4 test.upc
```

To set the number of threads for a program in the dynamic environment, you can use the following command:

```
export UPC_NTHREADS=N
```

Where `N` is the number of threads that the program will run with. The environment variable *UPC_NTHREADS* can be used to specify the number of threads that a Unified Parallel C program will run with in both the static and dynamic environments.

Note: If the number of threads for a program was specified at compile time, it is not allowed to attempt to run the compiled program with a different number of threads.

To set the number of nodes for a program in the static environment, you can use the compiler option **-qupc=dnodes=M**, where `M` is the number of nodes that the program will run on. To compile the `test.upc` program that will run with `N` threads on `M` nodes, enter the following command:

```
xlupc -qupc=threads=N -qupc=dnodes=M test.upc
```

Where `N ≥ M` and `N` must be a multiple of `M` (that is `N % M = 0`).

The executable program must be run on the same number of nodes as specified by the **-qupc=dnodes** option when compiling the program. To run the executable program, you must use the IBM Parallel Operating Environment (POE). For example, to run the executable program, `a.out`, enter the following command:

```
a.out -procs 3 -msg_api LAPI -hostfile hosts
```

Where:

-procs Specifies the number of processes.

-msg_api
Indicates to POE which message passing API is being used by the parallel job.

-hostfile
Specifies a host file that contains the names or the IP address of the hosts used.

For example, to specify 3 nodes, you can create the host file `hosts` listing the IP addresses of the 3 nodes in the cluster as follows:

```
1.2.3.4
1.2.3.5
1.2.3.6
```

Another example is given here to demonstrate how to specify the number of nodes in the dynamic environment.

```
xlupc test.upc

export UPC_NTHREADS=8

a.out -procs 4 -msg_api LAPI -hostfile hosts
```

Note: For more information about using POE, consult its documentation.

Compiling and running an example program

This section provides a simple Unified Parallel C program, the commands to compile and execute the program, and the program result.

In the following example (`hello.upc`), each thread prints a message to standard output:

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

void main()
{
  printf("Hello world! (THREAD %d of %d THREADS)\n", MYTHREAD, THREADS);
  return 0;
}
```

Use the following command to compile the program in the static environment targeting 4 threads:

```
xlupc -o hello -qupc=threads=4 hello.upc
```

The compiler compiles the code and generates an executable file, `hello`. To run the executable program, you can use the following command:

```
poe ./hello -hostfile hosts -procs 1 -msg_api lapi
```

Where **-procs** specifies the number of processes to use. The program prints to standard output a message on each thread, for example:

```
Hello world! (THREAD 3 of 4 THREADS)
Hello world! (THREAD 1 of 4 THREADS)
Hello world! (THREAD 0 of 4 THREADS)
Hello world! (THREAD 2 of 4 THREADS)
```

Related reference

- “Predefined identifiers” on page 17
- Execution environments

Chapter 4. Unified Parallel C language

This chapter describes the semantics and syntax of the Unified Parallel C parallel programming language.

Predefined identifiers

This section provides information about the predefined identifiers that the XL Unified Parallel C compiler defines: THREADS, MYTHREAD, and UPC_MAX_BLOCK_SIZE.

THREADS
Indicates the number of threads that are used in the current program execution. THREADS is an expression with value of type int, and has the same value on every thread. It is an integer constant in the static environment.

MYTHREAD
Represents the index of the thread that is currently being executed. The value is an integer ranging from 0 to THREADS-1.

UPC_MAX_BLOCK_SIZE
Indicates the maximum block size allowed by the compilation environment for shared data. The value is a predefined integer constant.

Unary operators

Unified Parallel C extends the C operators by adding the following unary operators:

- The address operator &
- The sizeof operator
- The upc_blocksizeof operator
- The upc_elemsizeof operator
- The upc_localsizeof operator

The address operator &

The address operator & returns a pointer to its operand. When the operand of the operator & is a shared object of Type T, the result has type shared [] T *. For instance:

```
typdef struct t
{
  int a;
} myt;

shared myt st; /*declares a shared scalar structure object st which is located in
                 the shared space of Thread 0
             */

shared[] int *p;

p=&st.a        // &st.a has the type shared[] int *
```

In the example code, the pointer-to-shared p is initialized with the address to the shared structure member st.a.

The sizeof operator

The `sizeof` operator returns the size, in bytes, of the operand. The operand can be either an expression or the parenthesized name of a type. You can apply the `sizeof` operator to shared data and shared types.

The sizeof operator syntax

```
►►──sizeof──┬─unary-expression─┬──────────────────────────────────────►◄
            └─(──type name──)──┘
```

When the `sizeof` operator is applied to a shared array with a definite block size in the dynamic environment, the operator returns a nonconstant integer value. All constraints on the C `sizeof` operator also apply to this operator in Unified Parallel C.

Example

```
shared int a;
shared [5] int b[5*THREADS];
int c;

sizeof(a); /* The return value is sizeof(int). The return value might be different
              depending on the 32bit mode or the 64bit mode
           */

sizeof(b); // The return value is 5*THREADS*sizeof(int).
sizeof(c); // Like in the C language, the operator returns sizeof(int).
```

The upc_blocksizeof operator

The `upc_blocksizeof` operator returns the block size of an operand, which is the value specified in the layout qualifier of a type declaration. The return value is an integer of type `size_t`.

If no layout qualifier is specified, the operator returns the default block size 1. If the operand of `upc_blocksizeof` has an indefinite block size, the operator returns 0.

The upc_blocksizeof operator syntax

```
►►──upc_blocksizeof──┬─unary-expression─┬─────────────────────────────►◄
                     └─(──type name──)──┘
```

The `upc_blocksizeof` operator applies only to shared-qualified expressions or types. If the operand is an expression, that expression is not evaluated. All constraints on the `sizeof` operator described in C also apply to this operator.

Example

```
shared [] int a[5];   // upc_blocksizeof(a)=0
shared [5] int b[5];  // upc_blocksizeof(b)=5
shared int c;         // upc_blocksizeof(c)=1
upc_blocksizeof(shared[3] int) = 3;
```

Related reference

"Type qualifiers" on page 39.

The upc_elemsizeof operator

The `upc_elemsizeof` operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For nonarray objects, `upc_elemsizeof` returns the same value as `sizeof`. This rule also applies to the array defined by `typedef`.

The upc_elemsizeof operator syntax

```
►►──upc_elemsizeof──┬─unary-expression─┬──────────────────────────────────►◄
                    └─(──type name──)──┘
```

The `upc_elemsizeof` operator only applies to shared-qualified expressions or types, and it returns an integer constant of type `size_t`. All constraints on the `sizeof` operator described in C also apply to this operator.

Example

```
typedef shared [] char type;
type a[10];           // upc_elemsizeof(a)=sizeof(char)
shared [3] int b[20]; // upc_elemsizeof(b)=sizeof(int)
shared [3] char c[9]; // upc_elemsizeof(c)=sizeof(char)
shared int d;         // upc_elemsizeof(d)=sizeof(int)
```

The upc_localsizeof operator

The `upc_localsizeof` operator returns the size of the local portion of a shared object or a shared-qualified type. The return value is an integer constant of type `size_t`. All threads receive the same return value, which is the maximum size of the portion that can be allocated with affinity to each thread.

The upc_localsizeof operator syntax

```
►►──upc_localsizeof──┬─unary-expression─┬─────────────────────────────────►◄
                     └─(──type name──)──┘
```

The `upc_localsizeof` operator only applies to shared-qualified types or expressions. All constraints on the `sizeof` operator described in C also apply to this operator.

Example

Assume that there are 3 THREADS:

```
shared [] int a[10];  // upc_localsizeof(a)=10*sizeof(int)
shared [3] int b[10]; // upc_localsizeof(b)=4*sizeof(int)
shared [3] int c[9];  // upc_localsizeof(c)=3*sizeof(int)
shared int d[10];     // upc_localsizeof(d)=4*sizeof(int)
```

Data and pointers

This section provides information about shared data declarations and pointers to shared data in Unified Parallel C.

Shared and private data

Unified Parallel C has two types of data, shared data and private data. Shared data is declared with the `shared` type qualifier, and is allocated in the shared memory space. Private data is declared without the `shared` type qualifier. One instance of the private data is allocated in the private memory space of each thread. Private data can be accessed only by the thread to which it has affinity.

Example

This example shows the data layout for different declarations. The following code declares three identifiers of different types. Assuming four threads, these data objects are allocated as shown in Figure 5.

```
int a;           // private scalar variable
shared int b;     // shared scalar variable
shared int c[10]; // shared array
```

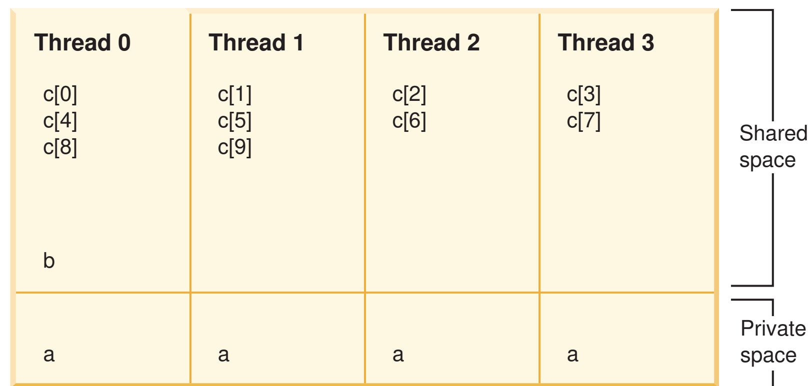


Figure 5. Memory allocation and data affinity

Figure 5 shows that a thread-local instance of the scalar variable *a* is allocated in the private memory space of each thread. The shared scalar variable *b* is allocated in the shared memory space of thread 0. The array elements of the shared array `c` are allocated in the shared memory space in a round robin manner across all threads.

Blocking of shared arrays

By default, shared array elements are allocated in the shared memory space across all threads in a round robin fashion. However, you can change the default way of distributing shared array elements by declaring the shared array with a layout qualifier.

The layout qualifier instructs the compiler to allocate the specified number of consecutive array elements in the shared memory space of the same thread. By using the layout qualifier, you can optimize the distribution of a shared array and minimize the number of shared remote accesses that your parallel computation performs. If there is no layout qualifier, the default value is 1.

The affinity that an array element has to a particular thread is determined by the thread number and the block size. For example, in the shared array declared as follows:

```
shared [block_size] A [number-of-elements]
```

element `i` of array A has affinity to thread `(i/block_size)%THREADS`.

Examples

Example 1: A program declares the shared array `shared[3] int a[14]` and is run by 3 threads. The data layout of the shared array `a` is shown in Figure 6.

Thread 0	Thread 1	Thread 2
a[0] a[1] a[2] a[9] a[10] a[11]	a[3] a[4] a[5] a[12] a[13]	a[6] a[7] a[8]

Figure 6. data layout of one-dimensional array

In Figure 6, every three contiguous array elements are grouped as a block and are allocated in a block-cyclic manner across all threads. Each block of elements has affinity to a particular thread.

Example 2: A program declares the following shared array and is run by 3 threads:

```
typedef int m[3];
shared [2] m a[5];
```

The block size always applies to the underlying scalar type, regardless of whether there are any `typedefs` involved. The underlying scalar type is the leftmost nonarray type. The array is blocked as if it were declared as follows:

```
shared [2] int a[5][3];
```

The data layout of the shared array `a` is shown in Figure 7.

Thread 0	Thread 1	Thread 2
a[0][0] a[0][1]	a[0][2] a[1][0]	a[1][1] a[1][2]
a[2][0] a[2][1]	a[2][2] a[3][0]	a[3][1] a[3][2]
a[4][0] a[4][1]	a[4][2]	

Figure 7. Data layout of two-dimensional array

Shared and private pointers

Unified Parallel C allows the declaration of four types of pointers. The four types of pointers allow for the fact that the pointer variable, the data pointed to, or both might be shared-qualified.Table 3 illustrates the four types of pointers as shown in Figure 8

Table 3. Unified Parallel C pointer types

Where the pointer resides	Where the pointer points	Pointer type
private	private	Private-to-private pointer
private	shared	Private-to-shared pointer
shared	private	Shared-to-private pointer
shared	shared	Shared-to-shared pointer

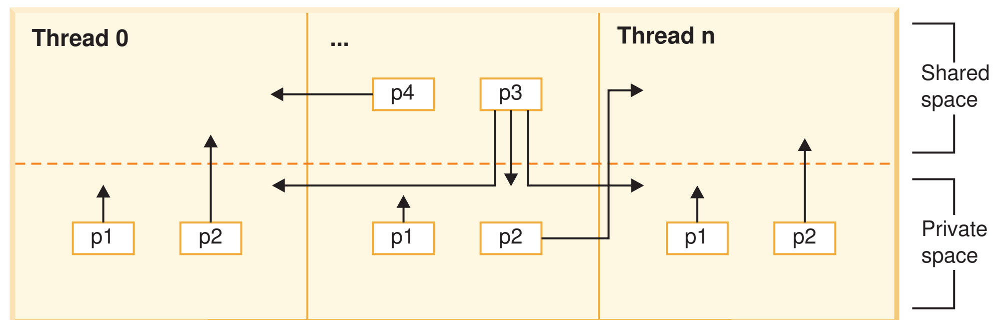


Figure 8. The memory view of the four types of Unified Parallel C pointers

Private-to-private pointer

The following statement declares a private-to-private pointer p1:

```
int *p1;
```

p1 is a private pointer that points to private data. Unified Parallel C allocates memory for an instance of p1 in the private memory space of each thread. This pointer can be used to access data in the private memory space of a thread.

Example

The following example demonstrates the usage of the private-to-private pointer. Figure 9 on page 23 illustrates where p1 is located and where it points.

```
#include <upc.h> // assume 2 threads

int a=-1;

void main()
{
   int *p1;

   if(MYTHREAD == 1)
   {
      // p1 points to the dynamically allocated private space.
      p1=malloc(1*sizeof(int));
      *p1=1;
   }
```

```
   else
   {
      // p1 points to the variable a.
      p1=&a;
   }
   printf("Th=%d,*p1=%d\n",MYTHREAD,*p1);
}
```

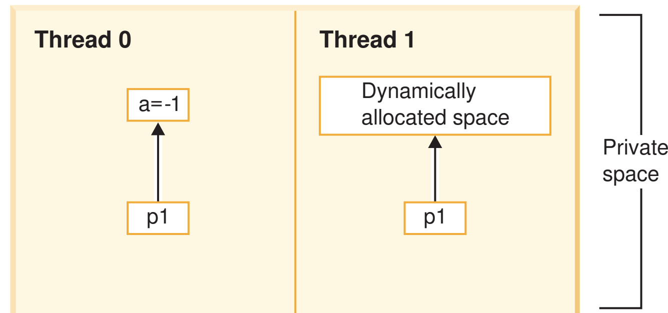


Figure 9. The memory view of the private-to-private pointer p1

The output of this program is as follows:

```
Th=0,*p1=-1
Th=1,*p1=1
```

Private-to-shared pointer

The following statement declares a private-to-shared pointer p2:

```
shared int *p2;
```

p2 is private pointer that points to shared data. Each thread has an independent and private instance of p2. This type of pointer can be used to access shared data. Unified Parallel C allocates memory for an instance of p2 in the private memory space of each thread.

Example

The following example demonstrates the usage of the private-to-shared pointer. Figure 10 on page 24 illustrates where p2 is located and where it points.

```
#include <upc.h> // assume 2 threads

shared int a=0;

void main()
{

   shared int *p2; // p2 is a private pointer to shared

   // THread 1 initializes variable a
   if(MYTHREAD == 1)
     a=-1;
   upc_barrier;

   // each pointer p2 points to the shared space where shared variable a stays
   p2=&a;

   // every thread dereferences p2, and value should be the same for each thread
   printf("Th=%d,*p2=%d\n",MYTHREAD,*p2);
}
```

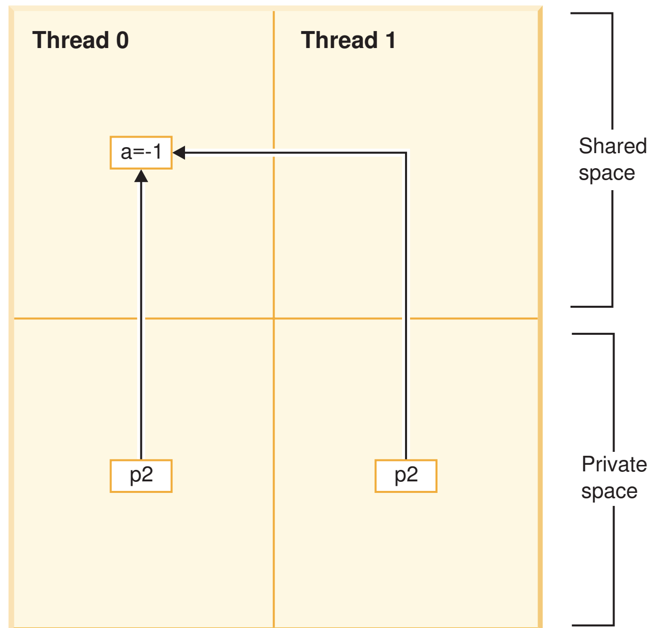


Figure 10. The memory view of the private-to-shared pointer p2

The output of this program is as follows:

```
Th=1,*p2=-1
Th=0,*p2=-1
```

Shared-to-private pointer

The following statement declares a shared-to-private pointer p3:

```
int * shared p3;
```

p3 is a shared pointer that points to private data. Any thread can dereference this pointer. However, the value obtained is well-defined only for the thread that initialized the pointer, or for threads co-located on the same node.

Example

The following example demonstrates the usage of the shared-to-private pointer. Figure 11 on page 25 illustrates where p3 is located and where it points.

```
# include <upc.h> // assume 2 threads

int *shared p3;

void main()
{
   int a=MYTHREAD+99;

   if(MYTHREAD == 1)
   {
     // assign the address of thread 1's variable a to p3.
     p3=&a;
   }

   upc_barrier;

   printf("Th=%d,*p3=%d\n",MYTHREAD,*p3);
}
```

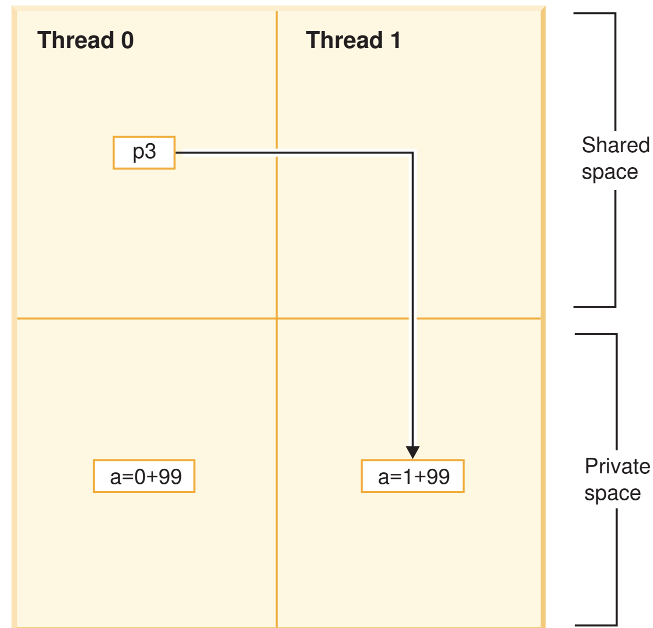


Figure 11. The memory view of the shared-to-private pointer p3

The output of this program is as follows:

```
Th=1,*p3=100 // defined behaviour
Th=0,*p3=99  // undefined behaviour
```

Shared-to-shared pointer

The following statement declares a shared-to-shared pointer p4:

```
shared int *shared p4;
```

p4 is a shared pointer that points to shared data. Memory for the pointer p4 is allocated on thread 0.

Example

The following example demonstrates the usage of the shared-to-shared pointer. Figure 12 on page 26 illustrates where p4 is located and where it points.

```
# include <upc.h>                // assume 2 threads

shared int *shared p4[THREADS]; // declare a shared pointer array
shared int a[THREADS];          // declare a shared array

void main()
{
  a[MYTHREAD]=MYTHREAD+99;

  if(MYTHREAD == 0)
    p4[MYTHREAD]=&a[1];
  else
    p4[MYTHREAD]=&a[0];

  upc_barrier;

  printf("Th=%d,*p4[%d]=%d\n",MYTHREAD,MYTHREAD,*p4[MYTHREAD]);
}
```

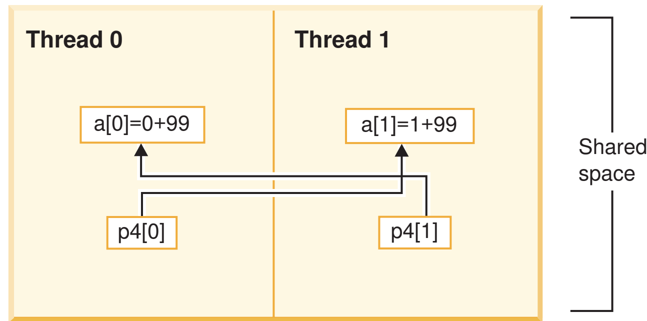


Figure 12. The memory view of the shared-to-shared pointer p4

The output of this program is as follows:

```
Th=1,*p4[1]=99
Th=0,*p4[0]=100
```

Related reference

"upc_threadof" on page 61

Pointer-to-shared arithmetic

A pointer-to-shared is a pointer that points to shared data. Unlike a C pointer, a pointer-to-shared tracks the following information:

- The thread that the pointer-to-shared currently points to
- The offset that is used to calculate the phase within that thread

When incremented, a pointer-to-shared with an indefinite block size behaves similarly to a C pointer. After the increment, the thread of the pointer remains unchanged, and the phase of the pointer is always 0.

When incremented, a pointer-to-shared with a definite block size advances according to array element order. After the increment, both the thread and phase of the pointer might change. The following example describes the pointer arithmetic:

```
# include <upc.h> // assume 3 THREADS

#define B 3
#define N 20

shared [B] int arr[N];
shared [B] int *p, *p1;

void main()
{
  int i;
  upc_forall(i=0;i<N;i++;&arr[i])
  {
    arr[i]=i;
  }
  upc_barrier;
  p=&arr[0]; // p point to arr[0]
  p1=p+5;    // p1 will point to arr[5] after executing the statement

  printf("1-Th=%d,*p=%d,Phaseof(p)=%d,Threadof(p)=%d\n",MYTHREAD,*p,upc_phaseof(p),
  upc_threadof(p));

  printf("2-Th=%d,*p1=%d,Phaseof(p1)=%d,Threadof(p1)=%d\n",MYTHREAD,*p1,upc_phaseof(p1),
```

```
  upc_threadof(p1));

  upc_threadof(p1));
}
```

The output of the program is as follows:

```
1-Th=1,*p=0,Phaseof(p)=0,Threadof(p)=0
2-Th=1,*p1=5,Phaseof(p1)=2,Threadof(p1)=1
1-Th=0,*p=0,Phaseof(p)=0,Threadof(p)=0
1-Th=2,*p=0,Phaseof(p)=0,Threadof(p)=0
2-Th=0,*p1=5,Phaseof(p1)=2,Threadof(p1)=1
2-Th=2,*p1=5,Phaseof(p1)=2,Threadof(p1)=1
```

After the assignment `p1=p+i`, the following equations must be true in any Unified Parallel C implementation. In each case, the `div` indicates integer division rounding towards negative infinity and the `mod` operator returns the nonnegative remainder:

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

Comparison between two pointers-to-shared is meaningful only when the two pointers point to the same array.

Subtraction between two pointers-to-shared is meaningful only when there exists an integer x that satisfies both the following conditions:

```
pts1 + x == pts2
upc_phaseof (pts1 + x) == upc_phaseof (pts2)
```

`pts1` and `pts2` are two pointers-to shared. The result of `(pts2-pts1)` equals to `x`, and in this case, the subtraction is meaningful.

Note: Binary operations (comparison, subtraction) between pointers-to-shared and pointers-to-private are meaningless, because these two different types of pointers have types that are not compatible.

Examples

Example 1, Example 2 and Example 3 illustrate various pointer arithmetic operations on pointers-to-shared having definite block size.

Example 1

```
# include <upc.h> // assume 3 threads

# define N 8

shared[2] int a[N];
shared[2] int *p1,*p2;

void main()
{
   int i=0;
   upc_forall(i=0;i<N;i++;i)
      a[i]=i;
   upc_barrier;

   p1=&a[1];
   p2=p1+MYTHREAD;  //note:p2-p1 must be equal to MYTHREAD
   printf("Th:%d,*p1=%d,*p2=%d\n",MYTHREAD,*p1,*p2);
}
```

Figure 13 demonstrates where the pointers in this program point.

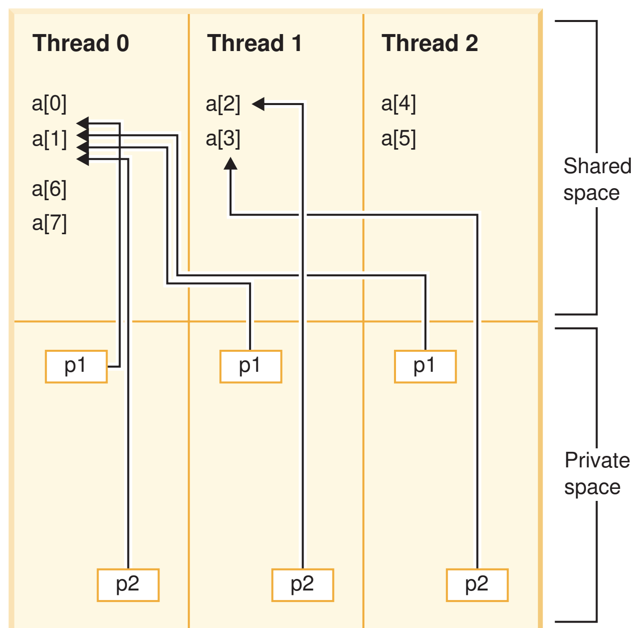


Figure 13. The memory view of pointers in Example 1

The output of this program is as follows:

```
Th:0,*p1=1,*p2=1,p2-p1=0
Th:2,*p1=1,*p2=3,p2-p1=2
Th:1,*p1=1,*p2=2,p2-p1=1
```

Example 2

```
# include <upc.h> // assume 3 threads

# define N 8

shared[2] int a[N];
shared[2] int *p1,*p2;

void main()
{
   int i=0;
   upc_forall(i=0;i<N;i++;i)
      a[i]=i;
   upc_barrier;

   p1=a+1+MYTHREAD;
   p2=p1+MYTHREAD;
   printf("Th:%d,*p1=%d,*p2=%d\n",MYTHREAD,*p1,*p2);
}
```

Figure 14 on page 29 demonstrates where the pointers in this program point.

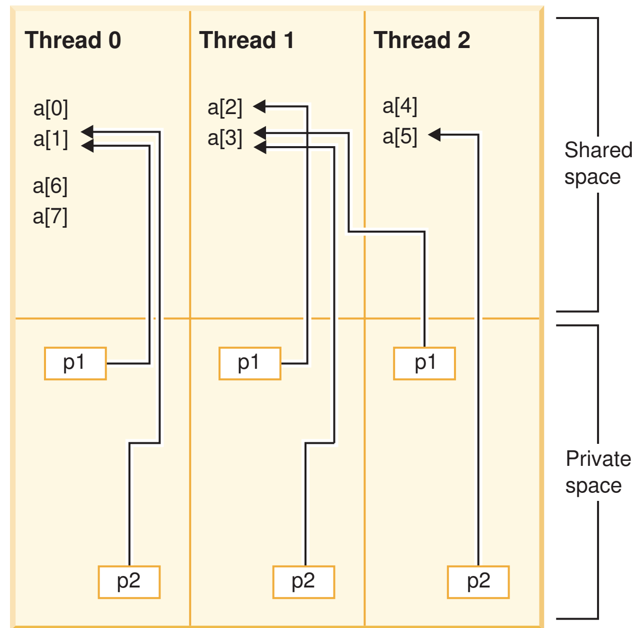


Figure 14. The memory view of pointers in Example 2

The output of this program is as follows:

```
Th:1,*p1=2,*p2=3
Th:2,*p1=3,*p2=5
Th:0,*p1=1,*p2=1
```

Example 3

```
# include <upc.h> // assume 3 threads

# define N 8

shared[2] int a[N];
shared[2] int *p1,*shared p2;

void main()
{
   int i=0;
   upc_forall(i=0;i<N;i++;i)
      a[i]=i;
   upc_barrier;

   p1=&a[1];
   p2=&a[5];

   upc_barrier;

   p1++;
   if(MYTHREAD == 0)
     p2--;

   upc_barrier;
   printf("Th:%d,*p1=%d,*p2=%d\n",MYTHREAD,*p1,*p2);
   printf("phase(p1)=%d,threadof(p1)=%d\n",upc_phaseof(p1),upc_threadof(p1));
   printf("phase(p2)=%d,threadof(p2)=%d\n",upc_phaseof(p2),upc_threadof(p2));
}
```

Figure 15 on page 30 demonstrates where the pointers in this program point.

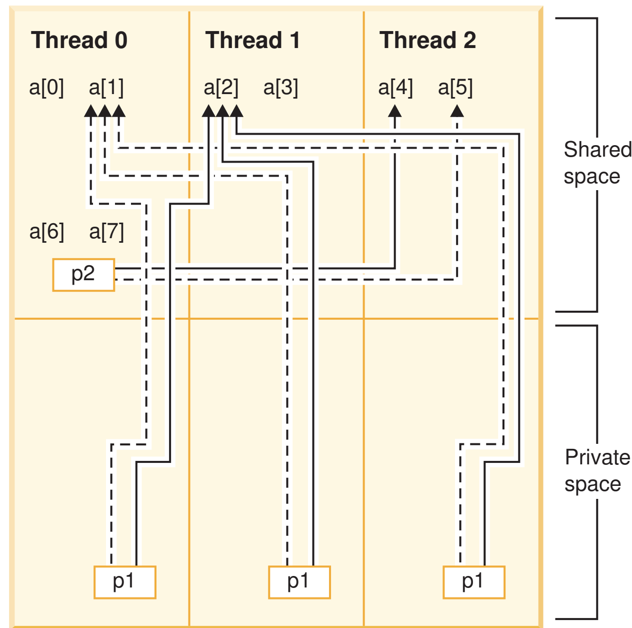


Figure 15. The memory view of pointers in Example 3

The output of this program is as follows:

```
Th:1,*p1=2,*p2=4
Th:2,*p1=2,*p2=4
phase(p1)=0,threadof(p1)=1
phase(p1)=0,threadof(p1)=1
phase(p2)=0,threadof(p2)=2
phase(p2)=0,threadof(p2)=2
Th:0,*p1=2,*p2=4
phase(p1)=0,threadof(p1)=1
phase(p2)=0,threadof(p2)=2
```

Example 4 and Example 5 show how pointers-to-shared with an indefinite block size are incremented.

Example 4

```
# include <upc.h> // assume 3 threads

# define N 8

shared[] int a[N];
shared[] int *p1,*shared p2;

void main()
{
   int i=0;
   upc_forall(i=0;i<N;i++;i)
      a[i]=i;
   upc_barrier;

   p1=&a[1];
   if(MYTHREAD == THREADS -1)
     p2=p1+5;

   upc_barrier;

   printf("Th:%d,*p1=%d,*p2=%d\n",MYTHREAD,*p1,*p2);
   printf("phaseof(p2)=%d,threadof(p2)=%d\n",upc_phaseof(p2),upc_threadof(p2));
}
```

Figure 16 demonstrates where the pointers in this program point.

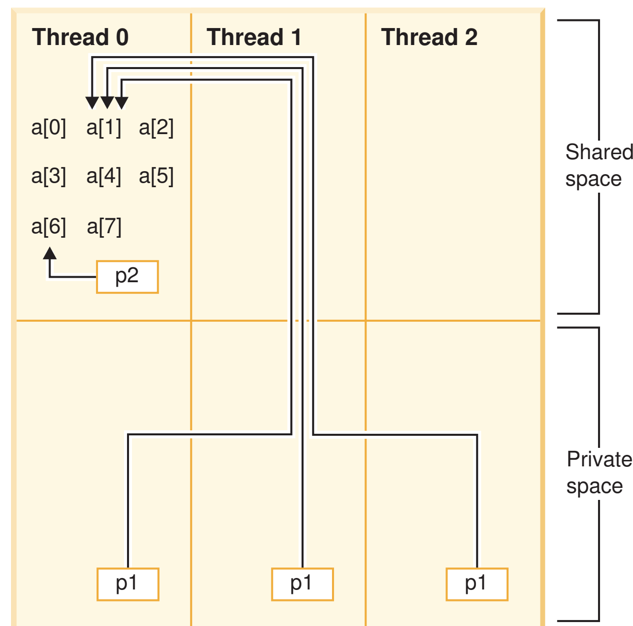


Figure 16. The memory view of pointers in Example 4

The output of this program is as follows:

```
Th:2,*p1=1,*p2=6
Th:1,*p1=1,*p2=6
phaseof(p2)=0,threadof(p2)=0
Th:0,*p1=1,*p2=6
phaseof(p2)=0,threadof(p2)=0
phaseof(p2)=0,threadof(p2)=0
```

Example 5

```
# include <upc.h> // assume 3 threads

# define N 8

shared[] int *shared p1;
shared[] int *p2;

void main()
{
   int i=0;
   if(MYTHREAD == 1)
     p1=upc_alloc(N*sizeof(int));
   upc_barrier;
   upc_forall(i=0;i<N;i++;i)
      p1[i]=i;
   upc_barrier;

   p2=p1+MYTHREAD;

   upc_barrier;

   printf("Th:%d,*p1=%d,*p2=%d\n",MYTHREAD,*p1,*p2);
   printf("phaseof(p2)=%d,threadof(p2)=%d\n",upc_phaseof(p2),upc_threadof(p2));
}
```

Figure 17 on page 32 demonstrates where the pointers in this program point.

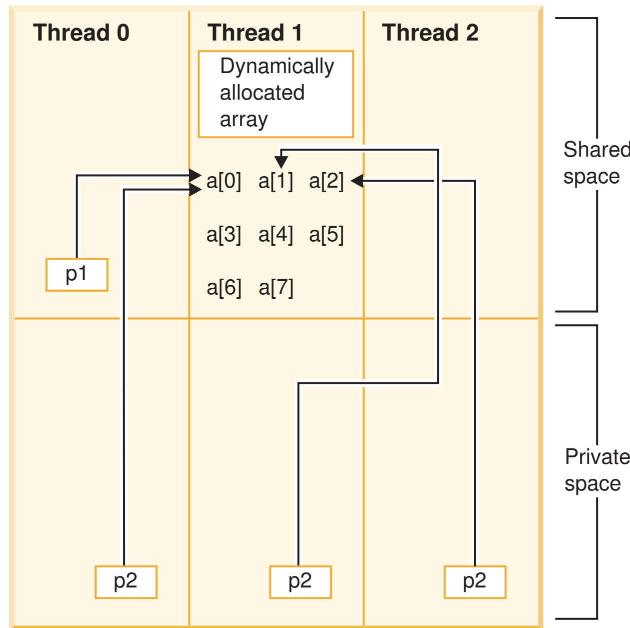


Figure 17. The memory view of pointers in Example 5

The output of this program is as follows:

```
Th:1,*p1=0,*p2=1
phaseof(p2)=0,threadof(p2)=1
Th:2,*p1=0,*p2=2
phaseof(p2)=0,threadof(p2)=1
Th:0,*p1=0,*p2=0
phaseof(p2)=0,threadof(p2)=1
```

Related reference

"upc_phaseof" on page 59

Cast and assignment expressions

Casting between pointers in Unified Parallel C can be classified into the following categories:

- Cast of a pointer-to-shared to a pointer-to-shared
- Cast of a pointer-to-shared to a pointer-to-private
- Cast of a pointer-to-private to a pointer-to-shared
- Cast of a pointer-to-private to a pointer-to-private

Cast of a pointer-to-shared to a pointer-to-shared

If you cast between two pointers-to-shared that have different type sizes or block sizes, the phase of the resulting pointer is zero.

The following example demonstrates the casting between two pointers-to-shared with different block sizes.

```
# include <upc.h> // assume 4 threads

# define FAILURE 166
# define SUCCESS 155

# define BLKSIZE  3
```

```
# define ARRSIZE  60

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  {   \printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
  }   \
}

shared int *p1,arr2[ARRSIZE];
shared[BLKSIZE] int *p2,arr1[ARRSIZE];

int main()
{
  int i;
  /* initialize array */
  upc_forall(i=0;i<ARRSIZE;i++;i)
  {
     arr1[i]=i;
     arr2[i]=-i;
  }

  upc_barrier;

  /* test 1 */
  /* cast a pointer-to-shared with block size 3 to a pointer-to-shared with block
     size 1
  */

  /*p1 is a pointer-to-shared with block size 1, and arr1 is a shared array with
    block size 3. When p1 is assigned with the address of arr1[5], it involves
    a cast from a pointer-to-shared with block size 3 to a pointer-to-shared with
    block size 1 as follows:

    p1=(shared int *)&arr1[5];

    To find out which element p1 will point to after the cast, you can take the
    following steps:

      1 Re-layout pointer p1.
      2 Locate the same position (which has the same thread and offset
        with arr1[5]).
      3 Check the phase of that location under block size 1 data layout.
        If the phase of that position is 0, p1 will point to the same position
        where arr1[5] stays. If the phase of that position is not 0, p1 will
        point to the location with phase 0 within the same block. (When two
        pointers-to-shared with different block size are cast, the resulting
        pointer will always have a 0 phase.)
  */
  /* T0      T1       T2       T3
     0       3        6        9    // arr1 is a shared array with block size 3
     1       4        7        10
     2       5        8        11   // arr1[5]: thread=1, phase=2, value=5
     12      15       18       21
     13      16       19       22
     14      17       20       23
     ....


     0       1        2        3    // p1 points to a shared array with block size 1
     4       5        6        7
     8       9        10       11   /*After assignment, p1 still points to the
                                      arr1[5], as position 9 in the data layout
                                      of block size 1 has the same offset, the
                                      same thread with arr1[5], and has a zero
                                      phase.
```

```
                                    */
     12      13       14       15
     16      17       18       19
     20      21       22       23
     ....
  */
  p1=(shared int *)&arr1[5];

  verify(upc_phaseof(p1),0);        // Phase of p1 must be zero after the cast

  verify(upc_threadof(p1),upc_threadof(&arr1[5])); // keep the same thread

  verify(*p1,5);      // p1 still points to arr1[5]

  verify(*(p1-2),10); /* Move backward 2 elements based on the data layout of block
                         size 1, and p1 will point to arr1[10].
                      */
  verify(*(p1+2),11); /* Move forward 2 elements based on the data layout of block
                         size 1, and p1 will point to arr1[11].
                      */

  /* test 2 */
  //cast a pointer-to-shared with block size 1 to a pointer-to-shared with block size 3
  /* T0     T1      T2       T3
     0      1       2        3      // arr2 is a shared array with block size 1
     4      5       6        7      // arr2[5]:thread=1, phase=0
     8      9       10       11
     12     13      14       15
     16     17      18       19
     20     21      22       23
     . . .

     0      3       6        9     // p2 points to a shared array with block size 3
     1      4       7        10    /* after assignment, p2 will point to position 3
                                      (arr2[1]) in order to keep the same thread,
                                      same offset with arr2[5] and ensure phase is
                                      also zero.
                                   */
     2      5       8        11
     12     15      18       21
     13     16      19       22
     14     17      20       23
     . . .
  */

  p2=(shared[BLKSIZE] int *)&arr2[5];

  verify(*p2,-1);                                  // p2 will point to arr2[1]

  verify(upc_phaseof(p2),0);                       // phase has been reset to 0

  verify(upc_threadof(p2),upc_threadof(&arr2[5])); // keep the same thread

  /*Move forward according to new data layout with block size 3,
    and it will point to arr2[5]
  */
  verify(*(++p2),-5);

  verify(*(p2-2),-8); // Then move backward 2 elements, and it will point to arr2[8].
  return SUCCESS;
}
```

Another example is given to demonstrate the casting between a pointer-to-shared with a definite block size and another pointer-to-shared with an indefinite block size.

```
# include <upc.h>

# define BLKSIZE  3
# define ARRSIZE  60
# define FAILURE  166
# define SUCCESS  155

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  {   \
      printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
  }   \
}

shared[] int *p1,arr2[ARRSIZE];
shared[BLKSIZE] int *p2,arr1[ARRSIZE];

int main()
{
   int i;

   /* initialize array */
   upc_forall(i=0;i<ARRSIZE;i++;i)
   {
         arr1[i]=i;
         arr2[i]=-i;
   }

   upc_barrier;

   /* T0    T1    T2    T3
      0     3     6     9      // arr1 is a shared array with block size 3.
      1     4     7     10
      2     5     8     11     // arr1[5]: thread=1, offset=2*sizeof(int), phase=2
      12    15    18    21
      13    16    19    22
      14    17    20    23
      ...


   p1 points to a shared array with indefinite block size. The thread is
   determined by the assignment which sets p1, so p1 will have the same
   thread as arr1[5]. The phase for a pointer-to-shared array with an
   indefinite block size is always zero, there is no need to reset phase.
   p1 will point to arr1[5].
      0
      1
      2
      3
      4
      5
      .
      .
      .
   */

   /* test 1 */
   /* cast a pointer-to-shared array with block size 3 to a pointer-to-shared with
      an indefinite block size
   */
   p1=(shared[] int *)&arr1[5];

   verify(upc_phaseof(p1),0); //Phase is reset to zero.

   verify(upc_threadof(p1),upc_threadof(&arr1[5]));  // Remain the same thread.
```

```
verify(*p1,5);

p1++; // Move forward 1 position, and it will point to arr1[15].

verify(upc_phaseof(p1),0);

verify(upc_threadof(p1),1); // Thread should remain same thread with arr1[5]

verify(*p1,15);

verify(*(p1-3),3); // Move backward 3 positions, and it will point to arr1[3].

/* test 2 */
/* Cast a pointer-to-shared with an indefinite block size to a pointer-to-shared with
   block size 3.
*/

/* T0    T1    T2    T3
   0
   1
   2
   3
   4
   5  // arr2[5]: thread=0, offset=5*sizeof(int), phase=0
   6
   7
   8
   9
   10
   11
   12
   13
   14
   15
   .
   .
   .


   0     3     6     9
   1     4     7     10
   2     5     8     11
   12    15    18    21
   13    16    19    22
   14    17    20    23  /*Position 14 has the same offset,same thread with
                           arr2[5],but phase of position 14 is not zero, after
                           reset phase, p2 will point to position 12 ( arr2[3]).
                         */

   24    27    30    33
   25    28    31    34
   26    29    32    35
   . . .

*/

p2=(shared[BLKSIZE] int *)&arr2[5];
verify(upc_phaseof(p2),0);  // phase is reset to 0
verify(upc_threadof(p2),0); // thread remains same with arr2[5]
verify(*p2,-3);             // p2 will point to arr2[3]

/*Move back THREADS positions. There are no elements that have affinity to T1,
  T2, T3. p2 will point to arr2[2].
*/
p2 -=10; // Move backward 10 positions from arr2[12] to arr2[2]
```

```
   verify(upc_phaseof(p2),2);
   verify(upc_threadof(p2),0);
   verify(*p2,-2);
   return SUCCESS;
}
```

However, if either the source or the destination pointer type is the generic pointer-to-shared, `shared void *`, the phase value is preserved in the resulting pointer. For example:

```
# include <upc.h>

shared void *p1;
shared[3] int arr[10],*p2;

int main()
{
   p2=&arr[2];
   p1=&arr[2]; // p1 has zero phase
   p2=p1;      // phase of p2 is preserverd

   printf("Th:%d,upc_phaseof(p1)=%d,upc_phaseof(p2)=%d\n",MYTHREAD,upc_phaseof(p1),

   upc_phaseof(p2));

   return 0;
}
```

The output is as follows:

```
Th:0,upc_phaseof(p1)=0,upc_phaseof(p2)=2
Th:1,upc_phaseof(p1)=0,upc_phaseof(p2)=2
```

If you cast a generic pointer-to-shared to a nongeneric pointer-to-shared, there are two situations:

- For the nongeneric pointer-to-shared type with an indefinite block size or block size 1, the result is a pointer with phase 0. For example:

  ```
  #include <upc.h> // assume 2 threads

  shared void *p1;
  shared[] int arr1[10],*p2;
  shared int arr2[10],*p3;

  int main()
  {
     p2=arr1; // p2 has phase 0
     p3=arr2; // p3 has phase 0
     p2=p1;   // generic pointer p1 is cast to non-generic pointer p2
     p3=p1;   // generic pointer p1 is cast to non-generic pointer p3
     printf("Th:%d,upc_phaseof(p2)=%d,upc_phaseof(p3)=%d\n",MYTHREAD,
     upc_phaseof(p2),upc_phaseof(p3));
     return 0;
  }
  ```

  The output is as follows:

  ```
  Th:0,upc_phaseof(p2)=0,upc_phaseof(p3)=0
  Th:1,upc_phaseof(p1)=0,upc_phaseof(p2)=0
  ```

- For the nongeneric pointer-to-shared type with a definite block size that is greater than one, the result is undefined if the phase value of the source pointer is beyond the range of the possible phases of the destination pointer type.

Cast of a pointer-to-shared to a pointer-to-private

If you cast a pointer-to-shared to a pointer-to-private, there are the following situations:

- If a pointer-to-shared is cast to a C pointer and the pointer-to-shared does not have affinity to the current thread, the result is undefined.
- If a null pointer-to-shared is cast to a pointer-to-private, the result is null pointer. For example:

```
shared int *p1=NULL;
int *p2;
p2=(int *)p1; // p2 will be a null pointer
```

- It is legal to cast a pointer-to-shared to a C pointer only when the pointer-to-shared has affinity to the current thread. For example:

```
# include <upc.h> // assume 2 threads

shared int *p1;
int *p2;
shared int arr[8]={0,1,2,3,4,5,6,7};

int main()
{
   p1=&arr[3];
   p2=(int *)p1;

   //dereferencing p1 under MYTHREAD will access arr[3]
   printf("Th: %d,*p1=%d\n",MYTHREAD,*p1);

   //dereferencing p2 under upc_threadof(p1) will access arr[3]
   if(MYTHREAD == upc_threadof(p1))
      printf("Th: %d,*p2=%d\n",MYTHREAD,*p2);
   return 0;
}
```

The output is as follows:

```
Th: 0,*p1=3
Th: 1,*p1=3
Th: 1,*p2=3
```

Cast of a pointer-to-private to a pointer-to-shared

A pointer-to-private cannot be cast to a pointer-to-shared. The exception is made only when the constant expression 0 is cast. In this case, the result is a null pointer-to-shared type. For example:

```
# include <upc.h>

int i;
shared int *p1;
int *p2=0;

void main ()
{
   p1=0;  // legal
   p1=p2; // illegal, compiler will report error
   p1=&i; // illegal, compiler will report error
}
```

Cast of a pointer-to-private to a pointer-to-private

A pointer-to-private in Unified Parallel C is equivalent to a normal pointer in C. All semantic rules applicable to pointer casts in C are also applicable to a pointer-to-private.

Declarations

Unified Parallel C extends the C language to support shared data declaration.

Type qualifiers

Unified Parallel C provides the following type qualifiers:

- The *Shared* type qualifier
- The reference type qualifier
- The layout qualifier

Shared type qualifiers

A `shared` type qualifier is used to declare data in the shared memory space.

The shared type qualifier syntax

```
►►──shared──┬────────────────────┬──type──identifier──;──────────────────────────►◄
            └──layout qualifier──┘
```

For example:

```
shared int i;                    // shared scalar variable
shared [10] int a[10*THREADS];   // shared array
shared int* p;                   // private pointer-to-shared
shared int* shared p;            // shared pointer-to-shared
```

A `shared` type qualifier can be used wherever a type qualifier can be applied. However, it cannot be used in the *specifier-qualifier-list* of a structure declaration unless the `shared` type qualifier qualifies the type pointed to by a pointer. For example:

```
struct S1
{
  shared int *p1;                // fine
  int *shared p2;                // error
  shared int *shared p3;         // error
  shared int a;                  // error
  shared int b[10];              // error
};
```

The `shared` type qualifier cannot be used to declare variables at local scope unless the declaration is *`static`* qualified. In addition, the `shared` type qualifier cannot be used to declare function arguments.

Reference type qualifiers

Unified Parallel C provides the following reference type qualifiers:

relaxed
: The `relaxed` reference type specifies the access as relaxed.

strict
: The `strict` reference type specifies the access as strict.

With no reference type qualifier, the reference type is determined by the Unified Parallel C directives.

The reference type qualifier cannot appear in a qualifier list unless the list also has a `shared` type qualifier. Shared accesses must be either `relaxed` or `strict`. The declaration specifiers cannot include both `relaxed` and `strict` at the same time, either directly or by one or more `typedefs`. For example:

```
shared relaxed int a;           // correct
relaxed int b;                  // incorrect
shared relaxed strict int c;    // incorrect

typedef relaxed shared int rsi;
rsi rsi_array[5*THREADS];       // correct
rsi strict rsi_array[3*THREADS]; // incorrect
```

Layout qualifiers

A layout qualifier specifies the number of consecutive array elements to be allocated in the shared memory space of the same thread.

The layout qualifier syntax

Different block sizes are set depending on the way that the layout qualifier is specified:

- If the optional constant expression is 0 or not specified in the square brackets [], it indicates an indefinite block size where all elements have affinity to the same thread.
- If there is no layout qualifier, the block size is 1 by default.
- If the layout qualifier has the form [*], the shared object is distributed as if it had a block size of `( sizeof(array) / upc_elemsizeof(array) + THREADS - 1 ) / THREADS`.

The layout qualifier is one of the factors that determines whether two qualifier types are compatible. However, generic pointers-to-shared are always treated as if they had a compatible block size in order to be assignment compatible. A generic pointer-to-shared has type `shared void *`. For example:

```
shared [2] int *p1;
shared [3] int *p2;
shared void *p3;

p1=p2;                  // illegal
p1=(shared[2] int *)p2; // legal
p3=p1;                  // legal
p2=p3;                  // legal
```

The layout qualifier `[*]` cannot be used in the declaration specifiers of a pointer type. In addition, a layout qualifier cannot be used in the type qualifiers for the referenced type in a pointer to void type. For example:

```
shared[*] int *p1;      // not allowed
shared[2] void *p2;     // not allowed
```

Related reference

- Declarators
- Unified Parallel C directives

Declarators

The syntax and semantics of declarators in C also apply to declarators in Unified Parallel C.

Note: All shared objects that are designated by nonarray static declarators have affinity to thread 0.

Restrictions

Type qualifier list cannot specify more than one layout qualifier, either directly or indirectly by one or more typedefs. For example:

```
typedef shared[3][5] int T;     // wrong
T a[20];                        // wrong
shared[3][5] int b[20];         // wrong
```

Type qualifier list cannot include both strict and relaxed at the same time, either directly or indirectly by one or more typedefs. For example:

```
typedef strict shared int T;
relaxed T a;                    // wrong
strict shared relaxed int b;    // wrong
```

The shared type qualifier cannot specify the following two kinds of data objects:

- Objects with automatic storage duration
- Elements of an array object with automatic storage duration

For example:

```
shared int func(shared int arg) /* wrong, function return type and function
                                   parameter cannot be shared type
                                */
{
  shared int x;                 // wrong
  shared int y[10];             // wrong
  static shared int z;          // correct
  shared int *shared p1;        // shared pointer can not be automatic variable
  shared int *p2;               // correct
  return x;
}
```

Array declarators

Shared array elements are allocated by blocks of elements across threads. The block size is determined by the layout qualifier.

For a shared array x, the ith element has affinity to the thread (floor (i/block size) mod THREADS), and upc_phaseof(&x) is 0.

For any shared array with a definite block size in the dynamic environment, the THREADS expression must be applied only once in one dimension of the array declarator. The THREADS expression must be either used alone, or used when it is multiplied by an integer constant expression. For example:

```
shared int x[THREADS][THREADS]; /* wrong, you can only apply the THREADS expression
                                   once in one dimension of the array declarator
                                */
```

```
shared int y[THREADS][4];       // correct
shared int z[THREADS*5][4];     // correct
shared int x [10+THREADS];      // not allowed in the dynamic environment
shared [] int x [THREADS];      // not allowed in the dynamic environment
shared int x [10];              // not allowed in the dynamic environment
```

Related reference

- "upc_phaseof" on page 59
- "Predefined identifiers" on page 17

Statements and blocks

This section contains information about synchronization statements and iteration statements.

Synchronization statements

This section describes the Unified Parallel C synchronization statements which include blocking barriers, nonblocking barriers, and fences.

A barrier statement can be used to ensure that all threads reach the same program execution point before any of them proceeds further in the execution of the program. Typically, a barrier statement is used to ensure that modifications of a shared object done by any thread are visible to all threads after the barrier.

Unified Parallel C provides the following synchronization statements:
- Blocking barriers
- Nonblocking barriers
- Fences

Blocking barriers

The blocking barrier is invoked by calling `upc_barrier`. The reason that it is called the blocking barrier is because all threads are blocked at the point where `upc_barrier` is invoked. After reaching a barrier, a thread can proceed further in the program execution only after all other threads have executed the barrier.

The blocking barrier statement syntax

```
▶▶──upc_barrier──┬────────────┬──;──────────────────────────────────▶◀
                 └─expression─┘
```

`upc_barrier` *`expression`* has the same effect as the following compound statement:

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

Where `barrier_value` is the result of evaluating *expression*. If no *expression* is present, `barrier_value` can be omitted in the compound statement.

All expressions in blocking barrier statements must be of type `int`.

Example

This example demonstrates the usage of the blocking barrier.

```
# include <upc.h> // assume 4 threads
# include <stdio.h>

shared int a;

int main()
{
   if(MYTHREAD == 0)
      a=-1;

   /*threads are blocked until all threads reach the barrier and execute the
     upc_barrier statement
   */
   upc_barrier;

   // all threads must see the updated value of a
   printf("Th=%d,a=%d\n",MYTHREAD,a);

   return 0;
}
```

This program produces the output as follows:

```
/*The sequence of output is not guaranteed.*/
Th=0,a=-1
Th=3,a=-1
Th=2,a=-1
Th=1,a=-1
```

In this example, you must insert a `upc_barrier` before printing the value of the shared variable *a*. The barrier guarantees that all threads print the same value of *a*.

Nonblocking barriers

The nonblocking barrier, also called the split-phase barrier, consists of `upc_notify` and `upc_wait`. The reason that it is called the nonblocking barrier is because threads are not blocked at `upc_notify`. The thread that calls `upc_notify` can continue to do other local work between `upc_notify` and `upc_wait`. The thread is blocked only when it executes `upc_wait`. After all threads execute `upc_notify`, the thread that is blocked can continue its execution.

Each thread must execute `upc_notify` and `upc_wait` statements alternately, beginning from the `upc_notify` and ending at the `upc_wait` statement. For a given thread, after the execution of a `upc_notify` statement, the next collective operation to perform must be a `upc_wait` statement. A synchronization phase contains the executions of all statements from the completion of one `upc_wait` to the start of the next `upc_wait`.

The notify statement syntax

```
►►──upc_notify──┬────────────┬──;───────────────────────────────────►◄
                └─expression─┘
```

The wait statement syntax

```
►►──upc_wait──┬────────────┬──;─────────────────────────────────────►◄
              └─expression─┘
```

After executing the upc_wait statement, the thread does not enter the next synchronization phase until all threads have completed the upc_notify statement in the current synchronization phase.

A upc_notify or a upc_wait statement can have an optional expression. A upc_wait statement interrupts the execution of the program if the expression is different from the expression of the upc_notify statement executed by any thread in the current synchronization phase. The behavior after such an interruption is undefined. When either one of the upc_wait and upc_notify statements is missing this expression, the two statements are still considered to have the same expression. For example:

```
upc_notify 1;
//do some work
upc_wait 1;      // legal

upc_notify 1;
//do some work
upc_wait 2;      // illegal, mismatched expression

upc_notify 1;
//do some work
upc_wait;        // legal, considered as upc_wait 1;
```

All expressions in nonblocking barrier statements must be of type int.

Example

The following example demonstrates the usage of the nonblocking barrier.

```
# include <upc.h> // assume 4 threads
# include <stdio.h>

# define ARRSIZE 10

shared int a;
int b=0;

int main()
{
   int i;

   if(MYTHREAD == 0)
      a=-1;
   upc_notify;   /* The thread executing the upc_notify statement will
                    notify other threads that it reaches this program
                    point.
                 */

   b +=MYTHREAD; /*threads are not blocked and can continue to do other
                   computation
                 */

   upc_wait ;    // wait until all threads have called upc_notify

   printf("Th=%d,a=%d,b=%d\n",MYTHREAD,a,b); /* all threads must see the
                                                updated value of a and
                                                their own copy of b
                                             */

   return 0;
}
```

This program produces the output as follows:

```
//The sequence of the output is not guaranteed
Th=3,a=-1,b=3
Th=0,a=-1,b=0
Th=1,a=-1,b=1
Th=2,a=-1,b=2
```

Unlike `upc_barrier`, `upc_notify` does not block the execution of a program. By using a nonblocking barrier, you can insert or schedule independent computation to overlap the latency of the synchronization operation.

Related reference

Barriers

Fences

The `upc_fence` statement ensures that all shared accesses issued by the calling thread before the fence are completed. Unlike `upc_barrier` which synchronizes shared accesses for all threads, `upc_fence` synchronizes shared accesses only for the calling thread.

The fence statement syntax

```
►►──upc_fence──;──────────────────────────────────────────────────────►◄
```

`upc_fence` imposes an order between shared accesses. It is a noncollective operation that is much faster than `upc_barrier`.

Example

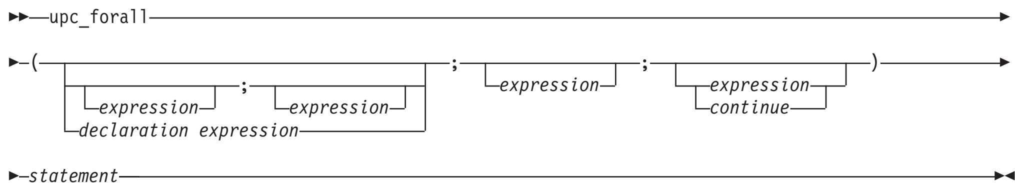
```
# include <upc.h>

# define SUCCESS 155
# define FAILURE 166
# define GENRANDOMTHREADNUM ( random()%THREADS )

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

shared int A[THREADS];

int main()
{
   int i=0;

   // change the sequence of thread execution
   if(MYTHREAD == GENRANDOMTHREADNUM)
      for(i=0;i<10000;i++);
   A[MYTHREAD]=MYTHREAD;

   /*upc_fence here must guarantee that all shared accesses are propagated to
     memory for the issuing thread
   */
   upc_fence;

   verify(A[MYTHREAD],MYTHREAD);
```

```
   // change shared data value when issuing thread is thread 0
   if(MYTHREAD == 0)
      A[MYTHREAD]=99;

   /*Having executed the above statements, thread 0 should see the updated shared
     value in the memory after it executes the following upc_fence
   */
   upc_fence;

   if(MYTHREAD == 0)
      verify(A[MYTHREAD],99);

   return SUCCESS;
}
```

Iteration statements

This section provides information about the syntax and usage of the work sharing construct `upc_forall`.

The upc_forall syntax



The `upc_forall` statement is used to distribute tasks among multiple threads. It is similar to the `for` statement except that the `upc_forall` statement has an optional fourth expression (*affinity*). The *affinity* expression, if present, is used to determine which thread executes a given loop iteration.

Each thread evaluates the first three expressions of the `upc_forall` statement following the semantics of the corresponding expressions of the `for` statement in the C language. In addition, each thread evaluates the affinity expression at every loop iteration.

The *affinity* expression determines which thread to execute a given loop iteration according to the following rules:

- If *affinity* is an integer expression, MYTHREAD executes each iteration of the loop body only if MYTHREAD == *affinity* % THREADS.
- If *affinity* is an expression of pointer-to-shared type, MYTHREAD executes each iteration of the loop body only if MYTHREAD == `upc_threadof(`*`affinity`*`)`.
- If *affinity* is *`continue`* or unspecified, each thread executes the loop body of the `upc_forall` statement.

A `upc_forall` loop can be nested inside one or more `upc_forall` loops, either directly or indirectly through a function call. In a nested `upc_forall` statement, the `controlling upc_forall` statement is the outermost `upc_forall` statement with an affinity expression that is not `continue`. Except for the `controlling upc_forall` statement, all `upc_forall` statements behave as if their affinity expressions were `continue`.

The result is undefined if any of the following situations is true:

- Any thread stops or performs a collective operation within the dynamic scope of the `upc_forall` statement.
- Any thread stops the `upc_forall` statement by using the `break`, `goto`, `return` statements, or the `longjmp` function.
- Any thread enters the body of the `upc_forall` statement by using the `goto` statement.
- The execution of a loop iteration affects the execution of another loop iteration by another thread

Example

This example demonstrates how the `upc_forall` loop is used to update three components of the position, velocity, and acceleration of each particle in parallel.

```
# include <upc.h>

# define NPARTS 1000*THREADS
# define BF NPARTS/THREADS
# define POSITION 10

typedef struct t
{
   double p[POSITION];
   double v[POSITION];
   double a[POSITION];
   double f[POSITION];
} particle_t;

shared[BF] particle_t PARTS[NPARTS]; // lattice of NPART particles

void update_particle_position()
{
   int i,j;
   double rmass=5.0,dt=1.0;
   /*parallel computation: compute new particles position, velocity, acceleration
     (on each dimension), given the current particles position, velocity,
     acceleration and the force acting on them.
   */
   upc_forall(i = 0; i < NPARTS; i++; &PARTS[i])
   {
      for(j=0;j<POSITION;j++)
      {
         PARTS[i].p[j] += PARTS[i].v[j]*dt + 0.5*dt*dt*PARTS[i].a[j];
         PARTS[i].v[j] += 0.5*dt*(PARTS[i].f[j]*rmass + PARTS[i].a[j]);
         PARTS[i].a[j] = PARTS[i].f[j]*rmass;
      }
   }
}
```

Predefined macros and directives

This section describes the macros predefined and the directives supported by the XL Unified Parallel C compiler.

- Unified Parallel C directives
- “Predefined macros” on page 48

Unified Parallel C directives

Unified Parallel C supports the following directives:

`#pragma upc strict`
Controls the memory consistency model followed by the program. It

informs the compiler that shared accesses follow the `strict` memory model. The directive can be overridden by declaring a shared variable using the `relaxed` declaration qualifier.

#pragma upc relaxed
: Controls the memory consistency model followed by the program. It informs the compiler that shared accesses follow the `relaxed` memory model. The directive can be overridden by declaring a shared variable using the `strict` declaration qualifier.

No macro substitution takes place if the preprocessing token `upc` follows the directive immediately.

The directives can be used either outside external declarations, or before all statements and explicit declarations that are inside a compound statement. The directives have different scopes in the following situations:

- When the directives are used outside external declarations, the scope of the directives applies until the appearance of the same directive or the end of the translation unit.
- When the directives are used inside a compound statement, the scope of the directives applies until the end of the compound statement. At the end of the compound statement, the state of the directive is restored to the state preceding the compound statement.

Note: If the directives are used in any other context, the behavior is undefined.

Predefined macros

The XL Unified Parallel C compiler predefines the following macros:

UPC
: The integer constant 1 that represents a conforming implementation.

_UPC_VERSION_
: The integer constant 200505L.

UPC_MAX_BLOCK_SIZE
: The integer constant that represents the maximum permissible value in a layout qualifier for shared data.

The conditional implementation-defined macros are listed as follows:

_UPC_DYNAMIC_THREADS_
: The integer constant 1 in the dynamic *THREADS* environment. Otherwise, it is undefined.

_UPC_STATIC_THREADS_
: The integer constant 1 in the static *THREADS* environment. Otherwise, it is undefined.

Chapter 5. Unified Parallel C library functions

The following sections provide information about the syntax and descriptions of the Unified Parallel C library functions.

Unified Parallel C provides the following standard header files:
- `upc_strict.h`
- `upc_relaxed.h`
- `upc_collective.h`
- `upc.h`

The inclusion of `upc_strict.h` or `upc_relaxed.h` has the effect of setting the default memory consistency mode for the entire compilation unit.

Utility functions

The Unified Parallel C utility functions are declared in header file `upc.h`. Include `upc.h` whenever using a Unified Parallel C utility function.

The Unified Parallel C utility functions can be classified into the following categories:
- Program termination
- Dynamic memory allocation
- Pointer-to-shared manipulation
- Serialization
- Memory transfer

Program termination

Unified Parallel C provides the following program termination function:

upc_global_exit

upc_global_exit

Frees all storages, flushes all I/O, and ends program execution for all active threads.

Prototype

void upc_global_exit(int status);

Parameters

status
: Represents the exit status code.

Example

```
//do some work
..
..
upc_global_exit(130); /*first thread which invokes this function will terminate
```

© Copyright IBM Corp. 2010

```
                            program execution for all active threads
                          */

printf("Error: no thread should reach here\n");
```

Dynamic memory allocation

This section describes the dynamic memory allocation functions in Unified Parallel C

To dynamically allocate shared memory, Unified Parallel C provides functions that are collective or noncollective, global or local. These functions are described as follows:

- **collective**: The function is called by all threads, and each calling thread receieves the same return value.
- **noncollective**: The function is called by one thread. If the function is called by different threads, each calling thread receives different return values.
- **global**: The function allocates shared memory across all threads.
- **local**: The function allocates shared memory with affinity to the calling thread.

The following memory allocation utilities are available in Unified Parallel C to dynamically allocate shared memory:

- upc_all_alloc
- upc_global_alloc
- upc_alloc
- upc_free

upc_all_alloc

A collective function used to allocate shared memory distributed on all threads.

Prototype

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

Parameters

nblocks
Represents the number of the memory blocks.

nbytes
Represents the block size measured in bytes.

Return value

Returns the same generic pointer-to-shared on all threads. The return value is a null pointer-to-shared, when:

- `nblocks*nbytes` is 0.
- The required memory fails to be allocated.

For example:

```
typedef float T;
shared T *p;

p=(shared T *)upc_all_alloc(THREADS,0*sizeof(T));// p == NULL
p=(shared T *)upc_all_alloc(0,1*sizeof(T));      // p == NULL
p=(shared T *)upc_all_alloc(0,0*sizeof(T));      // p == NULL
```

Usage

upc_all_alloc is a collective function with single-valued parameters that allocates shared memory equivalent to the following declaration:

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

Note: The dynamic lifetime of an allocated object begins when a thread receives a pointer to the shared memory allocated, and ends when any thread deallocates the object.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

# define BLKS 10
# define BLKSIZE 2
typedef struct t
{
  int *pi;
} st;

typedef st T;
shared[BLKSIZE] T *pt;

int main()
{
   int i=0;

   //pt points to the dynamically allocated shared memory
   pt=(shared[BLKSIZE] T *)upc_all_alloc(BLKS,BLKSIZE*sizeof(T));
   if(pt != NULL)
   {
      //initialize shared data
      upc_forall(i=0;i<BLKS*BLKSIZE;i++;pt+i)
      {
          //allocate private space when MYTHREAD == upc_threadof(pt+i)
          (pt+i)->pi=malloc(1*sizeof(int));
          *((pt+i)->pi)=i;
      }
      upc_barrier;

      // verify results
      for(i=0;i<BLKS*BLKSIZE;i++)
      {
        if(MYTHREAD == upc_threadof(&pt[i]))

        // it makes sense to dereference pi when MYTHREAD == upc_threadof(&pt[i])
        verify(*(pt[i].pi),i);
      }
      upc_barrier;

      // free allocated space
      for(i=0;i<BLKS*BLKSIZE;i++)
```

```
         if(MYTHREAD == upc_threadof(&pt[i]))
             free(pt[i].pi);
      if(MYTHREAD == 0)
        upc_free(pt);
   }
   return 0;
}
```

upc_alloc

Allocates shared memory space with affinity to the calling thread.

Prototype

shared void *upc_alloc(size_t nbytes);

Parameters

nbytes
Represents the size of the allocated shared space measured in bytes.

Return value

The return value is a generic pointer-to-shared. It is a null pointer-to-shared, when:

- `nbytes` is 0.
- The required memory fails to be allocated.

Usage

It is a noncollective function. The `upc_alloc` function is similar to the `malloc()` function. One of the differences between these two functions is that `upc_alloc` returns a pointer-to-shared value.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n", __LINE__,
    MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

# define BLKS 10
# define BLKSIZE 0
typedef struct t
{
  int *pi;
} st;

typedef st T;
shared[BLKSIZE] T *shared pt; // declare a shared pointer-to-shared

int main()
{
   int i=0;
   if(MYTHREAD == THREADS-1)
   /*There is only one instance of pt which is located in thread 0's space, and it
```

```
    points to dynamically allocated memory space. Shared space is located in
    THREADS-1's space
  */
  pt=(shared[BLKSIZE] T *shared)upc_alloc(BLKS*sizeof(T));

  upc_barrier;

  if(pt != NULL)
  {
     // initialize shared data
     upc_forall(i=0;i<BLKS;i++;pt+i)
     {
         // note:malloc only happens in THREADS-1's private space
         (pt+i)->pi=malloc(1*sizeof(int));
         *((pt+i)->pi)=i;
     }
     upc_barrier;

     // verify results
     for(i=0;i<BLKS;i++)
     {
        if(MYTHREAD == THREADS-1)
           verify(*(pt[i].pi),i); /*it makes sense to dereference pi for
                                    THREADS -1, but dereferecing pi for
                                    other threads produces undefined results
                                  */
     }
  upc_barrier;

  // free allocated space
  for(i=0;i<BLKS;i++)
  {
    if(MYTHREAD == THREADS-1)
      free(pt[i].pi);
  }

  if(MYTHREAD == 0)
     upc_free(pt);

  }

  return 0;
}
```

upc_free

Releases dynamically allocated shared memory space.

Prototype

void upc_free(shared void *ptr);

Parameters

ptr Points to the dynamically allocated shared memory space to be freed.

Usage

If `ptr` is a null pointer, the function performs no action. Otherwise, the behavior is undefined unless both the following two conditions are true:

- The argument matches a pointer previously returned by the `upc_alloc`, `upc_global_alloc`, or `upc_all_alloc` function.
- The memory pointed to by `ptr` has not been deallocated yet by any thread.

Example

```
# include <upc.h>

# define  SUCCESS    155
# define  FAILURE    166
# define  LOOPCOUNT  10
# define  ARR_SIZE   100

typedef struct a
{
    char a[ARR_SIZE];
} my_a;

typedef struct b
{
    my_a ma[ARR_SIZE];
} my_b;

shared my_b *p;

int main()
{
   int i=0,j=0,k=0;

   /*we do muliptle allocations and deallocation to make sure that we
     successfully free the space
   */

   // program allocated and there is no memory leak
   upc_forall(i=0;i<LOOPCOUNT;i++;)
   {
        upc_forall(j=0;j<LOOPCOUNT;j++;)
        {
             upc_forall(k=0;k<LOOPCOUNT;k++;)
             {
                 upc_barrier;

                 // allocate a chunk of shared memory across threads
                 p=(shared my_b *)upc_all_alloc(ARR_SIZE,1*sizeof(my_b));
                 upc_barrier;

                 // free shared space by one of threads
                 if(p != NULL)
                 {
                   if(MYTHREAD == 0)
                     upc_free(p);
                 }

                 upc_barrier;

                 // allocate another chunk of memory which has affinity to MYTHREAD
                 p=(shared my_b *)upc_alloc(ARR_SIZE*sizeof(my_b));

                 // every thread frees its allocated shared space
                 if(p != NULL)
                     upc_free(p);

                 upc_barrier;

                 // every thread allocates a chunk of shared space across threads
                 p=(shared my_b *)upc_global_alloc(ARR_SIZE,1*sizeof(my_b));

                 // every thread frees its allocated shared space
                 if(p != NULL)
                     upc_free(p);

                 upc_barrier;
```

```
            }
        }
    }

    return SUCCESS;

}
```

upc_global_alloc

A noncollective function used to allocate shared memory distributed on all threads.

Prototype

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

Parameters

nblocks
: Represents the number of the memory blocks.

nbytes
: Represents the block size measured in bytes.

Return value

The return value is a generic pointer-to-shared. It is a null pointer-to-shared, when either of the following situations is true:

- `nblocks*nbytes` is 0.
- The required memory fails to be allocated.

Usage

`upc_global_alloc` allocates shared memory equivalent to the following declaration:

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

Both `upc_global_alloc` and `upc_all_alloc` allocate shared memory space across all threads, but `upc_global_alloc` is a noncollective function. If `upc_global_alloc` is called by multiple threads, each calling thread receives different allocations.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

# define BLKS 20
# define BLKSIZE 4
typedef struct t
{
  int a;
  shared int *pi;              // a pointer to shared memember
```

```
} st;

typedef st T;
shared[BLKSIZE] T *shared pt; // declared a shared pointer-to-shared

int main()
{
   int i=0;

   if(MYTHREAD == THREADS-1)
   {
     // pt points to dynamically allocated shared memory
     pt=(shared[BLKSIZE] T *shared)upc_global_alloc(BLKS,BLKSIZE*sizeof(T));
   }
   upc_barrier;

   if(pt != NULL)
   {
      // initialize shared data
      upc_forall(i=0;i<BLKS*BLKSIZE;i++;pt+i)
      {
          pt[i].a=-i;
          /*pi points to dynamically allocated shared space which is located
            in the calling thread's space and is visible to all threads
          */
          (pt+i)->pi=(shared int *)upc_alloc(1*sizeof(int));
          *((pt+i)->pi)=i;
      }
      upc_barrier;

      // verify results
      for(i=0;i<BLKS*BLKSIZE;i++)
      {
         verify(pt[i].a,-i);
         verify(*(pt[i].pi),i);
      }
      upc_barrier;

      // free allocated space
      for(i=0;i<BLKS*BLKSIZE;i++)
      {
         if(MYTHREAD == upc_threadof(pt+i))
         {
           upc_free(pt[i].pi);
         }
      }

      upc_barrier;

      if(MYTHREAD == 0)
        upc_free(pt);
   }

   return 0;
}
```

Pointer-to-shared manipulation

Unified Parallel C provides the following functions to manipulate pointers-to-shared:

- upc_addrfield
- upc_affinitysize
- upc_phaseof
- upc_resetphase

- upc_threadof

upc_addrfield

Returns an implementation-defined value representing the local address of the object pointed to by the argument.

Prototype

size_t upc_addrfield(shared void *ptr);

Parameters

ptr Points to the shared object whose local address is to be obtained.

Example

```
# include <upc.h>

# define  SUCCESS    155
# define  FAILURE    166
# define  BLKSIZE    THREADS // assume THREADS = 4
# define  ARRSIZE    10*THREADS

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  {   \
      printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
  }   \
}

typedef struct t
{
    int a;
} myt;

typedef myt T;

shared[BLKSIZE] T *S1,*S2;      /* declare 2 pointers pointing to shared memory
                                   S1 and S2
                                */

shared[BLKSIZE] T arr[ARRSIZE]; // declare a shared array with block size THREADS
T *P1,*P2;                      // declared 2 private pointers

int main()
{
   int i=0;

   //initialize the shared array
   upc_forall(i=0;i<ARRSIZE;i++;&arr[i])
   {
       arr[i].a=MYTHREAD;
   }
   upc_barrier;

   /* S1 and S2 point to two distinct elements of the same shared array object which
      has affinity to thread 1
   */
   S1=&arr[0+THREADS];
   S2=&arr[3+THREADS];

   //assignment is legal when S1 and S2 have affinity to the same thread
   if(MYTHREAD == upc_threadof(S1) && MYTHREAD == upc_threadof(S2))
   {
```

```
        P1=(T*)S1;
        P2=(T*)S2;
    }

    upc_barrier;

    //verify results
    if(MYTHREAD == upc_threadof(S1) && MYTHREAD == upc_threadof(S2))
    {
         /*program should pass following check according to UPC spec1.2 6.4.2,
           upc_addrfield function returns implementation-defined local address
           pointed to by the pointer-to-shared argument
         */
         verify( upc_addrfield(S2) - upc_addrfield(S1), (P2 -P1)*sizeof(T));
         verify(P1->a,upc_threadof(&arr[0+THREADS]));
         verify(P2->a,upc_threadof(&arr[3+THREADS]));
         verify(S1->a,upc_threadof(&arr[0+THREADS]));
         verify(S2->a,upc_threadof(&arr[3+THREADS]));
    }

    return SUCCESS;

}
```

upc_affinitysize

Returns the size of the local portion of the data in a shared object with affinity to a given thread.

Prototype

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

Parameters

totalsize
: Represents the total size of the shared memory allocation measured in bytes.

nbytes
: Represents the block size measured in bytes.

threadid
: Represents the thread whose affinity size is to be evaluated. It is an integer value that ranges from 0 to THREADS-1.

Usage

For a dynamically allocated shared object, the `totalsize` argument must be `nbytes*nblocks`, where `nblocks` and `nbytes` are exactly the arguments of `upc_global_alloc` or `upc_all_alloc` when the memory space for the shared object is allocated.

For a statically allocated shared object declared as:

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

The `totalsize` argument must be *`s * sizeof (t)`*, and the `nbytes` argument must be *`b * sizeof (t)`*.

Note: If the block size is indefinite, `nbytes` must be 0.

Example

```
# include <upc.h> // assume 3 threads

# define SIZE sizeof(int)
# define GETBLKSIZE(X) \
( ( sizeof((X)) / upc_elemsizeof((X)) + THREADS - 1 ) / THREADS )

shared int arr1[10];
shared [2] int arr2[10];
shared [] int arr3[10];
shared [*] int arr4[10];

int main(void)
{
   upc_affinitysize(sizeof(arr1),1*sizeof(int),MYTHREAD);
   // Th0:4*SIZE, Th1:3*SIZE, Th2:3*SIZE

   upc_affinitysize(sizeof(arr2),2*sizeof(int),MYTHREAD);
   // Th0:4*SIZE, Th1:4*SIZE, Th2: 2*SIZE

   upc_affinitysize(sizeof(arr3),0*sizeof(int),MYTHREAD);
   // Th0:10*SIZE, Th1:0,Th2: 0

   upc_affinitysize(sizeof(arr4),GETBLKSIZE(arr4)*sizeof(int),MYTHREAD);
   // Th0:4*SIZE, Th1:4*SIZE, Th2: 2*SIZE

   // dynamically allocated shared space with totalsize 10*SIZE, block size 2
   upc_affinitysize(10*sizeof(int),2*sizeof(int),MYTHREAD);
   // Th0:4*SIZE, Th1:4*SIZE, Th2: 2*SIZE
}
```

upc_phaseof

Returns the position within a block of the shared object pointed to by the argument.

Prototype

    size_t upc_phaseof(shared void *ptr);

Parameters

ptr Points to the shared object whose phase is to be evaluated.

Return value

If `ptr` is a null pointer-to-shared, the return value is 0.

Example

```
// assume 3 threads
shared int arr1[10];    // phase=0 for all elements

shared[3] int arr2[10]; // phase=0 for arr2[0],arr2[3],arr2[6],arr2[9].
                        // phase=1 for arr2[1],arr2[4],arr2[7]
                        // phase=2 for arr2[2],arr2[5],arr2[8]

shared[] int arr3[10];  // phase=0 for all elements

shared[*] int arr4[10]; // phase=0 for arr4[0],arr4[4],arr4[8]
                        // phase=1 for arr4[1],arr4[5],arr4[9]
                        // phase=2 for arr4[2],arr4[6]
                        // phase=3 for arr4[3],arr4[7]

shared int *ptr=NULL;   // upc_phaseof(ptr)=0
```

upc_resetphase

Returns a copy of the given pointer with its phase set to zero.

Prototype

shared void *upc_resetphase(shared void *ptr);

Parameters

ptr The input pointer to be copied.

Example

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

# define  SUCCESS    155
# define  FAILURE    166
# define  BLKSIZE1   5
# define  BLKSIZE2   10
# define  ARRSIZE    100

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  {   \
      printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
  }   \
}
typedef struct t
{
    // structure contains pointer-to-shared member
    shared[BLKSIZE2] int *pi;
} myt;

// declare a shared structure array with block size BLKSIZE1
shared[BLKSIZE1] myt arrt[ARRSIZE];

// declare a shared integer array with block size BLKSIZE2
shared[BLKSIZE2] int arri[ARRSIZE];

// declared a pointer array pointing to shared data with block size BLKSIZE1
shared[BLKSIZE1] myt *p1[ARRSIZE];

// declare a pointer array pointing to shared data with block size BLKSIZE2
shared[BLKSIZE2] int *p2[ARRSIZE];

int main()
{
   int i,j,k;

   //initialize the shared array
   upc_forall(i=0;i<ARRSIZE;i++;&arrt[i])
        arrt[i].pi=arri;
   upc_barrier;

   //invoke upc_resetphase
   for(i=0;i<ARRSIZE;i++)
       for(j=0;j<ARRSIZE;j++)
       {
            p1[i]=upc_resetphase(arrt+i);
            p2[j]=upc_resetphase((arrt+i)->pi+j);
       }

   upc_barrier;
```

```
   /* The upc_resetphase function returns pointer to shared which
      is identical to its input except it has zero phase
   */
    for(i=0;i<ARRSIZE;i++)
   {
        // phase becomes zero
        verify(upc_phaseof(p1[i]),0);
        verify(upc_phaseof(p2[i]),0);
        // remain the same thread number
        verify(upc_threadof(p1[i]),upc_threadof(&arrt[i]));
        for(j=0;j<ARRSIZE;j++)
        {
            // remain the same thread number
            verify(upc_threadof(p2[j]),upc_threadof(&(arrt[i].pi[j])));
        }
   }
   // make sure that the phase value of the original shared array remains the same
   for(i=0;i<ARRSIZE/BLKSIZE1;i++)
      for(j=0;j<BLKSIZE1;j++)
          verify(upc_phaseof(arrt+j+i*BLKSIZE1),j);

   for(k=0;k<ARRSIZE;k++)
     for(i=0;i<ARRSIZE/BLKSIZE2;i++)
       for(j=0;j<BLKSIZE2;j++)
          verify(upc_phaseof(arrt[i].pi+j+i*BLKSIZE2),j);

   return SUCCESS;

}
```

upc_threadof

Returns the index of the thread that has affinity to the shared object pointed to by the argument.

Prototype

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

Parameters

ptr Points to the shared data that has affinity to any single thread.

Return value

If `ptr` is a null pointer-to-shared, the thread index to which `ptr` has affinity is 0.

Example

```
# include <upc.h>

# define ARR_SIZE 10
# define SUCCESS 155
# define FAILURE 166

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %l, expect= %l\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

shared int su[10][10];
```

```
shared int *shared sp;
shared int *p;

int main()
{
   int i=0,j=0,k=0;
   upc_forall(i=0;i<ARR_SIZE;i++;)
   {
      upc_forall(j=0;j<ARR_SIZE;j++;&su[i][j])
         su[i][j]=upc_threadof(&su[i][j]);
   }

   upc_barrier;

   for(i=0;i<ARR_SIZE;i++)
   {
       for(j=0;j<ARR_SIZE;j++)
       {
          verify(su[i][j],upc_threadof(&su[i][j]));
       }
   }
   if(MYTHREAD == 0)
     sp=(shared int *)upc_global_alloc(THREADS,1*sizeof(shared int));

   upc_barrier;
   if( sp != NULL)
   {
      verify(upc_threadof(&sp[MYTHREAD]),MYTHREAD);
      if(MYTHREAD == 0)
         upc_free(sp);
   }
   return SUCCESS;
}
```

Serialization

Unified Parallel C provides the following utility functions which can be used to serialize access to a critical section of the program:

- upc_all_lock_alloc
- upc_global_lock_alloc
- upc_lock
- upc_lock_attempt
- upc_unlock
- upc_lock_free

upc_lock_t type

`upc_lock_t` is the Unified Parallel C lock type, which is an opaque data type. An opaque type is a data type whose size, layout, or both are defined by implementation. Objects of the type `upc_lock_t` have two states: locked or unlocked.

If two pointers point to the same lock object, the two pointers are equal. If you apply `upc_phaseof()`, `upc_threadof()`, or `upc_addrfield()` to the pointers pointing to an object of `upc_lock_t` type, the result is undefined.

upc_all_lock_alloc

Dynamically allocates shared space for a lock object. The allocated lock is initialized to an *unlocked* state.

Prototype

upc_lock_t *upc_all_lock_alloc(void);

Return value

Returns a pointer to the dynamically allocated lock object.

Usage

`upc_all_lock_alloc` is a collective function used to allocate a shared lock. Each thread receives the same pointer value.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

# define ARRSIZE 10*THREADS
typedef struct t
{
  int a;
} my_t;

shared my_t st[ARRSIZE];
upc_lock_t *lock;

int main()
{
   int i;

   lock=(upc_lock_t *)upc_all_lock_alloc();
   if( lock != NULL)
   {
      upc_lock(lock);
      for(i=0;i<ARRSIZE;i++)
         (st[i].a)++;
      upc_unlock(lock);

      upc_barrier; // make sure that all threads finish increment

      for(i=0;i<ARRSIZE;i++)
          verify(st[i].a,THREADS);

      if(MYTHREAD == 0)
         upc_lock_free(lock);
    }

   return SUCCESS;
}
```

upc_global_lock_alloc

Dynamically allocates shared space for a lock object. The allocated lock is initialized to an *unlocked* state.

Prototype

upc_lock_t *upc_global_lock_alloc(void);

Return value

Returns a pointer to the dynamically allocated lock object.

Usage

`upc_global_lock_alloc` is a noncollective function used to allocate a shared lock. Each thread receives a pointer to a different lock.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155
# define N 100000
shared int A[N];
shared int sum=0;

upc_lock_t *shared lock;
int main()
{
  int i,rc=SUCCESS;

  // intialize shared data
  upc_forall(i=0;i<N;i++;&A[i])
      A[i]=1;

  if(MYTHREAD == 0)
     lock=upc_global_lock_alloc(); //create a lock

  upc_barrier;

  if(lock != NULL)
  {
     upc_forall(i=0;i<N;i++;&A[i])
     {
         upc_lock(lock);
         sum +=A[i];
         upc_unlock(lock);
     }
     upc_barrier;

     // verify results
     if(MYTHREAD == 0)
     {
        if(sum != N)
        {
           printf("Th:%d,%d\n",MYTHREAD,sum);
           rc=FAILURE;
        }
     }
     upc_barrier

     // free space
     if(MYTHREAD == 0)
```

```
      upc_free(lock);
  }
  return rc;
}
```

upc_lock

Sets the state of the lock object pointed to by the argument to the *locked* state.

Prototype

void upc_lock(upc_lock_t *ptr);

Parameters

ptr Is a pointer that points to a lock object of type `upc_lock_t`.

Usage

If the lock pointed to by `ptr` is in unlocked state, a call to this function by a single thread sets the state of the lock to locked.

If the lock is already in locked state, then the calling thread waits for some other thread to set the state of the lock to unlocked before proceeding.

If the lock is already set as locked by the calling thread itself, the result is undefined.

Example

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

# define  SUCCESS    155
# define  FAILURE    166
# define  COUNT      10000

# define verify(result,expect) \
{ if ((result)!=(expect)) \
   {   \
      printf("Error: fail at line %d: mythread=%d, result= %l, expect= %l\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
   }   \
}

shared long sum=0;

/*declare a private pointer-to-shared that points to a shared opaque datatype
  upc_lock_t
*/
upc_lock_t *lock;

int main()
{
   int i;

   /* every thread's lock pointer points to the same dynamically allocated
      shared space
   */
   lock=upc_all_lock_alloc();

   if(lock != NULL)
   {
       for(i=0;i<COUNT;i++)
```

```
        {
            /* upc_lock ensures that the thread which successfully retrives the lock
               and sets the lock state to locked has the priviledge to execute the
               sum += i; statement while other threads have to wait until this thread
               releases the lock by calling the upc_unlock function. It avoids race
               condition when multiple threads try to read and write the same shared
               variable at the same time
            */
            upc_lock(lock);
            sum +=i;

            // upc_unlock sets the state of the lock pointed to by lock to unlocked
            upc_unlock(lock);
        }
        upc_barrier;

        // verify results
        verify(sum,(COUNT-1)*COUNT/2*THREADS);

        upc_barrier;

        /*deallocate the shared space associated with the dynamically allocated
          upc_lock_t pointed to by lock
        */
        if(MYTHREAD == 0)
           upc_lock_free(lock);
    }

    return SUCCESS;

}
```

upc_lock_attempt

Attempts to set the state of the lock object pointed to by the argument to the *locked* state.

Prototype

int upc_lock_attempt(upc_lock_t *ptr);

Parameters

ptr Is a pointer that points to a lock object of type `upc_lock_t`.

Return value

The return value is 1 if the function succeeds in setting the state of the lock to locked. Otherwise, the return value is 0.

Usage

If the lock is in unlocked state, the thread which calls `upc_lock_attempt` sets the state of the lock to locked, and the function returns 1.

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

If the lock has already been set to locked by the calling thread, the result is undefined.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155

shared int sum=0;
upc_lock_t *lock;

int main()
{
   int i,rc=SUCCESS;

   lock=upc_all_lock_alloc();                // dynamically allocate lock

   if(lock != NULL)
   {
      while(upc_lock_attempt(lock) == 0) {}; /* only proceed when thread
                                                successfully retrieves lock
                                             */
      sum += MYTHREAD;
      upc_unlock(lock);

      upc_barrier;

      // verify result
      if(MYTHREAD == 0)
        if(sum != (THREADS-1)*THREADS/2)
        {
           printf("Th: %d,result: %d, expect: %d\n",MYTHREAD,sum, (THREADS-1)
           *THREADS/2);
           rc=FAILURE;
         }

     // free space
     if(MYTHREAD == 0)
        upc_lock_free(lock);
   }

   return rc;
}
```

upc_unlock

Sets the state of the lock object pointed to by the argument to the *unlocked* state.

Prototype

void upc_unlock(upc_lock_t *ptr);

Parameters

ptr Is a pointer that points to a lock object of type `upc_lock_t`.

Usage

The result is undefined when either of the following situations is true:

- The lock is already in an unlocked state.
- The calling thread is not the locking thread.

Example

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

```
# define  SUCCESS    155
# define  FAILURE    166
# define  COUNT      1000

/*declare a shared pointer-to-shared which points to a shared opaque datatype
  upc_lock_t
*/
upc_lock_t *shared lock;

int main()
{
   int i,j;

   // pointer lock points to the dynamically allocated shared space
   if(MYTHREAD == THREADS -1)
       lock=upc_global_lock_alloc();

   upc_barrier;
   // if the allocation is successful
   if(lock != NULL)
   {
       for(i=0;i<COUNT;i++)
       {
           for(j=0;j<COUNT;j++)
           {
               // verify if thread can successfully retrive and release lock
               upc_lock(lock);
               /*upc_unlock sets the state of the lock pointed to by lock to
                 unlocked
               */
               upc_unlock(lock);
           }
       }
       upc_barrier;

     /*deallocate the shared space associated with the dynamically allocated
       upc_lock_t pointed to by lock
     */
      if(MYTHREAD == 0)
          upc_lock_free(lock);
   }

   return SUCCESS;

}
```

upc_lock_free

Deallocates the dynamically allocated lock object pointed to by the argument.

Prototype

void upc_lock_free(upc_lock_t *ptr);

Parameters

ptr Is a pointer that points to a lock object of type `upc_lock_t`.

Usage

`upc_lock_free` deallocates the memory pointed to by the argument, regardless of whether the lock pointed to by the argument is in the locked or unlocked state.

If `ptr` is a null shared pointer, the function performs no action. Calling `upc_lock_free` has undefined effects when either of the following conditions is true:

- The argument of this function does not match a pointer that was previously returned by `upc_global_lock_alloc` or `upc_all_lock_alloc`.
- The dynamically allocated memory for `upc_lock_t` lock object has already been deallocated by any thread that called `upc_lock_free`.

Note: After the allocated memory for `upc_lock_t` lock object is freed, a subsequent invocation of a locking functions with `ptr` as argument has undefined behavior.

Example

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

# define  SUCCESS    155
# define  FAILURE    166
# define  COUNT      10000

// declare a pointer array which points to shared data type upc_lock_t
upc_lock_t *shared lock[COUNT];

int main()
{
   int i,j;

   /* make sure all shared upc_lock_t object allocated can be successfully
      deallocated by invoking the upc_lock_free function
   */
   for(i=0;i<COUNT;i++)
   {
        upc_barrier 1;

        // lock[i] points to the dynamically allocated shared space
        lock[i]=upc_all_lock_alloc();

        upc_barrier 2;

        upc_lock(lock[i]);

        // read and write the shared data here with lock guarded

        upc_unlock(lock[i]);

        upc_barrier 3;

        /*deallocate the shared space associated with the dynamically allocated
          upc_lock_t pointed to by lock[i]
        */
        if(lock[i] != NULL)
            if(MYTHREAD == 0)
                upc_lock_free(lock[i]);

        upc_barrier 4;

        // one of the threads allocates the share space
        if(MYTHREAD == THREADS -1)
             lock[i]=upc_global_lock_alloc();

        upc_barrier 5;

        upc_lock(lock[i]);

        // read and write the shared data here with lock guarded
```

```
        upc_unlock(lock[i]);

        upc_barrier 6;

        /*deallocate the shared space associated with the dynamically allocated
          upc_lock_t pointed to by lock[i]
        */
        if(lock[i] != NULL)
          if(MYTHREAD == 0)
            upc_lock_free(lock[i]);

   }

   return SUCCESS;
}
```

Memory transfer

Unified Parallel C provides the following functions to copy data to and from shared memory:
- upc_memcpy
- upc_memget
- upc_memput
- upc_memset

upc_memcpy

Copies data between shared objects.

Prototype

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

Parameters

dst Points to the shared object to which the data is to be copied.

src Points to the shared object from which the data is copied.

n Represents the size of the data in bytes to be copied.

Usage

`upc_memcpy` copies `n` bytes from a shared object having affinity with a thread to a shared object having affinity with the same or another thread. The effect is equivalent to copying the entire content from one shared array with the type `shared [] char[n]` to another shared array with the type `shared [] char[n]`.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155
# define ARRSIZE 20

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
```

```
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

typedef struct t
{
  int a;
} myt;

shared[] myt src[ARRSIZE];
shared[] myt *shared dst;

int main()
{
  int i;

  // dst points to dynamically allocated shared memory space
  if(MYTHREAD == THREADS -1)
     dst=(shared[] myt *shared)upc_alloc(ARRSIZE*sizeof(myt));

  // initialize src
  upc_forall(i=0;i<ARRSIZE;i++;&src[i])
      src[i].a=i;
  upc_barrier;

  // copy shared data from src to dst if(MYTHREAD == THREADS -1)
  upc_memcpy(dst,src,ARRSIZE*sizeof(myt));
  upc_barrier;
  for(i=0;i<ARRSIZE;i++)
      verify(dst[i].a,i);

  upc_barrier;

  if(MYTHREAD == 0)
     upc_free(dst);

  return SUCCESS;
}
```

upc_memget

Copies data from a shared object with affinity to any single thread to a private object on the calling thread.

Prototype

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

Parameters

dst Points to the private object to which the data is to be copied.

src Points to the memory space from which the data is copied.

n Represents the size of the data in bytes to be copied.

Usage

`upc_memget` copies `n` bytes from a shared object that has affinity to a single thread to a private object on the calling thread. The effect is equivalent to copying the entire content from a shared array with the type `shared [] char[n]` to a private array with the type `char[n]`.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155
# define ARRSIZE 20

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

typedef struct t
{
  int a;
  shared int *p; // a pointer to shared member
}myt;

shared[] myt *shared src;
myt dst[ARRSIZE];

int main()
{
  int i;

  /*src points to dynamically allocated shared memory space which has affinity
    to THREADS-1
  */
  if(MYTHREAD == THREADS -1)
  {
     src=(shared[] myt *shared)upc_alloc(ARRSIZE*sizeof(myt));
     for(i=0;i<ARRSIZE;i++)
     {
          src[i].a=-i;
         //structure member p points to dynamically allocated shared space
          src[i].p=(shared[] int *shared)upc_alloc(1*sizeof(int));
          *(src[i].p)=i;
     }
  }

  upc_barrier;

  // copy data from src to dst
  upc_memget(dst,src,ARRSIZE*sizeof(myt));

  upc_barrier;

  for(i=0;i<ARRSIZE;i++)
  {
       verify(dst[i].a,-i);
       verify(*(dst[i].p),i);
  }


  if(MYTHREAD == 0)
  {
      for(i=0;i<ARRSIZE;i++)
         upc_free(src[i].p);
      upc_free(src);
  }
  return SUCCESS;
}
```

upc_memput

Copies data from a private object on the calling thread to a shared object that has affinity to any single thread.

Prototype

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

Parameter

dst Points to the shared object to which the data is to be copied.

src Points to the memory space from which the data is copied.

n Represents the size of the data in bytes to be copied.

Usage

`upc_memput` copies `n` bytes from a private object on the calling thread to a shared object that has affinity to any single thread. The effect is equivalent to copying the entire content from a private array with type `char[n]` to a shared array with the type `shared [] char[n]`.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155
# define ARRSIZE 20

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__, MYTHREAD, result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

typedef struct t
{
   int a;
   int *p; // p points to private data
} myt;

shared[] myt *dst;
myt *src;

int main()
{
    int i;

    // every src pointer points to dynamically allocated private memory space
    src=malloc(ARRSIZE*sizeof(myt));
    for(i=0;i<ARRSIZE;i++)
    {
        src[i].a=-i;
        // structure member p points to dynamically allocated private memory space
        src[i].p=malloc(1*sizeof(int));
        *(src[i].p)=i+MYTHREAD;
    }

    /* every dst pointer points to its own dynamically allocated shared space which
       has affinity to MYTHREAD
```

```
    */
    dst=(shared[] myt *)upc_alloc(ARRSIZE*sizeof(myt));

    // copy data from src to dst
    upc_memput(dst,src,ARRSIZE*sizeof(myt));

    for(i=0;i<ARRSIZE;i++)
    {
        verify(dst[i].a,-i);
        verify(*(dst[i].p),i+MYTHREAD);
    }


    // free allocated space
    for(i=0;i<ARRSIZE;i++)
        free(src[i].p);
    free(src);
    upc_free(dst);
    return SUCCESS;
}
```

upc_memset

Copies a given value, which is converted to an unsigned char, to a shared object with affinity to any single thread.

Prototype

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

Parameters

dst Points to the shared object to which the value `c` is to be copied.

c A parameter of type `int` that is converted to an unsigned char to initialize the shared object.

n Represents the size of the data in bytes to be set with the value `c`.

Usage

`upc_memset` sets the value `c` to the `n` bytes of shared memory. The effect is equivalent to setting the value `c` to the entire content of a shared array with the type `shared [] char[n]`.

Example

```
# include <upc.h>

# define FAILURE 166
# define SUCCESS 155
# define ARRSIZE 40

# define verify(result,expect) \
{ if ((result)!=(expect)) \
  { \
    printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
    __LINE__,MYTHREAD,result,expect); \
    upc_global_exit(FAILURE); \
  } \
}

shared[] int *shared dst;

int main()
{
```

```
    int i;
    char result[ARRSIZE*sizeof(int)];

    if(MYTHREAD == THREADS -1)
      dst=(shared[] int *)upc_alloc(ARRSIZE*sizeof(int));
    upc_barrier;

    if(dst != NULL)
    {
       upc_memset(dst,0x12,ARRSIZE*sizeof(int));
       for(i=0;i<ARRSIZE;i++)
          upc_memget(result+i*sizeof(int),dst+i,sizeof(int));

       for(i=0;i<ARRSIZE*sizeof(int);i++)
          verify(result[i],0x12);
       upc_barrier;

       if(MYTHREAD == 0)
          upc_free(dst);
    }

    return SUCCESS;
}
```

Collective functions

The Unified Parallel C collective functions are declared in header file `upc_collective.h`. Include `upc_collective.h` whenever using a Unified Parallel C collective utility function.

Collective functions in Unified Parallel C can be classified as relocalization functions and computational functions. Collective functions cannot be called between the `upc_notify` statement and the corresponding `upc_wait` statement.

Synchronization options

The memory semantics of collective Unified Parallel C library functions control the way in which data is synchronized.

The integer type `upc_flag_t`, which is defined in the header file `upc.h`, controls data synchronization semantics of certain collective Unified Parallel C library functions. A function argument of type `upc_flag_t` has a value yielded by applying the operator bitwise inclusive OR (|) to the two operands, UPC_IN_*X*SYNC and UPC_OUT_*Y*SYNC. That is, the function argument of type `upc_flag_t` has the value (UPC_IN_*X*SYNC | UPC_OUT_*Y*SYNC), where the variables *X* and *Y* can be specified with `NO`, `MY`, or `All`.

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

Suppose that the function argument of type `upc_flag_t` has the value (UPC_IN_*X*SYNC | UPC_OUT_*Y*SYNC).

UPC_IN_*X*SYNC
Specifies one of the following synchronization mechanisms for input data:

- If *X* is `NO`, the function can begin to read, or write data when the first thread enters the collective function call. The function does not perform data synchronization.

- If *X* is `MY`, the function can begin to read or write data when the thread that has affinity to the data calls the collective function. The function must perform data synchronization for all threads except for the current thread.
- If *X* is `ALL`, the function can begin to read or write data only after all threads have called the collective function. The function must synchronize all data. With `UPC_IN_ALLSYNC`, the function ensures that all threads read the same value of the input data after all threads have called the function.

UPC_OUT_*Y*SYNC

Specifies one of the following synchronization mechanisms for output data:

- If *Y* is `NO`, the function can read, or write data until the last thread returns from the collective function call. The function does not perform data synchronization.
- If *Y* is `MY`, the function can return in a thread only after all reading and writing operations on the data with affinity to the thread are finished. It indicates that after a thread returns from the function call, the thread does not read any earlier values of the output data with affinity to that thread. With `UPC_OUT_MYSYNC`, the function ensures that, after a thread returns from the function call, the thread reads the latest value of the output data with affinity to that thread.
- If *Y* is `All`, the function call can return only after the reading and writing operations on data are finished. The function must synchronize all data. With `UPC_OUT_ALLSYNC`, the function ensures that the thread reads the latest value of the output data after a thread returns from the function call.

Note: Passing the following corresponding arguments has the same effect:

Arguments	Equivalent arguments
UPC_IN_*X*SYNC	UPC_IN_*X*SYNC | UPC_OUT_ALLSYNC
UPC_OUT_*y*SYNC	UPC_IN_ALLSYNC | UPC_OUT_*y*SYNC
0	UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC

Relocalization functions

Unified Parallel C provides the following functions to move data to and from regions of memory space:

- upc_all_broadcast
- upc_all_scatter
- upc_all_gather
- upc_all_gather_all
- upc_all_exchange
- upc_all_permute

upc_all_broadcast

Copies a block of shared memory with affinity to a single thread to a block of shared memory on each thread.

Prototype

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

Parameters

dst Points to the destination memory which the block of shared memory is copied to. This pointer points to a shared object that has affinity to thread 0.

src Points to the block of shared memory to be copied.

nbytes
Represents the block size measured in bytes.

flags
Controls the data synchronization semantics.

Usage

`upc_all_broadcast` copies `nbytes`-byte block of memory that has affinity to a single thread to a block of shared memory on each thread. `nbytes` must be greater than 0.

The effect is equivalent to copying the entire content of a shared array with the type `shared [] char[nbytes]` to another shared array with the type `shared [nbytes] char[nbytes*THREADS]`.

Example

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

# define NELEMS 4
# define FAILURE 166
# define SUCCESS 155

shared[] int A[NELEMS];              // src pointer
shared[NELEMS] int B[NELEMS*THREADS]; // dst pointer

int main()
{
     int i=0;
     // initialize A
     upc_forall(i=0;i<NELEMS; i++; A+i)
        A[i]=i;

     upc_all_broadcast(B,A,sizeof(int)*NELEMS,UPC_IN_ALLSYNC|UPC_OUT_NOSYNC);

     upc_barrier;

     // verify results
     for(i=0;i<NELEMS; i++)
     {
         if(B[MYTHREAD*NELEMS+i] !=i)
         {
            printf("Error: thread %d,B[%d]=%d,expect: %d\n",MYTHREAD,i,B[MYTHREAD*
            NELEMS+i],i);
            upc_global_exit(FAILURE); /*terminate all threads and force to exit
                                        program
                                      */
         }
     }
     return SUCCESS;
}
```

upc_all_scatter

Copies the *i*th block of an area of shared memory that has affinity to a single thread to a block of shared memory that has affinity to the *i*th thread.

Prototype

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

Parameters

dst Points to the destination memory to which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

src Points to the source memory from which the block of shared memory is copied. This pointer points to a block of shared memory from which a block of memory is copied.

nbytes
Represents the block size measured in bytes.

flags
Controls the data synchronization semantics.

Usage

The function treats:

- The pointer `src` as if it pointed to a shared memory area with the type `shared [] char[nbytes*THREADS]`.
- The pointer `dst` as if it pointed to a shared memory area with the type `shared [nbytes] char[nbytes*THREADS]`.

For each thread *i*, `upc_all_scatter` copies the *i*th `nbytes`-byte block pointed to by `src` to the `nbytes`-byte block pointed to by `dst` that has affinity to thread *i*. `nbytes` must be greater than 0.

Example

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

# define NELEMS 10
# define FAILURE 166
# define SUCCESS 155

shared[] int *shared A;              // src pointer
shared[NELEMS] int B[NELEMS*THREADS]; // dst pointer

int main()
{
     int i=0;

     // dynamically allocate shared memory with affinity to thread THREADS-1
     if(MYTHREAD == THREADS -1)
         A=(shared[] int *)upc_alloc(THREADS*NELEMS*sizeof(int));
     upc_barrier;

     // initialize A
     upc_forall(i=0;i<NELEMS*THREADS;i++;A+i)
         A[i]=i;

     upc_all_scatter(B,A,sizeof(int)*NELEMS,UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

```
    // verify results
    for(i=0;i<NELEMS*THREADS;i++)
       if(B[i] != i)
       {
          printf("Error: thread=%d,B[%d]=%d,expect=%d\n",MYTHREAD,i,B[i],i);
          upc_global_exit(FAILURE); /*terminate all threads and force to exit
                                      program
                                    */
       }

    return SUCCESS;
}
```

upc_all_gather

Copies a block of shared memory that has affinity to the *i*th thread to the *i*th block of a shared memory area that has affinity to a single thread.

Prototype

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

Parameters

dst Points to a block of shared memory to which a block of memory is copied.

src Points to the source memory from which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

nbytes
Represents the block size measured in bytes.

flags
Controls the data synchronization semantics.

Usage

The function treats:

- The pointer `src` as if it pointed to a `nbytes`-byte block on each thread, and it had the type `shared[nbytes] char[nbytes*THREADS]`.
- The pointer `dst` as if it pointed to the shared memory area with the type `shared [] char[nbytes * THREADS]`.

`nbytes` must be greater than 0. For each thread *i*, the function has the same effect as copying the `nbytes`-byte block that has affinity to thread *i* pointed to by `src` to the *i*th `nbytes`-byte block pointed to by `dst`.

Example

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

# define NELEMS 10
# define FAILURE 166
# define SUCCESS 155

shared[NELEMS] int A[NELEMS*THREADS]; // src pointer
shared[] int B[NELEMS*THREADS];       // dst pointer

int main()
{
```

```
    int i=0;
    // initialize A
    upc_forall(i=0;i<NELEMS*THREADS; i++; A+i)
        A[i]=i;

    upc_all_gather(B,A,sizeof(int)*NELEMS,UPC_IN_ALLSYNC|UPC_OUT_NOSYNC);

    upc_barrier;

    // verify results
    for(i=0;i<NELEMS*THREADS;i++)
       if(B[i] != i)
       {
          printf("Error: thread=%d,B[%d]=%d,expect=%d\n",MYTHREAD,i,B[i],i);
          upc_global_exit(FAILURE); /*terminate all threads and force to exit
                                      program
                                    */
       }

    return SUCCESS;
}
```

upc_all_gather_all

Copies a block of memory from one shared memory area that has affinity to the *i*th thread to the *i*th block of a shared memory area on each thread.

Prototype

> void upc_all_gather_all(shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);

Parameters

dst Points to the destination memory to which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

src Points to the source memory from which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

nbytes
Represents the block size measured in bytes.

flags
Controls the data synchronization semantics.

Usage

`nbytes` must be greater than 0. The function treats:

- The pointer `src` as if it pointed to the `nbytes`-byte shared memory area on each thread, and had the type `shared[nbytes] char[nbytes*THREADS]`.
- The pointer `dst` as if it pointed to a shared memory area with the type `shared[nbytes*THREADS] char[nbytes*THREADS*THREADS]`.

`upc_all_gather_all` has the same effect as copying the *i*th `nbytes`-byte block pointed to by `src` to the *i*th `nbytes`-byte block that is pointed to by `dst` on each thread.

Example

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

```
# define NELEMS 10
# define FAILURE 166
# define SUCCESS 155

shared[NELEMS] int A[NELEMS*THREADS];
shared[NELEMS*THREADS] int B[THREADS][NELEMS*THREADS];

int main()
{
   int i=0;
   // initialize A
   upc_forall(i=0;i<NELEMS*THREADS;i++;A+i)
       A[i]=i;

   upc_barrier; /*need to synchronize data before invoking the collective
                  function
                */

   upc_all_gather_all(B,A,sizeof(int)*NELEMS,UPC_IN_NOSYNC|UPC_OUT_NOSYNC);

   upc_barrier; /*need to synchronize data to make sure the copy operation is
                  done for all threads
                */

   // verify results
   for(i=0;i<NELEMS*THREADS;i++)
     if(B[MYTHREAD][i] != i)
     {
       printf("Error: thread=%d,B[%d][%d]=%d,expect=%d\n",MYTHREAD,MYTHREAD,i,
       B[MYTHREAD][i],i);
       upc_global_exit(FAILURE); //terminate all threads and force to exit program
     }
   return SUCCESS;
}
```

upc_all_exchange

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*.

Prototype

> void upc_all_exchange(shared void * restrict dst, shared const void * restrict src, size_t nbytes, upc_flag_t flags);

Parameters

dst Points to the destination memory which the block of shared memory is copied to. This pointer points to a shared object that has affinity to thread 0.

src Points to the source memory from which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

nbytes
Represents the block size measured in bytes.

flags
Controls the data synchronization semantics.

Usage

The function treats the pointers `src` and `dst` as if each of them pointed to a `nbytes*THREADS`-byte shared memory area on each thread, and each had the type `shared[nbytes*THREADS] char[nbytes*THREADS*THREADS]`.

For thread *i* and thread *j*, `upc_all_exchange` is equivalent to copying the *i*th `nbytes`-byte block that has affinity to thread *j* pointed to by `src` to the *j*th `nbytes`-byte block that has affinity to thread *i* pointed to by `dst`. `nbytes` must be greater than 0.

Example

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

# define NELEMS 3
# define FAILURE 166
# define SUCCESS 155

shared[NELEMS*THREADS] int A[THREADS][NELEMS*THREADS];
shared[NELEMS*THREADS] int B[THREADS][NELEMS*THREADS];

int main()
{
  int i=0,j=0;

  // initialize A
  for(i=0;i<NELEMS*THREADS;i++)
  A[MYTHREAD][i]=1;
  upc_barrier;

  upc_all_exchange(B,A,sizeof(int)*NELEMS,UPC_IN_NOSYNC|UPC_OUT_NOSYNC);
  upc_barrier;

  // verify results
  for(i=0;i<NELEMS*THREADS;i++)
  {
    if(B[MYTHREAD][i] != 1)
    {
      printf("Error: thread=%d,result=%d,expect=%d\n",MYTHREAD,B[MYTHREAD][i],1);
      upc_global_exit(FAILURE);
    }
  }

  return SUCCESS;
}
```

upc_all_permute

Copies a block of memory from a shared memory area that has affinity to the *i*th thread to a block of shared memory that has affinity to thread `perm[i]`.

Prototype

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

Parameter

dst Points to the destination memory to which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

src Points to the source memory from which the block of shared memory is copied. This pointer points to a shared object that has affinity to thread 0.

perm
Points to a shared object that has affinity to thread 0.

nbytes
Represents the block size measured in bytes.

flags
   Controls the data synchronization.

Usage

nbytes must be greater than 0. THREADS distinct values, such as 0,1,..., THREADS-1, must be included in perm[0..THREADS-1].

The function treats the pointers src and dst as if each of them pointed to a nbytes-byte shared memory area on each thread, and each had the type shared[nbytes] char[nbytes*THREADS].

The function has the same effect as copying the nbytes-byte block that has affinity to thread *i* pointed to by src to the nbytes-byte block that has affinity to the thread perm[i] pointed to by dst.

Example

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

# define NELEMS 10
# define FAILURE 166
# define SUCCESS 155

shared[NELEMS] int A[NELEMS*THREADS]; //src pointer
shared[NELEMS] int B[NELEMS*THREADS]; //dst pointer
shared int P[THREADS];
shared[] int *myB;

int main()
{
     int i=0,j=0;
     // initialize P
     P[MYTHREAD]=THREADS-1-MYTHREAD;
     // initialize A
     upc_forall(i=0;i<NELEMS*THREADS;i++;A+i)
         A[i]=i;
     upc_barrier;
     upc_all_permute(B,A,P,sizeof(int)*NELEMS,UPC_IN_NOSYNC|UPC_OUT_NOSYNC);

     upc_barrier;

     // myB points to the first element of the array for corresponding threads
     myB=(shared[] int *)&B[MYTHREAD*NELEMS];

     // verify results
     for(i=0;i<NELEMS;i++)
        if(myB[i] != (THREADS-1-MYTHREAD)*NELEMS+i)
        {
           printf("Error: thread:%d, myB[%d]=%d,expect=%d\n",MYTHREAD,i,myB[i],
           (THREADS-1-MYTHREAD)*NELEMS+i);
            upc_global_exit(FAILURE);
        }

     return SUCCESS;
}
```

Computational functions

A variable of type upc_op_t can be specified with the values in Table 4 on page 84. The computational operations that are represented by variables of the type upc_op_t or by user-provided operators, are assumed to be associative.

All operations represented by a variable of type `upc_op_t`, excluding the operation provided by the value `UPC_NONCOMM_FUNC`, are assumed to be communicative. For a reduction or a prefix reduction using operators other than `UPC_NONCOMM_FUNC`, if the result of the operation depends on the order of the operands, the result is undefined.

Table 4. Values and Corresponding Operations

Values	Corresponding operations
UPC_ADD	Addition.
UPC_MULT	Multiplication.
UPC_AND	Bitwise AND for integer and character variables. Results are undefined for floating point numbers.
UPC_OR	Bitwise OR for integer and character variables. Results are undefined for floating point numbers.
UPC_XOR	Bitwise XOR for integer and character variables. Results are undefined for floating point numbers.
UPC_LOGAND	Logical AND for all variable types.
UPC_LOGOR	Logical OR for all variable types.
UPC_MIN	For all data types, find the minimum value.
UPC_MAX	For all data types, find the maximum value.
UPC_FUNC	Use the specified commutative function `func` to operate on the data in the `src` array at each step.
UPC_NONCOMM_FUNC	Use the specified non-commutative function `func` to operate on the data in the `src` array at each step.

Unified Parallel C provides the following functions to perform computational operations on data:

- upc_all_reduce, upc_all_prefix_reduce

upc_all_reduce, upc_all_prefix_reduce

Performs a user-specified operation on all the elements, and returns the value produced with the computational operation to a single thread.

Prototype

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

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

Parameters

dst Points to a shared array that stores the elements produced by the computational operation.

src Points to a shared array that stores the elements to be manipulated.

op Specifies the operation to be performed on all elements.

nelems
Represents the number of the elements in each shared array.

blk_size
Represents the size of the block measured in bytes.

func
Points to a function that is used to perform user-defined operations on all elements.

flags
Controls the data synchronization semantics.

Usage

The function prototypes have different variants depending upon different values of *T*. The following table shows the correspondences between *T* and *TYPE*.

Table 5. T and TYPE Correspondences

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

The ⊕ operator represents the operation performed on all the elements. It is specified by the parameter *op*.

After the `upc_all_reduceT` function returns, the value of the *TYPE* shared object pointed to by `dst` is `src[0]`⊕`src[1]` ⊕• • •⊕ `src[nelems-1]`.

After the `upc_all_prefix_reduceT` function returns, the value of the *TYPE* shared object pointed to by `dst[i]` is `src[0]`⊕`src[1]` ⊕• • •⊕ `src[i]` for `0≤i≤nelems-1`.

Based on the different values of `blk_size` passed to the function, the `src` pointer is treated in the following two ways:

- If `blk_size` is greater than 0, the function treats `src` pointer as if it pointed to a shared memory area with the type `shared [blk_size] TYPE [nelems]`.
- If `blk_size` is 0, the function treats `src` pointer as if it pointed to a shared memory area with the type `shared [] TYPE[nelems]`.

The phase of the pointer `src` is followed in referencing array elements. The `upc_all_reduceT` function treats the pointer `dst` as if it had the type `shared TYPE *`. For the `upc_all_prefix_reduceT` function, the affinity and phase of the pointer `src` must match those of the pointer `dst`.

Examples

Example 1 demonstrates the usage of `upc_all_reduceT`. Example 2 demonstrates the usage of `upc_all_prefix_reduceT`.

Example 1

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

# define  SUCCESS    155
# define  FAILURE    166
# define  BLKSIZE1   5
# define  BLKSIZE2   0
# define  N         100

# define verify(result,expect) \
{ if ((result)!=(expect)) \
   {   \
      printf("Error: fail at line %d: mythread=%d, result= %d, expect= %d\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
   }   \
}

typedef int T; //TYPE is an integer

shared[BLKSIZE1] T srcA[N];
shared[BLKSIZE2] T srcB[N];
shared T *dst;
shared T result;

int main()
{
   int i,j;

   // initialize the shared arrays
   upc_forall(i=0;i<N;i++;&srcA[i])
          srcA[i]=i;

   upc_forall(i=0;i<N;i++;&srcB[i])
          srcB[i]=-i;
   upc_barrier;

   // initialize the dst pointer
   dst=&result;

   /* the value of the TYPE shared object referenced by dst is srcA[0] + srcA[1]
      + srcA[N-1] where 0<= i <=N-1, + is the operator specified by the variable
      op e.g. UPC_ADD
   */
   upc_all_reduceI(dst,srcA,UPC_ADD,N,BLKSIZE1,NULL,UPC_IN_NOSYNC |
   UPC_OUT_NOSYNC);

   upc_barrier;

   // verify results
   verify(*dst,(N - 1)*N/2);

   verify(result,(N - 1)*N/2);

   upc_barrier;

   /* the value of the TYPE shared object referenced by dst is
      srcB[0] + srcB[1] + srcB[N-1] where 0<= i <=N-1, + is
      the operator specified by the variable op e.g. UPC_ADD
   */
```

```
   upc_all_reduceI(dst,srcB,UPC_ADD,N,BLKSIZE2,NULL,UPC_IN_NOSYNC |
   UPC_OUT_NOSYNC);

   upc_barrier;

   //verify results
   verify(*dst,-1*(N - 1)*N/2);

   verify(result,-1*(N - 1)*N/2);

   return SUCCESS;

}
```

Example 2

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

# define  SUCCESS    155
# define  FAILURE    166
# define  BLKSIZE1   5
# define  BLKSIZE2   0
# define  N          100

# define verify(result,expect) \
{ if ((result)!=(expect)) \
   {   \
      printf("Error: fail at line %d: mythread=%d, result= %l, expect= %l\n",
__LINE__, MYTHREAD, result,expect); \
      upc_global_exit(FAILURE); \
   }   \
}

typedef long T; // TYPE is a long

shared[BLKSIZE1] T srcA[N];
shared[BLKSIZE2] T srcB[N];
shared[BLKSIZE1] T dstA[N];
shared[BLKSIZE2] T dstB[N];

int main()
{
   int i;

   // initialize srcA & srcB
   upc_forall(i=0;i<N;i++;&srcA[i])
         srcA[i]=i;

   upc_forall(i=0;i<N;i++;&srcB[i])
         srcB[i]=-i;
   upc_barrier;

   /* function requires upc_threadof(srcA) == upc_threadof(dstA) &&
      upc_phaseof(srcA) == upc_phaseof(dstA), the value of the TYPE
      shared object referenced by dstA[i] is the srcA[0] + srcA[1] +
      ... + srcA[i], 0<= i <=N-1, + is the operator specified
      by the variable op e.g. UPC_ADD
   */
   upc_all_prefix_reduceL(dstA,srcA,UPC_ADD,N,BLKSIZE1,NULL,UPC_IN_NOSYNC |
   UPC_OUT_NOSYNC);

   upc_barrier;

   // verify results
   for(i=0;i<N;i++)
      verify(dstA[i],i*(i+1)/2);
```

```
   /* function requires upc_threadof(srcB) == upc_threadof(dstB) &&
      upc_phaseof(srcB) == upc_phaseof(dstB), the value of the TYPE
      shared object referenced by dstB[i] is the srcB[0] + srcB[1] +
      ... + srcB[i], 0<= i <=N-1, + is the operator specified
      by the variable op e.g. UPC_ADD
   */
   upc_all_prefix_reduceL(dstB,srcB,UPC_ADD,N,BLKSIZE2,NULL,UPC_IN_ALLSYNC |
   UPC_OUT_ALLSYNC);

   // verify results
   for(i=0;i<N;i++)
     verify(dstB[i],-i*(i+1)/2);

   return SUCCESS;

}
```

Chapter 6. Compiler optimization

The XL Unified Parallel C compiler offers a comprehensive set of performance enhancing optimizations which you can take advantage of to develop high performance parallel applications.

The compiler uses a runtime system (RTS) that is designed for scalability in a large parallel computing environment. The RTS exposes an application programming interface (API) to the compiler, and the compiler calls the API functions in the RTS for efficient manipulation of shared data.

In the partitioned global address space (PGAS) programming model, a thread can access a shared object either locally or remotely. A local access is an access to a shared object allocated in the address partition mapped to the accessing thread. A remote access is an access to a shared object allocated in an address partition mapped to a different thread than the accessing thread.

Access to remote shared memory might require network communication between the accessing thread and the thread that has affinity to the shared object. In contrast, a shared local access does not require any network transfer. To improve application performance, the compiler can often bypass RTS calls for local shared accesses and address the shared local memory directly.

To optimize remote shared memory accesses, the compiler can (under certain circumstances) coalesce together remote accesses that requires communication to the same remote thread.

Shared object access optimizations

This topic and the following subtopics describe the optimizations that the compiler can perform on shared object accesses.

In a Unified Parallel C application, shared objects are distributed into different threads. For example, elements of a shared array are distributed among threads of the application based on the layout qualifier used on the shared array declaration.

When a thread reads from or writes to an element of a shared array, the access it performs might be a shared local access or a shared remote access. In the absence of optimizations, the compiler generates calls to the RTS API to read the value of a shared array element or to assign a new value to it. Calling the RTS API to read or write a local shared object value impacts application performance significantly. In the case of a remote shared access, the processing time of a runtime call is compounded by the necessary network communication latency.

The shared object access optimizations have the objective of eliminating unnecessary runtime calls for local shared accesses, and reduce the network communication time required to perform remote shared accesses.

Shared object access privatization

This optimization technique has the goal of eliminating unnecessary runtime calls to access local shared objects.

© Copyright IBM Corp. 2010

Typically, the compiler translates accesses to shared objects by generating appropriate runtime function calls. Any call to the runtime system requires some processing time. When the underlying memory of the shared object is in the local address space of the accessing thread, the thread can access the object without issuing a runtime system call. The compiler achieves this by converting the references to the shared object to traditional C references, thus bypassing the runtime function calls.

Take the following code as an example:

```
shared [N/THREADS] int A[N], B[N], C[N];

void foo()
{
   int i;
   upc_forall(i=0; i<N; ++i; &A[i])
   {
      A[i] = B[i] + C[i];
   }
}
```

In this example, the thread that executes the `i`'th loop iteration is the thread that has affinity with the array element `A[i]`. The `B` and `C` arrays have the same blocking factor as `A`; therefore, all the shared accesses performed in the loop body are local to the issuing thread. The compiler can privatize all the 3 shared array accesses and it does not need to generate runtime calls for translating these accesses.

Data-parallel applications tend to spend most of the computation time in loops. Loop iterations and shared data are typically distributed into multiple threads. When the shared object access privatization optimization is enabled, each thread can access its local shared data directly without calling any runtime function.

Note: When the compiler cannot prove that an access to a shared object is local (the compiler cannot prove that the accessing thread has affinity to the shared object), it must assume that the access is remote and call the appropriate runtime function to access the object.

Shared object access coalescing

This optimization technique is designed to replace multiple shared remote accesses to elements of a shared array having affinity to the same thread with a single remote access.

The compiler typically translates a remote shared access into a call to the appropriate runtime function. For example, reading from or writing to multiple shared array elements that have affinity to a remote thread causes the compiler to generate a runtime call for each of the array elements read or modified. The compiler can combine multiple remote accesses into a single access when it can prove that the shared array elements satisfy the following conditions:

- The shared array elements have affinity to the same thread.
- The distance or stride between consecutive shared array elements accesses is constant.

When multiple remote accesses are combined, the compiler issues a single call to the runtime, thus reducing the number of the communication messages between the accessing thread and the owner thread.

Take the following code as an example:

```
# define ARRAY_SIZE 100
# define BF (ARRAY_SIZE/THREADS)

shared [BF] int A[ARRAY_SIZE];
shared [BF] int B[ARRAY_SIZE];

void loop()
{
   int i;
   upc_forall(i=0; i<ARRAY_SIZE-BF; i++; &A[i])
   {
      A[i] = B[i+BF];
   }
}
```

If the example is compiled in a static environment targeting 2 execution threads running on 2 distinct nodes, then the access `B[i+BF]` is executed by thread 0 for all loop iterations. Elements `B[50]` through `B[99]` have affinity to thread 1 located on a remote node.

In this example, the compiler attempts to coalesce the accesses to `B[i+BF]` together instead of generating a call to dereference individually each of the 50 remote shared array elements.

Shared object remote updating

This optimization technique is designed to replace an update operation to a remote memory object with a specialized runtime call having reduced communication requirements.

When one thread updates a shared object that has affinity to a thread running on a remote node, the two threads perform the following operations:

1. The accessing thread makes a request to the owner thread to retrieve the value of the object and the owner thread responds with the requested value.
2. The accessing thread locally update the value received from the owner thread.
3. The accessing thread sends the new value to the owner thread.

This communication pattern requires 2 calls between the accessing thread and the owner thread. Communication calls impact the performance of an application, and the impact can be significant when the communication is carried over a network.

To mitigate such impact, the compiler can perform the following operations:

1. Recognize the update operation.
2. Generate a special runtime call to instruct the remote thread that owns the array element to perform the update locally.

This optimization results into the execution of a single communication call from the accessing thread to the owner thread, with the update instructions encoded in the call. The update of the shared object is performed by the remote thread, and the number of communication calls is therefore reduced from 2 to 1.

Consider the following example:

```
shared int histogram[N];
extern int my_poll();

int main()
{
   upc_forall(int i=0; i<N; i++; i)
   {
```

```
      histogram[i] = 0;
   }

   upc_barrier;

   for(int i=0; i<K; i++)
   {
      histogram[my_poll()] += 1;
   }
}
```

In this example, we assume that the call to my_poll() produces a pseudo-random number ranging from 0 to N-1. Therefore the shared access to histogram[my_poll()] might be remote to the accessing thread, and in that case its value is updated remotely from another thread. When the shared object remote updating optimization is enabled, the number of communication calls required by this program is reduced in half.

Array idiom recognition

This optimization technique is designed to replace commonly user shared array access patterns with the appropriate Unified Parallel C string handling functions.

Unified Parallel C programs often include statements that manipulate the values of local and shared arrays. Consider the following code as an example:

```
shared [BF] int a[N];
int b[N];

int main ()
{
  int i;
  if (MYTHREAD==0)
  {
    for (i=0; i<N; i++)
      a[i]=b[i];
  }
}
```

In this example, the statement a[i]=b[i] copies a local array to a shared array. Each loop iteration copies one array element at a time. A naive translation of this code results in fine-grained communication for each array element copy. To reduce the number of communication calls, the compiler can perform the following steps:

1. Identify commonly used array manipulation statements (patterns).
2. Classify the operations based on the access pattern.
3. Transform the identified array pattern into a call to one of the Unified Parallel C string handling function calls (upc_memset, upc_memget, upc_memput, or upc_memcpy).

When the array idiom recognition optimization is enabled, the compiler transforms the loop in the preceding example as follows:

```
...
for (i=0; i<N; i+=BF)
{
   upc_memput(&a[i], &b[i], BF*sizeof(b[i]));
}
...
```

Note: The array idiom recognition optimization only considers relaxed shared accesses.

Parallel loop optimizations

This section describes the optimizations that the compiler performs on the `upc_forall` loop.

The `upc_forall` loop is extensively used in XL Unified Parallel C applications to distribute work across threads, and thus the performance of the loop can significantly affect the performance of an application.

To improve the runtime performance of the `upc_forall` loop, the compiler uses the following optimization techniques:

- Create multiple versions of the loop in preparation for locality analysis.
- Remove the affinity branch statement by reshaping the iteration space of the loop.

Loop reshaping

This compiler optimization has the objective of removing the overhead of the affinity branch from a `upc_forall` loop body.

The compiler typically transforms a `upc_forall` loop into a `for` loop. To respect the `upc_forall` semantics, the compiler can naively translate the affinity expression into a branch controlling the execution of the for loop. The branch ensures that only the correct loop iterations are executed by a given thread.

The following two examples illustrate two possible naive transformations of a `upc_forall` loop.

Example 1: Transforming a `upc_forall` loop that has the integer affinity:

```
shared double a[N], b[N], c[N];

void foo_1 ()
{
   int i;
   upc_forall (i=0; i<N; i++, i)
     a[i]=b[i]+c[i];
}
```

The `upc_forall` loop example 1 can be transformed into the following `for` loop containing an `if` conditional statement based on the integer affinity expression:

```
shared double a[N], b[N], c[N];

void foo_1 ()
{
   int i;

   for (i=0; i<N; i++)
   {
      if ((i%THREADS)==MYTHREAD)
         a[i]=b[i]+c[i];
   }
}
```

Example 2: Transforming a `upc_forall` loop that has the address affinity:

```
shared double a[N], b[N], c[N];

void foo_2 ()
{
```

```
   int i;
   upc_forall (i=0; i<N; i++, &a[i])
      a[i]=b[i]+c[i];
}
```

The `upc_forall` loop example 2 can be transformed into the following `for` loop containing an `if` conditional statement based on the address affinity expression:

```
shared double a[N], b[N], c[N];

void foo_2 ()
{
   int i;
   for (i=0; i<N; i++)
   {
      if (upc_threadof(&a[i])==MYTHREAD)
         a[i]=b[i]+c[i];
   }
}
```

In these examples, the compiler inserts a branch statement into the loop body. Note that the branch must be evaluated (by every thread) in each loop iteration and therefore it severely impacts the scalability of the loop and the performance of the program.

To address this issue, the XL Unified Parallel C compiler optimizes the `upc_forall` loop without inserting the branch statement. The following subtopics provide details about the loop reshaping optimization.

Reshaping loops with integer affinity

Describes the technique used to optimize a `upc_forall` loop with integer affinity.

To optimize a `upc_forall` loop with an integer affinity expression, the compiler changes the lower bound of the loop to `MYTHREAD` and the increment of the loop to `THREADS`.

Consider the `upc_forall` loop in the following example:

```
shared double a[N], b[N], c[N];

void foo ()
{
   int i;
   upc_forall (i=0; i<N; i++; i)
      a[i]=b[i]+c[i];
}
```

When the loop reshaping optimization is enabled, the compiler transforms the loop in the example as follows:

```
shared double a[N], b[N], c[N];

void foo ()
{
   int i;
   for (i=MYTHREAD; i<N; i+=THREADS)
      a[i]=b[i]+c[i];
}
```

Reshaping loops with address affinity

Describes the technique used to optimize a `upc_forall` loop with address affinity.

To optimize a upc_forall loop with an address affinity expression, the compiler translates the upc_forall loop into two nested loops-the outer loop iterates over blocks of array elements with affinity to the same thread and the inner loop iterates through each element in a block.

Consider the following code as an example:

```
shared [2] int a[N], b[N], c[N];
int i;

upc_forall (i=0; i<N; i++; &a[i])
{
  a[i]=b[i]+c[i];
}
```

When the loop reshaping optimization is enabled, the compiler translates the preceding code as follows:

```
shared [2] int a[N], b[N], c[N];
int i, j;

for (i=MYTHREAD*2; i<N; i+=THREADS*2)
{
   for (j=i; j<i+2; j++)
   {
      a[j]=b[i]+c[i];
   }
}
```

The absence of the affinity branch allows the loop to be executed efficiently and to scale up to large number of threads.

Note: If the compiler cannot determine how to optimize a upc_forall loop, it must insert a branch statement into the loop body based on the affinity test.

Loop versioning

This optimization has the objective of facilitating further analysis of a upc_forall loop containing indirect shared memory accesses.

Locality analysis is an important optimization technique. It helps the compiler identify opportunities for shared object privatization and coalescing.

To prepare a upc_forall loop containing indirect shared references (through a pointer-to-shared) for locality analysis, the compiler performs the following steps:

1. Creates two control flow paths, the *true* path and the *false* path.
2. Computes the expression that, when evaluates to *true*, leads the program control branch into the *true* path. The condition the compiler generates guarantees that all the shared accesses through a pointer-to-shared in the *true* path have known locality, that is, the pointer-to-shared points to the first array element in a block, and the pointer is loop invariant.
3. Creates two copies of the original loop and place each copy of the loop in one of the two control paths.

This transformation allows the compiler to perform locality analysis on the indirect accesses to shared objects through a pointer-to-shared in the *true* path.

Consider the following example:

```
...
void foo(shared [BF] int *p1, shared [BF] int *p2)
{
   int i;
   upc_forall (i=0; i<N; ++i; &p1[i])
      p1[i]=p2[i+1];
}
```

When the loop versioning optimization is enabled, the compiler transforms the code in this example as follows:

```
...
void foo(shared [BF] int *p1, shared [BF] int *p2)
{
   int i;
   _Bool ver_p1=(upc_phaseof(p1)==0)?1:0;
   _Bool ver_p2=(upc_phaseof(p2)==0)?1:0;
   if (ver_p1 && ver_p2)
   {
      // true path
      upc_forall (i=0; i<N; i++; &p1[i])
         p1[i]=p2[i+1];
   }
   else
   {
      // false path
      upc_forall (i=0; i<N; i++; &p1[i])
         p1[i]=p2[i+1];
   }
}
```

In the transformed code, `p1` and `p2` are considered for locality analysis in the *true* path, whereas they are not considered for the analysis in the *false* path.

The loop versioning transformation helps the compiler prepare the indirect accesses to a shared object in the `upc_forall` loop for locality analysis, and can consequently contribute to an improvement in application performance.

Note: In this particular example, note that in the *false* path, `p1` has the exact same index as the affinity expression and the compiler can still prove that it is always local, so the compiler can always privatize `p1`.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2010

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2010. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Index

Special characters

-qupc compiler option 9
    9, 11

C

compiler options
    -qupc 9

© Copyright IBM Corp. 2010

Printed in USA