HP UPC

Unified Parallel C (UPC) Programmer's Guide

April 2005

This manual contains information about developing HP UPC programs on HP UX, Tru64 UNIX, and XC Linux systems. It describes language features specific to the HP UPC product.

Revision/Update Information: Revised for Version 2.4 of HP UPC.

Software Version: HP UPC Version 2.4

Hewlett-Packard Company
Palo Alto, California

This document was prepared using DECdocument, Version 3.3-1n.

Contents

1 Introduction

This manual contains information about the UPC parallel extension to the C language, and about developing UPC programs on HP UX, Tru64 UNIX, and XC Linux systems. It is intended as a supplement to the *HP Tru64 UNIX Programmer's Guide*. Some of the information in this manual has been taken from the following sources:

- The *Introduction to UPC and Language Specification* CCS-TR-99-157, by William W. Carlson, Jesse M. Draper, et al, May 13, 1999
- The Quadrics™ *RMS User Manual*
- The UPC Language Specification

The information in this manual pertains to the HP UPC implementation for HP UX, Tru64 UNIX, and XC Linux systems, and is not intended to describe either the theoretical UPC language design or the implementation of UPC by any other company.

1.1 The UPC Language

The UPC language was designed as a unification of three earlier languages, all modifications to the C language and all intended to support an explicit parallel programming model. These three are:

AC	Designed and developed by Bill Carlson and Jesse Draper of the Center for Computing Sciences, originally for the CM-5 from Thinking Machines and later adapted to the Cray T3 series.
Split C	Designed and developed by David Culler, Kathy Yelick et al. at the University of California-Berkeley.
Parallel C Preprocessor	Designed and developed by Eugene Brooks and Karen Warren at Lawrence Livermore Laboratories.

The Unified Parallel C (UPC) language is not a superset of these three predecessors. Rather, it is a distillation of the essential features needed to support programming for a wide variety of parallel computer architectures in an efficient manner.

The original UPC implementation was adapted from the AC compiler for the Cray T3D and T3E machines. The HP UPC compiler and Run-Time System for AlphaServer SC systems is the first independent implementation, and is based on the HP C++ V6 series of compilers.

UPC is an extension to C99. It provides a simple shared memory model for parallel programming, allowing data to be shared or distributed among a number of communicating processors. Constructs are provided in the language to permit simple declaration of shared data, distribute shared data across threads, and synchronize access to shared data across threads.

1.2 Product

HP UPC is a fully conforming implementation of the UPC language, with some extensions, primarily for compatibility with the HP C and HP C++ products. UPC is fully compliant with the ANSI C99 specification except for complete implementation of the complex data type. Note also that the C run-time library used by UPC might not provide complete implementation of the C99 run-time components. UPC supports variable length arrays on the Tru64 UNIX platform, but not on the HP-UX platform. HP UPC supports the UPC language specification Version 1.2 developed by the UPC Consortium and released in May 2005.

The HP UPC product consists of the following components:

- The compiler driver program, `upc`, controlling the compilation and linking process.
- The compiler, `upc_cc`, run by the driver.
- The UPC Run-Time library, `libupc`, providing the interface between the compiler-generated code and the memory interconnect system. The library is also referred to as the *UPC Run-Time System*, or UPCRTS.
- The `prun` command, provided with the AlphaServer SC software or the `srun` command provided with the Simple Linux Utility for Resource Management (SLURM) on XC Linux systems.
- Optional Run-Time Environment components (see Chapter 5).

1.3 Language Overview

UPC has three concepts that distinguish it from other programming languages designed to support parallelism:

1. An explicitly parallel execution model
2. Separate private and shared address spaces
3. C-style pointers into the shared address space

1.3.1 Explicitly Parallel Execution Model

The execution model used by UPC is *Single Program Multiple Data (SPMD)*, and is *explicitly parallel*. That is, the executable UPC program is run, from the beginning and in its entirety, on each CPU independently. Thus, the programmer must constantly be aware that, unless he explicitly provides an alternative, the same actions will be performed on every CPU involved. If an ordinary ANSI C program is compiled and run as a UPC program, it will work normally, but it will be replicated on as many CPUs as requested in the `prun` command. The language features **THREADS**, **MYTHREAD**, **pointer to shared**, and **upc_barrier** are provided to allow the programmer to perform a different computation on each CPU, even though the same executable code is running on each.

The UPC explicitly parallel execution model is similar to that used by the MPI and PVM styles of parallel programming, and for the same rationale. That is, by allowing each instance to run independently, there is no need for one thread of control to interfere with the other threads achieving their best possible performance. The extent to which one thread affects the behavior of other threads of control is entirely under the programmer's control. This model also allows for the greatest breadth of implementation designs. Both shared memory and distributed memory architectures are supported by UPC in such a way as to deliver their best possible performance, respectively. Hybrid, or *distributed shared memory* systems can also be exploited to their best by the UPC programming model. Note, however, that HP UPC does not support mixed MPI.

The UPC execution model is dramatically different from the model used by the *OpenMP* extensions to the Fortran, C, and C++ languages, and the HPF language. The *OpenMP* model is *implicitly* parallel. That is, the programmer specifies the execution of a single *master* thread of control. At certain points in the program, the programmer defines opportunities for parallel operation by *slave* processors, whose recruitment, data availability, and synchronization is entirely under the control of the master. This strong interdependence means that these languages must provide a relatively rich set of parallelism primitives if adequate performance is to be maintained with a wide variety of algorithms. Because of its explicitly parallel model, UPC can express a wide variety of parallel concepts with a few simple extensions to the base language.

1.3.2 Separate Private and Shared Address Spaces

Like many other programming languages, UPC provides two separate address spaces: a *private* space, where the same address accessed on each thread operates on distinct physical memory cells, and a *shared* space, where the same address on each thread refers to the same memory cell. Typically, however, the concept of truly *shared* memory is foreign to most other *explicitly parallel* parallel programming paradigms, such as MPI. Likewise, truly *private* memory is foreign to implicitly parallel models like OpenMP.

In UPC, a compromise is used. Each shared memory cell in a running UPC program is actually associated with a particular thread of control. This association is called *affinity*. Thus, while every thread can address every shared memory cell, cells with *affinity* to a given thread can be accessed more quickly. This concept is expressed in the UPC syntax by the ability of the programmer to cast a pointer to a shared memory cell, with *affinity* to the current thread, to a pointer to a private memory cell. This ability is important, because it allows the programmer to access the *local* portion of shared memory with the same performance as that thread's access to its private memory. It is generally expected that access to any shared memory cell is slower than access to a private one. This performance difference can vary significantly among implementations, and from one shared reference to another within a single implementation. This happens because run-time optimizations, such as caching and prefetching, can turn a *remote* shared access into a *local* one.

Reading and writing to shared memory locations is the primary way that one thread affects the behavior of others in a running UPC program. Because all threads have access to the same pool of shared memory, the programmer must take care to insure that one thread doesn't destroy the correct inputs or outputs of the others.

The primary language feature that allows the UPC programmer to control this interference is the use the **relaxed** and **strict** type qualifiers. On one hand, it is absolutely necessary to serialize access to commonly used memory locations, otherwise each thread can destroy the proper inputs or outputs of another. On the other hand, when the programmer knows that the individual threads are not touching one another's inputs and outputs, he doesn't want to incur the performance impact of such serialization. How this choice is made for each individual shared memory location is determined by the **strict** or **relaxed** type qualifiers, used when the location (variable, array, etc.) is declared. Variables declared **strict** incur serialization overhead associated with each reference; variables declared **relaxed** do not.

Note

In UPC, the term *threads* means separate threads of execution, which in HP UPC are UNIX processes running on various CPUs. They have separate address spaces from each other (except for shared data, as described earlier). They are *not* the same as threads in a `pthread` application, nor are they kernel threads. UPC does not support mixed pthreads.

1.3.3 C-style Pointers Into the Shared Address Space

The use of pointers is central to the C language. Other dialects of C that support parallelism, such as the use of MPI in C programs, do not support pointers to reference locations outside each thread's local memory. This dramatically degrades the ease of programming such dialects. The UPC language derives most of its ease of programming from the ability to declare, define, and reference data through pointers into shared memory space. With a single stroke, this feature provides access to all the ordinary programming techniques used in C. By providing such pointers, the UPC language allows ordinary C language expressions to support parallelism in a straightforward way. To the extent that UPC programming differs from ordinary C, it is primarily the need to properly manage the type distinction between pointers to private and shared memory locations.

Inside the compiler and the Run-Time System for UPC, pointers to shared space are not implemented as ordinary addresses, but as composite structures, encoding not only the address but also the affinity, and possibly the block offset, of the destination value. Thus, arithmetic operations on such pointers, for example `p++`, can incur considerable computational load. This is another reason for providing the feature of casting a pointer to shared to a **pointer-to-private** space.

1.3.4 Large Scale Synchronization and Resource Management

All parallel programming techniques require mechanisms to bring all threads to a single point in the computation, and to manage resources such as dynamic memory. UPC provides these in a straightforward manner as statements like `upc_fence`, `upc_barrier`, `upc_forall` and functions such as `upc_all_alloc()`. Mastering the concepts discussed in this introduction will provide most of the tools needed to express parallel algorithms in a succinct and efficient manner in the UPC language.

2 Language Features

This chapter describes the following UPC lanugage features:

- Section 2.1, Data Sharing Model
- Section 2.2, Strict And Relaxed Data Access
- Section 2.3, Special Constants
- Section 2.4, UPC_MAX_BLOCK_SIZE Defined Constant
- Section 2.5, Type Qualifiers
- Section 2.6, Pragma upc strict and pragma upc relaxed
- Section 2.7, Pragma upc nested and pragma upc unnnested
- Section 2.8, Operators
- Section 2.9, Built-in Functions
- Section 2.10, Shared Arrays
- Section 2.12, Statements

HP UPC is in full compliance with Version 1.2 of UPC Language Specification, which you can from the George Washington University UPC Documentation page.

2.1 Data Sharing Model

A UPC program runs simultaneously on one or more threads per processor in a multiprocessor system. Each instance of the program has a separate memory space. Variables are not shared among the threads unless explicitly marked as shared.

Shared data always have affinity to some thread. *Having affinity to* some thread is sometimes referred to as *being local to* that thread. Arrays of shared data are distributed among the threads essentially equally. Shared scalars have affinity to thread 0. Dynamically allocated shared data can have affinity to any thread. Array elements are allocated in round-robin fashion to the various threads, beginning on thread 0. A *block size* indicates how many elements are consecutively allocated to one thread before proceeding to the next thread when allocating a shared array among the various threads.

References (fetches and stores) to private data proceed exactly as they do in an ordinary C program. Shared data references are transformed by the compiler into an appropriate series of function calls to fetch or store the data from the thread to which it has affinity. Typically this will involve transferring data over the Quadrics memory interconnect switch on AlphaServer SC and HP SMP systems, although optimizations may be performed based on knowledge of the exact memory configuration.

2.2 Strict And Relaxed Data Access

The keywords **strict** and **relaxed** specify a method to be used for access (fetches or stores) to shared data. The selection of a method is referred to as specifying the *coherence*. The coherence may be specified in the type of the variable being accessed, or it may be specified in the code that is performing the access.

In all cases, from the viewpoint of a single thread, references to shared data appear to execute in the order they were issued. For example:

```
shared int x, y;
x = 0;
...
x = 1;
y = x;
```

Unless x is overwritten by another thread, y receives the value 1, regardless of whether the various references to x and y are strict or relaxed.

strict
Indicates that all modifications to shared data are visible on all threads in the order they are performed. This behavior is implemented as follows:

- Strict stores:
  - Wait for any pending relaxed stores issued by the current thread to be completed before the current strict store is begun, to avoid data being overwritten.
  - Wait for the current store to be completed before proceeding.
- Strict fetches:
  - Wait for any pending stores issued by the current thread to be completed before the current fetch is begun, so as to get the most recently written value.

Note that *all* pending fetches or stores—strict or relaxed—issued by the current thread are completed before a strict access is performed. Also note that compiler code motion is restricted; that is, the compiler cannot move a fetch or store of shared data past a strict reference, in either direction.

relaxed
Indicates that data are to be written and read without cross-thread synchronization, and thus, typically, with higher performance. Results of a sequence of relaxed operations (fetches and stores) may appear to another thread to have been executed in any order. Programs must synchronize their use of data using strict references, fences, and barriers at appropriate times. Fence statements (Section 2.12.4), barrier statements (Section 2.12.2), and explicit strict references via `#pragma upc strict` (Section 2.6) provide the necessary synchronization by waiting for store completion or delaying fetches before proceeding.

Note that if a single thread stores a series of values to a single location, the last value stored is the one that is visible past a synchronization point, and all threads agree on values after synchronization. For example:

```
shared int x;
x = 0;
...
x = 1;
barrier;
```

After the barrier, all threads see x with a value of 1, regardless of whether the stores to x were strict or relaxed.

The coherence may be specified in the type (see Section 2.5.2), globally or locally in the source code via a pragma (see Section 2.6), or via a command line option (see Section 4.3), where the command line is equivalent to a global pragma setting at the beginning of the source code. (The UPC header files `upc_strict.h` and `upc_relaxed.h` set the pragma state to **strict** or **relaxed**, respectively.) If neither the pragma nor the command line switch is seen, the default value for the coherence is **relaxed**.

When determining the coherence to use for a particular shared data reference, the order of precedence is as follows:

1. Does the type specify the coherence? If not,
2. Is there a local pragma setting for the coherence? If not,
3. Is there a global pragma setting? If not,
4. Use relaxed.

Note that specifying the coherence in a variable declaration is equivalent to specifying it in the type. The coherence affects the type of the variable, not the variable itself. The **relaxed** and **strict** keywords are, formally, type qualifiers in the C language.

Table 2–1, derived from an example in *Introduction to UPC and Language Specification*, illustrates different ways to use **strict** and **relaxed**. All four sequences have identical meanings.

Table 2–1 Equivalent Code Sequences Illustrating Use of Strict and Relaxed

Strict Default / **Relaxed Default**

Using Types

```
#include <upc_strict.h>                 #include <upc_relaxed.h>
shared int flag;                        strict shared int flag;
relaxed shared int data1, data2;        shared int data1, data2;

send(int val1, val2) {                  send(int val1, val2) {
  while (flag) /* loop */ ;               while (flag) /* loop */ ;
  data1 = val1;                           data1 = val1;
  data2 = val2;                           data2 = val2;
  flag = 1;                               flag = 1;
}                                       }
int rcv() {                             int rcv() {
  int tmp;                                int tmp;
  while (!flag) /* loop */ ;              while (!flag) /* loop */ ;
  tmp = data1 + data2;                    tmp = data1 + data2;
  flag = 0;                               flag = 0;
  return tmp;                             return tmp;
}                                       }
```

Using Pragma

```
#include <upc_strict.h>                 #include <upc_relaxed.h>
shared int flag, data1, data2;          shared int flag, data1, data2;

send(int val1, val2) {                  send(int val1, val2) {
  while (flag) /* loop */ ;               { /* scope for pragma */
  { /* scope for pragma */              #pragma upc strict
#pragma upc relaxed                         while (flag) /* loop */ ;
    data1 = val1;                         }
    data2 = val2;                         data1 = val1;
  }                                       data2 = val2;
  flag = 1;                               { /* scope for pragma */
}                                       #pragma upc strict
int rcv() {                                 flag = 1;
  int tmp;                                }
  while (!flag) /* loop */ ;            }
  { /* scope for pragma */              int rcv() {
#pragma upc relaxed                       int tmp;
    tmp = data1 + data2;                  { /* scope for pragma */
  }                                     #pragma upc strict
  flag = 0;                                 while (!flag) /* loop */ ;
  return tmp;                             }
}                                         tmp = data1 + data2;
                                        #pragma upc strict
                                          { /* scope for pragma */
                                            flag = 0;
                                          }
                                          return tmp;
                                        }
```

2.3 Special Constants

The UPC language specifies two important constant integer values, **THREADS** and **MYTHREAD**. These provide the programmer with the primary means to specify different computational values on different threads, even though the same program is being executed.

2.3.1 THREADS

THREADS is a named constant whose value indicates how many instances of the program are being run. By default, it is a run-time constant, but it can be made a compile-time constant (see Section 4.3). As a run-time constant, it is used as a multiplier in one dimension of a shared array (see Section 2.10.1).

If **THREADS** is a compile-time value, it is treated exactly as a constant of the specified value, and may be used in all places where an integer constant can be used, including array dimensions. It may be used in preprocessing directives.

2.3.2 MYTHREAD

MYTHREAD is a named constant whose value indicates the thread number of the current thread. It is a run-time constant with a different value for each instance of the program. Its value ranges from 0 to (**THREADS** - 1).

2.4 UPC_MAX_BLOCK_SIZE Defined Constant

The macro `UPC_MAX_BLOCK_SIZE` is defined by the compiler as the largest value that may be specified for a block size. It is an unsigned long type and is a compile-time constant. See Section 4.3 for a description of the `-narrow` and `-wide` options used to specify the size of the the pointer to shared structure. The maximum block size is dependent on the size of the data structure used for pointers-to-shared, which is 128 bits on HP-UX XC Linux and and either 64 (default) or 128 bits on Tru64 UNIX.

2.5 Type Qualifiers

The following sections describe type qualifiers.

2.5.1 Shared

Shared data are indicated by the **shared** type qualifier, optionally followed by a *block size* in square brackets, and optionally accompanied by either the **strict** or **relaxed** type qualifier. Any data declared without the use of **shared** are local to each thread, and cannot be referenced or modified by a different thread.

Shared arrays (Section 2.10) are distributed across all threads. Shared scalars all have affinity to thread 0.

Shared data may not be part of a structure or union. Only the entire structure or union may be shared. Structures may, however, contain pointers to shared data. Shared structures may contain pointers to local data, but those pointers are not likely to be valid on all threads.

Shared data cannot have the `auto` storage class. They can use the `static` or `extern` storage classes, or they can be dynamically allocated (see Section 3.3). Shared data are not allowed to be statically initialized.

2.5.1.1 Block Size

The block size specifier is most appropriate for arrays and pointers, but can be used in scalar declarations. It indicates the number of underlying array elements that are grouped together on a single thread.

Block size is specified in brackets immediately following the **shared** type qualifier. The value specified must be a compile-time positive integer constant. If no block size is specified, a block size of one is used. For example:

```
shared [3] int x[3*THREADS];
shared [5] double *p;
shared long y[2*THREADS];  // Uses default block size of 1
shared [3*2] int w[2][3*THREADS];    // Block size is 6
```

Empty brackets indicate *indefinite block size*; all the elements have affinity to the same thread. Pointers and scalars may be specified with indefinite block size. If an array is specified with indefinite block size, the dimensions may not include a multiple of **THREADS**, and the entire array will have affinity to thread zero. For example:

```
shared [] double *p;
shared [] long a[35];
```

A block size specified as "*" indicates that the compiler is to calculate the block size as the number of elements per thread, resulting in all elements for a given thread being contiguous. This form of block size specification is called *pure block allocation*. If **THREADS** is specified at compile time, and the number of elements in an array is not an even multiple of **THREADS**, the value is rounded up to ensure all elements for a given thread are contiguous. Some examples:

```
shared [*] int A[3*THREADS][5][12];
/* Block size is 3*5*12 = 180 */

shared [*] int B[3][5][12];
/* If THREADS is 3, block size is 60 */
/* If THREADS is 7, block size is 26 */
```

If the **shared** type qualifier appears twice in a declaration, the block sizes (Section 2.5.1.1) must match. For example:

```
shared shared [3] int x; // Mismatch: default block size 1 vs explicit block size 3
typedef shared [3] int S3INT;
shared [3] S3INT y;      // Valid
shared S3INT z;          // Mismatch: default block size 1 vs explicit block size 3
```

Block size specifiers have higher precedence than array dimensions. A type specified as `int shared [5]` indicates a shared integer with a block size of 5, rather than an array[5] of shared integers with the default block size of one.

Multiple bracketed expressions after the **shared** keyword are interpreted as an attempt to declare multiple block sizes and are diagnosed as errors. In object declarations, or function prototypes where the arguments indicate variable names, the block size specification and an array dimension would be separated by an identifier. In cases where they are not, such as in casts or in function declarations without specifying argument variable names, the array dimensions must be enclosed in parentheses. Table 2–2 illustrates this.

Table 2–2 Resolving Ambiguity Between Block Size and Array Dimensions

Declaration	Interpretation	Comments
int shared [3]	shared (blocksize 3) integer	Bracketed expression is interpreted as block size
int shared [3] [4]	Rejected	Only one block size allowed
int shared [3] ([4])	Array[4] of shared (blocksize 3) integer	Parentheses avoid ambiguity
int shared [3] x[4]	x is an array[4] of shared (blocksize 3) integers	The identifier avoids ambiguity
shared [3] int [4]	Array[4] of shared (blocksize 3) integer	Putting the qualifier before the type rather than after it avoids ambiguity
int * shared [3]	Shared (blocksize 3) pointer to local integer	Bracketed expression is interpreted as block size; note that the pointer is shared, not the pointed-to data
int * shared [3] [4]	Rejected	Only one block size allowed
int * shared [3] ([4])	Array[4] of shared (blocksize 3) pointer to local integer	Parentheses avoid ambiguity
int * shared [3] x[4]	Array[4] of shared (blocksize 3) pointer to local integer	The identifier avoids ambiguity

See Section 2.10.2 for a discussion on how block size is used in shared array element distribution.

The block size of shared data matters when determining type compatibility of pointers to shared data. A pointer with a block size of one is compatible with a pointer with any other block size, but otherwise an explicit cast is required. See Section 2.11.

The `shared void *` type is a generic pointer to shared data. When an assignment is made to or from a `shared void *` pointer, no cast is necessary.

2.5.2 Strict and Relaxed

The **strict** and **relaxed** type qualifiers indicate that all references to data of the type being specified are to use the strict or relaxed coherence, respectively. If neither is specified, the coherence used will be governed by global and local settings in the code. **Strict** and **relaxed** may not be used together, and they may only be specified for shared data.

Example 2–1 Declarations Using strict and relaxed

```
strict shared float *p;    // Pointer to strict shared float
shared relaxed int x;

typedef strict shared int ssi;
ssi ssi_array[3*THREADS];  // Array of strict shared int
```

(continued on next page)

Example 2–1 (Cont.) Declarations Using strict and relaxed

```
typedef shared int SA[5*THREADS];
strict SA SA_strict;       // Array of strict shared int
relaxed SA SA_relaxed;     // Array of relaxed shared int
SA SA_unspecified;         // Array of shared int
```

See also Section 2.2 for further information on strict and relaxed data access.

2.6 Pragma upc strict and pragma upc relaxed

`#pragma upc strict` and `#pragma upc relaxed` specify a strict or relaxed coherence, respectively, for variables without an explicit coherence. There are two forms: *global* and *local*, distinguished by where the pragma appears in the source code.

The global form changes the setting for the remainder of the compilation unit. It appears at file scope, outside of any function definition.

The local form changes setting for the current block. It appears inside a block or inside a function body, before any data declarations. If the pragma appears anywhere else within a block, an error will be generated.

The global state of the pragma may be saved and restored. This can be useful inside included files, where some section of code needs a particular setting that may or may not be different. To save or restore the coherence state, specify `#pragma upc coherence` followed by either `save` or `restore`. (These options are not supported on the HP-UX platform).

```
#pragma upc coherence save
#pragma upc coherence restore
```

Example 2–2 illustrates how to use the pragma. Table 2–1 shows uses of the pragma in conjunction with type declarations specifying coherence.

2.7 Pragma upc nested and pragma upc unnested

HP UPC provides an additional pragma option for use with the `upc_forall statement`. The statement has significantly different semantics depending on whether it is dynamically nested within another `upc_forall statement`. If the programmer can assert that a particular `upc_forall statement` is never nested, or that all `upc_forall` statements are never nested, the compiler can be more aggressive in optimizing the loop. To allow programmers to make such assertion, HP UPC provides `#pragma upc nested` and `#pragma upc unnested`. If either `#pragma upc nested` or `#pragma upc unnested` appears at file scope, it affects all `upc_forall` statements in the remainder of the compilation unit. If the pragma appears inside the body of a `upc_forall` statement, before any other statements or any data declarations, it affects only that `upc_forall` statement. If the pragma appears in any other location, an error is generated. If a `upc_forall` statement is declared unnested, it makes a run-time assertion that no other `upc_forall` statement is active upon entry to the unnested loop, and it generates code that does not allow nesting. If a `upc_forall` statement is declared nested, it will allow for either nested or unnested execution.

Example 2–2 Uses of `#pragma upc strict` **and** `#pragma upc relaxed`

```
shared int x[3*THREADS], y;
int i;
#pragma upc strict             // Sets global state
void f1(void) {
  x[2] = y;                    // Strict references
  for (i = 0; i < 3*THREADS; i++) {
#pragma upc relaxed            // Sets state within for body
    x[i] = i;                  // Relaxed reference
    y++;                       // Relaxed reference
  }
  y--;                         // Strict reference
  {
#pragma upc relaxed            // Sets state within block
    x[3] = 2;                  // Relaxed reference
    y += x[7];                 // Relaxed references
  }
  }
#pragma upc coherence save     // Saves global state (strict)
#pragma upc relaxed            // Sets global state to relaxed
void f2(void) {
  x[3]++;                      // Relaxed reference
  y--;                         // Relaxed reference
 }
#pragma upc coherence restore  // Restores global state (strict)
void f3(void) {
  y++;                         // Strict reference
 }
```

Example 2–3 Uses of `#pragma upc nested` **and** `#pragma upc unnested`

```
#pragma upc unnested   // Sets global state
upc_forall(...) {...}  // Asserted not to be nested
upc_forall(...) {
#pragma upc nested     // This loop can be nested
  ...
}
upc_forall(...) {...}  // Asserted not to be nested
#pragma upc nested     // Sets global state
upc_forall(...) {...}  // Can be nested
upc_forall(...) {
#pragma upc unnested   // Asserted not to be nested
  ...
}
upc_forall(...) {...}  // Can be nested
```

2.8 Operators

Several function syntax operators are provided for determining information about shared types.

2.8.1 upc_blocksizeof

The **upc_blocksizeof** operatior returns the block size of a shared variable, type, or expression, in a manner similar to `sizeof`.

If the type was declared with indefinite block size, the value 0 is returned. The macro `UPC_INDEFINITE_BLOCK_SIZE` is defined to zero in `upc.h` for convenience.

If the type is not a shared type, a value exceeding the maximum block size (specifically, `UPC_MAX_BLOCK_SIZE+1`) is returned. The macro `UPC_NON_SHARED_BLOCK_SIZE` is defined in `upc.h` for convenience.

Given the following declarations:

```
shared [15] int x;
shared [] int *p;
shared [15] int * shared [3] sp;
shared [15] int a[15*THREADS];
```

The table below shows the result when **upc_blocksizeof** is applied to the operand.

Operand	Result	Comments
`x`	15	
`&x`	`UPC_NON_SHARED_BLOCK_SIZE`	Pointer to shared type is not shared
`p`	`UPC_NON_SHARED_BLOCK_SIZE`	Pointer is not itself shared
`*p`	`UPC_INDEFINITE_BLOCK_SIZE`	
`sp`	3	This pointer *is* shared
`*sp`	15	
`a[2]`	15	
`&a[2]`	`UPC_NON_SHARED_BLOCK_SIZE`	Pointer to shared type is not shared
`a`	15	

2.8.2 upc_localsizeof

The **upc_localsizeof** operator returns the size of that portion of a shared variable, type, or expression that has affinity to each thread, in a manner similar to `sizeof`. The operation returns the size of the *locally allocated* portion of the data. For shared arrays and shared array types, the returned value is the size used for static allocation. For any other types of data, this operation is the same as `sizeof`.

Shared array types that are not evenly blocked (see Section 2.10.3) may require extra allocated space to account for elements needed on a subset of the threads. Thus, the size of the array as returned by `sizeof` may not be the same as the value returned by `upc_localsizeof` multiplied by **THREADS**.

2.8.3 upc_elemsizeof

The **upc_elemsizeof** operator returns the *underlying element size* of an array type, in a manner similar to `sizeof`. It takes a single argument. If the type of the argument is an array type, **upc_elemsizeof** returns the size of the most significant sub-element of that array type that is not an array, disregarding typedefs. If the type of the argument is not an array type, **upc_elemsizeof** returns the size of the argument type.

For example, in all of the following cases `upc_elemsizeof(A)` would return the size of an `int`:

```
int A;

typedef int T1[3];
T1 A;

typedef int T2[3][6];
T2 A[5];
```

In this example, however, `upc_elemsizeof(A)` would return the size of a `T3`:

```
typedef struct {
  int inside[3];
} T3;
T3 A[5];
```

2.9 Built-in Functions

UPC provides several built-in functions for the purpose of detecting characteristics of pointers to shared data. Note that, because these are built-in functions rather than actual functions, they cannot have their addresses taken.

2.9.1 upc_threadof

The **upc_threadof** function takes one argument, a pointer to shared data, and returns the **MYTHREAD** value for the thread to which that shared location has affinity.

2.9.2 upc_phaseof

The **upc_phaseof** function takes one argument, a pointer to shared data, and returns the offset within the block containing it. Note that the offset is represented in terms of the base elements of the array. For example, if the pointer is pointing into a block of five elements (numbered 0 through 4) at element 3, `upc_phaseof` would return 3.

2.9.3 upc_addrfield

The **upc_addrfield** function takes one argument, a pointer to shared data, and returns the address field from the pointer structure. This function is intended for use in debugging only. It can help see the effects of shared pointer arithmetic.

The `upc_addrfield` function is *not* the equivalent of a cast to a pointer to private, even if the shared pointer refers to data with affinity to the current thread. The address field is an internal representation only.

2.9.4 upc_affinitysize

The **upc_affinitysize** function calculates the size of the local portion of the data in a shared object with affinity to a given thread. a value of zero for *nbytes* indicates indefinite block size, and the allocation is assumed to have affinity to thread zero in its entirety, as would be the case with a static declaration.

Syntax is as follows:

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

The *totalsize* variable is the total size of the allocation in bytes; *nbytes* is the number of bytes in a block; *threadid* is the thread whose **affinitysize** is to be evaluated.

Examples:

```
shared [5] int A1[3][7][2*THREADS];
shared int *A2;
shared [] int *A3;
size_t A1size, A2size, A3size;

A2 = upc_all_alloc(25, 3*sizeof(int));
A3 = upc_alloc(17*sizeof(int));
```

```
  A1size = upc_affinitysize(sizeof(A1), upc_blocksizeof(A1) * sizeof(int), threadid);
  A2size = upc_affinitysize(25*3*sizeof(int), 3*sizeof(int), threadid);
  A3size = upc_affinitysize(17*sizeof(int), 0, threadid);
```

In this example `A3size` will be zero if `threadid` is not zero.

2.9.5 upc_reset_phase

The `upc_reset_phase()` function returns a pointer that is a copy of its argument, except that the phase of the returned pointer is guaranteed to be zero. Syntax is as follows:

shared void *upc_reset_phase(shared void *p)

This function is useful when a function is receiving a generic argument using the shared `void *` type, but needs to ensure that the pointer, when assigned to another variable, has zero phase. For example:

```
  void func(shared void *p) {
    shared [8] char *cp;
    cp = upc_reset_phase(p);
.
.
.
```

If `p` were assigned directly into `cp`, it would keep whatever phase it might have had, leading to possibly incorrect results.

2.10 Shared Arrays

Shared arrays are arrays that are allocated in the shared address space. These elements of a shared array are distributed in a block-cyclic manner across all threads, with non-overlapping subsets of the elements having affinity to each thread. The compiler converts each reference to an element of a shared array into a remote reference to the appropriate element of the fragment of the array with affinity to another thread, performing shared memory I/O where necessary. Shared arrays are declared using the **shared** type qualifier.

C multidimensional arrays are implemented as "array of array of ... scalar." For the purposes of interpreting the block size, a shared array is considered a collection of elements of the *underlying* type, which is the first (that is, the leftmost) type that is not an array. The elements of a multidimensioned shared array can be thought of as a single-dimensioned array of elements of this underlying type, arranged in the same address order as that of the original multidimensioned array.

The elements of a shared array are distributed in a block-cyclic manner across all threads. The first block of elements has affinity to thread 0, the next block has affinity to thread 1, and so forth up to thread **THREADS** - 1, then the next block again has affinity to thread 0. Within a block, it is guaranteed that consecutive elements in the declared array are locally adjacent. It is also guaranteed that consecutive blocks with affinity to the same thread are locally adjacent.

2.10.1 THREADS Dimension

When **THREADS** is a run-time value, exactly one dimension of a shared array type (including typedefs) must be dimensioned to a multiple of **THREADS**. It may be convenient to specify a separate dimension:

```
        shared int A[3][THREADS];
```

or it may be preferable to indicate a multiplication:

```
        shared int A[3*THREADS];
```

These two examples are *not* syntactically equivalent (the first array has two dimensions, the second has one), but they do indicate the same number of underlying array elements (that is, the same number of ints).

When **THREADS** is a compile-time value (that is, when `-fthreads` is specified on the command line), no **THREADS** dimension is required when declaring a shared array. (See also Section 2.3.1.) The allocation for the array will be padded as necessary so that all threads allocate the same amount of memory. Any additional space should not be considered accessible, however. See Section 2.10.3 for more information.

If an array is declared with indefinite block size (see Section 2.5.1.1), no **THREADS** dimension is allowed, and the entire array has affinity to thread zero. For example:

```
shared [] int A[30]; // 30-element shared array,
                     // all elements with affinity to thread 0
```

2.10.2 Block Size

Block size indicates the number of underlying array elements that are grouped together on each thread. This section discusses various issues regarding proper interpretation and use of block sizes.

2.10.2.1 Block Size and Typedefs

The block size always applies to the underlying scalar type (that is, the leftmost non-array type), regardless of whether there are any typedefs involved. Consider:

```
typedef int arrayofints[12];
shared [5] int Array1[3*THREADS][12];
shared [5] arrayofints Array2[3*THREADS];
shared [5] struct { arrayofints x; } Array3[3*THREADS];
```

In the first two declarations, the block size of five applies to the `int` type. The arrays `Array1` and `Array2` are grouped into blocks of five ints on each thread. Note that, for any valid values of `i` and `j`, `Array1[i][j]` and `Array2[i][j]` will have affinity to the same thread.

In the third declaration, `Array3` is given a structure type. This structure type is the leftmost non-array type, so `Array3` is grouped into blocks of five of these structures. In this case, all the elements of `x[j]` have affinity to the same thread for a given `Array3[i]`, and `Array3[i].x[j]` will in many cases not have affinity to the same thread as `Array1[i][j]`.

Note that you can use the `upc_blocksizeof` operation to make one type's block size match that of another type:

```
#include "a_header.h"  // Defines types "Thing" and
                       // "SharedArrayOfThings"
shared [upc_blocksizeof(SharedArrayOfThings)] Thing *thingPointer;
```

The number of bytes in a block of a shared array may be calculated as:

```
        upc_blocksizeof(A) * upc_elemsizeof(A)
```

2.10.2.2 Block Size and Array Dimensions

In some cases, it is convenient to make the block size a divisor of the product of the dimensions to the right of the **THREADS** dimension:

```
shared [15] int ARR[2][7*THREADS][3][5];
```

In this example, the block size makes it convenient to think of the basic element of `ARR` as an array[3][5] of int. The block size guarantees that the array[3][5] will be at consecutive local addresses; a pointer to private can be created (see Section 2.11.4) that will reference this array correctly when subscripted. For example:

```
int (*p)[5];  // Declare a pointer to private

p = (int (*)[5])ARR[i][j];
p[a][b] = 0;
```

Out of these array[3][5] basic elements is made an array[2][7] on each thread.

Table 2–3 shows two examples of array elements distributed across threads. One example has a smooth element distribution, while the other has an uneven distribution.

Table 2–3 Example of Distribution of Array Elements Across Threads

	Column Number				
Row	**0**	**1**	**2**	**3**	**4**
shared [5] int A[3*THREADS][5]					
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	0	0	0	0	0
4	1	1	1	1	1
5	2	2	2	2	2
6	0	0	0	0	0
7	1	1	1	1	1
8	2	2	2	2	2
shared [7] int A[3*THREADS][5]					
0	0	0	0	0	0
1	0	0	1	1	1
2	1	1	1	1	2
3	2	2	2	2	2
4	2	0	0	0	0
5	0	0	0	1	1
6	1	1	1	1	1
7	2	2	2	2	2
8	2	2	0	0	0

The table shows the thread number to which each element has affinity. The value of **THREADS** is 3 for this example, thus the outer dimension of the array is 9.

Note that, for the first declaration, all threads have 15 elements, while in the second declaration thread 0 has 17 elements and the other threads have 14.

2.10.3 Size of a Shared Array

Because of the **THREADS** dimension, the size of a shared array is not a compile-time constant unless **THREADS** is a compile-time constant. In contrast, the size of a non-shared array is always a compile-time constant.

The `sizeof` operator will return the array total size, such that `sizeof(array) / upc_elemsizeof(array)` will always yield the total number of elements in the array.

Because the array is distributed across all threads, the amount of space allocated with affinity to any single thread doesn't need to be the total array size. In general, it is equivalent to the total array size divided by the number of threads. This yields a *local size*, which can be divided by the underlying element size to yield a *local number of elements*.

If the local number of elements is not an even multiple of the block size, then the grouping of the elements into blocks will cause some threads to have more elements than others. The local size as previously calculated would be too small for some of the threads. To avoid this problem, the local number of elements, for allocation purposes only, is rounded up to be a multiple of the block size. For example:

```
shared [5] int A[3][7*THREADS][11];
```

In this array, the local number of elements is 231. The total number of elements on a four-thread system would be 924. Grouped into blocks of five, there would be four extra elements that would be local to thread 0, yielding a total of 234 elements on thread 0 and 230 on the other threads. Rounding up to 235 local elements provides sufficient room for this case, while wasting one element on thread 0 and five on the other threads.

The local size of a shared array may be determined using the `upc_localsizeof` operator (Section 2.8.2).

2.11 Pointers to Shared Data

Pointers to shared data are presumed to point into arrays of shared data, which share elements that are distributed across threads and grouped in blocks. Pointer operations need to take this distribution and grouping into account. Thus, a pointer to shared data must convey more information than simply the local virtual address of the data. It must indicate to which thread the data have affinity, and it must provide enough information to allow thread switching to occur when the pointer is incremented past the end of a block. Because of these distinctions, pointers to shared data are maintained internally as structures.

2.11.1 Terminology

Pointers to shared data are typically referred to as *shared pointers*, even though the pointers themselves may or may not be shared.

Pointers themselves can be shared: a *pointer to shared to shared data* can be referred to as a *shared pointer to shared*, and a *pointer to shared to non-shared data* would be a *shared pointer to private*. Unless specified in this manner, the pointer value is not shared, regardless of whether it is pointing to shared data.

2.11.2 Implementation

The *pointer to shared structure* contains the following fields:

- `thread`, indicating the thread to which the pointed-to data have affinity;
- `phase`, indicating which element in the current block is pointed to;
- `addr`, a local virtual address for the data. Note that this virtual address may not be the same as an equivalent pointer to private; see Section 2.11.4 for details.

Values in these fields may be obtained by using the `upc_threadof` (Section 2.9.1), `upc_phaseof` (Section 2.9.2), and `upc_addrfield` (Section 2.9.3) built-in functions.

2.11.3 Restrictions

Pointers to shared data are not the same as pointers to private data and are not guaranteed to be the same size. Binary operations (comparison, subtraction) may not be performed between one pointer to shared and one pointer to private. Pointers to shared may be compared to and subtracted from other pointers to shared, but the results are meaningless if the pointers to shared point to different arrays.

When a conversion is made from one pointer to shared type to another, the phase of the resulting pointer to shared is set to zero, with one exception: if either the source or the destination is a generic pointer to shared (`shared void *` type), the phase is maintained across the conversion.

Pointers to shared with indefinite block size do not change phase as a result of pointer arithmetic operations. Such pointers to shared will always have a phase of zero.

Given an explicit or implicit conversion from one pointer-to-shared type to a different pointer-to-shared type, the phase value of the result will be zero, unless either the old or the new type is `shared void *`. This phase-saving characteristic of the shared `void *` type allows the phase value to be passed along to a function. For example:

```
void genfunc(shared void *p);

shared [8] int A[8*THREADS];

genfunc(&A[6]);  /* Phase of argument is 6 */

void genfunc(shared void *p) {
  i = upc_phaseof(p);  /* i gets the value 6 */
  ...
}
```

When assigning a shared `void *` to a different type, take care to ensure that the phase is appropriate to the destination pointer. For example:

```
shared [8] int A[8*THREADS];
shared [3] int *p;
shared void *svp;

svp = &A[6];  /* phase is 6 */
p = svp;  /* phase of 6 is invalid for p */
```

Generic pointers to shared (`shared void *` type) may be compared to other pointers to shared for equality or inequality, but they cannot be compared otherwise.

2.11.4 Casting to Pointers to Private

Pointers to shared may be cast to pointers to private only if the pointer to shared refers to data with affinity to current thread. The result will be a valid local pointer referring to the same data. This operation is useful if there is a large block of data on the current thread that may be more efficiently manipulated through pointers to private. The resulting pointer will correctly step through the current array element block when incremented.

Note that:

- Pointers to private cannot be cast to pointers to shared.
- If a pointer to shared does not refer to the current thread, casting that pointer to shared to a pointer to private will not produce an error, but the address will not be valid.

See also Section 6.1.1 for some issues regarding pointers to small data types.

2.12 Statements

The following sections describe UPC statements.

2.12.1 upc_forall Statement

UPC provides the **upc_forall** statement as a way to distribute loop iterations across the executing threads easily.

2.12.1.1 Semantics

In other parallel programming models, for example OpenMP, the goal of parallel looping constructs is to partition the iteration space efficiently among the threads. Each thread executes some subset of the entire iteration space. Different partitionings are possible for such constructs, for example blocked, round robin, guided self-scheduling, etc. However, many such looping constructs suffer major performance degradations because the data referenced in the iterations can predominantly reside on a different thread from the one executing the loop body for that iteration.

In marked contrast, the UPC **upc_forall** construct takes exactly the opposite approach. Instead of controlling which iterations are executing on which threads, the UPC **upc_forall** statement, conceptually, executes all the specified iterations on each thread. This obvious redundancy is balanced, however, by the ability of the programmer to suppress execution of the loop body for those iterations whose dominant data affinity is to a different thread.

Thus, although each thread samples the entire iteration space, as with an ordinary **for** loop, the programmer has the opportunity to partition the space of *executed* iterations on each thread in the most efficient manner possible. Since the cost of referencing remote memory typically dwarfs the cost of local iteration control expression evaluation, the UPC **upc_forall** statement actually provides a better tradeoff for overall performance than the more typical parallel looping constructs.

Furthermore, since the affinity expression in a UPC **upc_forall** construct can be either an integer or a pointer to shared, the programmer can easily adjust the partitioning of the executed iterations so as to globally minimize the number of remote shared memory references across the entire iteration space of the loop on all threads simultaneously. Such a capability is difficult, if possible at all, on other parallel loop iteration control constructs and is a major contributor to the efficiency of UPC over other parallel programming models.

Finally, the HP UPC compiler is capable, if the behavior of the expression can be determined at compile time, of eliminating the wasted control calculations entirely. This optimization is under the control of the `-O` compiler option. Thus, HP UPC provides the best aspects of both models.

2.12.1.2 Syntax

The **upc_forall** statement adds a fourth control expression for the loop, an *affinity expression* indicating whether the current thread should execute the loop body. For example:

```
shared int A[3*THREADS], B[3*THREADS], C[3*THREADS];
upc_forall (i = 0; i < 3*THREADS; i++; &A[i])
  A[i] = B[i] + C[i];
```

In this example, the affinity expression indicates that the body of the loop will only run on the thread where each `A[i]` has affinity. Thus, in this case, each iteration of the loop only references data with affinity to the running thread. Note that this property is true regardless of the block size used to declare the arrays.

A **upc_forall** loop may only be exited by completing the loop. Exiting the loop via a `goto` statement, a `return` statement, or a `break` statement is not allowed. There are cases the compiler cannot detect that could still cause the loop to exit incorrectly, such as a `longjmp` from within a called function; they must be avoided as well.

The **upc_forall** statement is considered a parallel statement, such that all iterations of the loop executed by different threads are independent and may be executed in any order. The compiler assumes that there are no loop-carried dependencies among the iterations performed by different threads (that is, that no iteration executed by one thread depends on a calculation performed in an iteration executed by a different thread), and may arrange code accordingly. This assumption is made regardless of the content of an affinity expression.

2.12.2 Barrier Statements

Barriers provide synchronization points among the executing threads. There are two actions taken by each thread: notify the other threads that the barrier has been reached; and wait for the other threads to reach the barrier before proceeding.

UPC provides three statements that implement barriers. Each statement takes an optional integer expression; this expression is a reference tag for the barrier and must evaluate to the same value for all threads. The run-time system will verify that all threads reach a barrier, and that all have specified the same value.

If no expression is present, the value will not be checked against those for other threads, and the barrier statement will be considered an acceptable match for any other barrier statement, with or without a value, executing on other threads.

The three statements are:

upc_notify
Notifies all other threads that the barrier has been reached. Before issuing the notification, ensures that all writes of shared data have completed.

upc_wait
Waits for all other threads to reach the **upc_notify** statement. The programmer is allowed to specify execution of statements not involving any references to shared data between the execution of the **upc_notify** and **upc_wait** statements.

upc_barrier
Performs the combined functions of a **upc_notify** and a **upc_wait**. The **upc_barrier** statement is used when there is no need to do any additional local-only work between the notification and the wait.

There is an implicit **upc_fence** before a **upc_notify** and after a **upc_wait**.

Example 2–4 Barrier Statements

```
upc_barrier;            // Notify+wait, default tag of 0
upc_notify 2;   // Notify thread has reached point 2
...                 // Do some additional local-only work
upc_wait 2;     // Waits for other threads to get here
```

Example 2–5 Value Matching for Barrier Statements

```
Thread 0         Thread 1

upc_barrier 1;       barrier 1;  // Values match
upc_barrier;         barrier 2;  // Null value matches value
upc_barrier 3;       barrier 4;  // Values do not match, error
```

There must always be a **upc_wait** following (at some point) a **upc_notify** with the same tag value. There cannot be a **upc_wait** with a different tag value, nor another **upc_notify** statement, following the **upc_notify**. Similarly, any **upc_wait** statement must be preceded by a **upc_notify** with the same tag value; two **upc_wait** statements, or a **upc_barrier** followed by a **upc_wait** statement, are not allowed.

UPC does not provide a mechanism for forcing a subset of threads to synchronize at a barrier. All threads must perform the synchronization.

2.12.3 Nested upc_forall Loops

To ensure coverage of all iterations of a nested series of loops, only one affinity expression will be honored at a time. This rule affects nested **upc_forall** loops, including those reached by function calls from within other **upc_forall** loops. If such a loop is entered while another one is active, the affinity expression will not be evaluated, and the body of the loop will be executed without further conditions.

If a **upc_forall** loop is physically inside another **upc_forall** loop, and both have non-null affinity expressions, the affinity expression for the inner loop is meaningless, since all iterations would be executed anyway due to the presence of the outer loop. The compiler will detect this case and ignore the inner loop's affinity expression.

2.12.4 upc_fence Statement

The **upc_fence** statement ensures that all previously issued stores to shared memory by this thread are completed before any new fetch or store operations are performed. The **upc_fence** statement provides the same synchronization as a reference to a strict variable, without requiring a specific strict variable reference. The following sequences are equivalent:

```
relaxed shared int x;    strict shared int y;

upc_fence;
t = x;                   t = y;

upc_fence;
x = 2;                   y = 2;
upc_fence;
```

Note that, in the preceding example, the **upc_fence** statements could be replaced by **upc_barrier** statements if that section of code is executed on all threads. However, the **upc_fence** statement is significantly faster than the **upc_barrier** statement, because it does not require all the threads to wait for the other threads to reach a communal synchronization point. Use a **upc_fence** statement (or a strict reference) to enforce data access ordering when the ordering is what is needed.

3 Library Functions

This chapter describe UPC library functions.

3.1 Locking Functions

Locks are of type `upc_lock_t`, which is an opaque type. Locks are allocated dynamically with `upc_all_lock_alloc` or `upc_global_lock_alloc`. Only pointers to locks may be declared. Locks should be freed with a call to `upc_lock_free` after they are no longer needed. Example 3–1 provides some sample code using the locking functions. The use of locks does not imply any other synchronization. Strict references and fences will still need to be used to ensure that data are available when needed.

Example 3–1 Using Locks

```
#include "upc_strict.h"

shared unsigned long counter;    /* Affinity to thread 0. */
upc_lock_t *lock1;

#include <time.h>

void main (int argc, char *argv[])
{
 time_t t1, t2;
 int i;
 int default_count = 50000;
 int count;
 int tmp;
 int max;
 int print = 0;
 unsigned long lcounter;

 if (argc > 1) {
    for (i = 1; i < argc; ++i) {
    count = atoi(argv[i]);
    }
    } else {
    count = default_count;
    }
    printf("%d: count=%d\n", MYTHREAD, count);
 lock1 = upc_all_lock_alloc();
 time(&t1);
upc_barrier 0;
 if (MYTHREAD == 0) {
   upc_lock(lock1);
   counter = 0;
   upc_unlock(lock1);
  }
  upc_barrier 9;
```

(continued on next page)

Example 3–1 (Cont.) Using Locks

```
 /* Have each thread increment counter count times.
 * Use the lock to guard the updates.
 */
 max = 0;
 for (i = 0; i < count; ++i) {
   upc_lock(lock1);
   counter += 1;
   upc_unlock(lock1);
 }
upc_barrier 0;
   time(&t2);

 printf("%d: at end, counter = %ld, %d secs\n",
       MYTHREAD, counter, t2-t1);
 if (counter != (count * THREADS))
   printf("ERROR: counter should be %d\n",
   (count * THREADS));
 }
}
```

3.1.1 The upc_lock Function

Synopsis

`void upc_lock(upc_lock_t *)`

Description

Obtains ownership of the specified lock. If ownership is not immediately available, execution is suspended until the lock is obtained. No deadlock detection is attempted; deadlock avoidance is the responsibility of the programmer.

3.1.2 The upc_lock_attempt Function

Synopsis

`int upc_lock_attempt(upc_lock_t *)`

Description

Attempts to get ownership of the specified lock. If ownership is not immediately available, return 0 (FALSE), otherwise it returns (1) TRUE.

3.1.3 The upc_unlock Function

Synopsis

`void upc_unlock(upc_lock_t *)`

Description

Releases ownership of the specified lock.

3.1.4 The upc_all_lock_alloc Function

Synopsis

`upc_lock_t *upc_all_lock_alloc(void)`

Description

Allocates a lock (which is shared data), and returns a pointer to it. Any lock allocated with this function should be deallocated via `upc_lock_free` when no longer needed.

This function is called simultaneously by all threads. A single lock is allocated, and all threads receive a pointer to it. A barrier statement is implicitly executed by calling this function.

Note

A **upc_barrier** is not required by the language and may be removed in a future version.

3.1.5 The upc_global_lock_alloc Function

Synopsis

```
upc_lock_t *upc_global_lock_alloc(void)
```

Description

Allocates a lock (which is shared data), and returns a pointer to it. Any lock allocated with this function should be deallocated via `upc_lock_free` when no longer needed.

This function is called by a single thread. A single lock is allocated, and only the calling thread receives a pointer to it, unless the returned lock pointer is stored in a shared variable. If multiple threads call this function, multiple locks will be allocated.

3.1.6 The upc_lock_free Function

This function is called by a single thread.

Synopsis

```
void upc_lock_free(upc_lock_t *)
```

Description

Frees the memory associated with a lock that was previously allocated via `upc_all_lock_alloc` or `upc_global_lock_alloc`.

3.1.7 Locking Algorithms

Two locking algorithms are available:

- FAIR
- GREEDY (default)

You can select the algorithm using the `UPCRTS_LOCK_TYPE` environment variable (see Chapter 7).

The FAIR setting uses a fair (no starvation) algorithm. GREEDY locking is faster than FAIR locking but does not guarantee that a waiting thread will eventually be granted a lock.

3.2 Memory Transfer Functions

UPC provides several functions to facilitate copying to or from shared data.

Note

Note that, for all of these functions, it is assumed that all the shared data being transferred have **affinity to the same thread** and **are locally contiguous**.

3.2.1 The upc_memcpy Function

Synopsis

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

Description

Copies shared data to another shared location. Both `src` and `dst` point to locally contiguous shared data areas of at least `num_bytes` bytes.

3.2.2 The upc_memget Function

Synopsis

```
void upc_memget(void *dst, const shared void *src, size_t num_bytes)
```

Description

Copies shared data to a non-shared location. `dst` points to a locally contiguous shared data area of at least `num_bytes` bytes.

3.2.3 The upc_memput Function

Synopsis

```
void upc_memput(shared void *dst, const void *src, size_t num_bytes)
```

Description

Copies non-shared data to a shared location. `src` points to a locally contiguous shared data area of at least `num_bytes` bytes.

3.2.4 The upc_memset Function

Synopsis

```
void upc_memset(shared void *dst, int cval, size_t num_bytes)
```

Description

Sets `num_bytes` bytes of shared data to the value `cval`. `dst` points to a locally contiguous shared data area of at least `num_bytes` bytes.

3.3 Allocations

This section describes the functions for dynamically allocating and freeing shared data. Such data may have affinity to any thread, or (in the case of arrays) may be distributed across all threads.

Example 3–2 Examples of Dynamic Allocation of Shared Data

```
shared [5] int *p1, *p2, *p3;
shared [5] int * shared p4, * shared p5;

/* Allocate 25 elements per thread, with each thread doing its
   portion of the allocation. */
p1 = (shared [5] int *)upc_all_alloc(25*THREADS), sizeof(int));

/* Allocate 25 elements per thread, but just run the allocation
   on thread 5. */
if (MYTHREAD == 5)
   p2 = (shared [5] int *)upc_global_alloc(25*THREADS), sizeof(int));

/* Allocate 5 elements only on thread 3. */
if (MYTHREAD == 3)
   p3 = (shared [5] int *)upc_alloc(5, sizeof(int));

/* Allocate 25 elements per thread, just run the allocation
   on thread 4, but have the result be visible everywhere. */
if (MYTHREAD == 4)
   p4 = (shared [5] int *)upc_global_alloc(25*THREADS), sizeof(int)):

/* Allocate 5 elements only on thread 2, but have the result
   visible on all threads. */
if (MYTHREAD == 2)
   p5 = (shared [5] int *)upc_alloc(5, sizeof(int));
```

Notes on Example 3–2:

- The variable `p1` is local to each thread. The call to `upc_all_alloc` will return the same value on each thread, a pointer to shared referring to element 0 of the allocated array. Each thread will access the same elements when the pointer is subscripted.
- The variable `p2` is also local to each thread; in this case, though, only thread 5 fills it in. On all other threads, `p2` is uninitialized and therefore invalid.
- When `p2` is subscripted (on thread 5), it will reference data that have affinity to various threads, even though those other threads do not have a valid pointer to that data.
- The variable `p3` is only valid on thread 3, because it is only filled in on thread 3. The shared data pointed to by `p3` have affinity to thread 3.
- The variable `p4` is a *shared pointer to shared*. When thread 4 fills in the value, all other threads can see and use it. The value of `p4` is a pointer to shared referring to element 0 of the allocated array; `p4` itself is a shared variable with affinity to thread 0.
- The variable `p5` is a *shared pointer to shared*. Thread 2 will fill in the value of `p2` with a pointer to data that have affinity to thread 2, and all other threads will be able to see the pointer value.

3.3.1 The upc_alloc Function

Synopsis

`shared void *upc_alloc(size_t num_bytes)`

Description

Allocates a shared region of `num_bytes` bytes, residing entirely on the current thread, and returns a pointer to that array. The allocated array will be shared data and thus will be visible to all threads.

3.3.2 The upc_global_alloc Function

Synopsis

```
shared void *upc_global_alloc(size_t num_elems, size_t elem_size)
```

Description

Allocates a shared array of `num_elems` elements, each of size `elem_size`, spread across all threads, and returns a pointer to that array. The array can be referenced in the normal block cyclic fashion of shared arrays. A single thread will call this function, but the resulting allocation will take place across all threads.

The size of the memory pool for global allocation can be application dependent. Refer to Section A.3 for information about adjusting pool size.

3.3.3 The upc_all_alloc Function

Synopsis

```
shared void *upc_all_alloc(size_t num_elems, size_t elem_size)
```

Description

Allocates a shared array of `num_elems` elements, each of size `elem_size`, spread across all threads, and returns a pointer to that array. The array can be referenced in the normal block cyclic fashion of shared arrays. *All* threads must call this function. All threads must specify the same array size. A barrier is implicitly executed by calling this function.

Note

A **upc_barrier** is not required by the language and may be removed in a future version.

3.3.4 The upc_free Function

Synopsis

```
void upc_free(shared void *p)
```

Description

Frees shared memory that was allocated by any of the allocation functions:

```
upc_all_alloc
upc_global_alloc
upc_alloc
```

It is called by a single thread, which need not be the thread that allocated the memory. It should be called with an argument of a pointer to the base of the allocated memory. The entire allocated amount is freed.

3.3.5 The upc_global_exit Function

Synopsis

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

Description

Causes all threads to exit immediately. It is called by a single thread. The **status** argument is used as the exit status code.

3.4 Collective Functions

Descriptions of the collective functions are derived largely from Section 7 of the Version 1.2 UPC language specification.

Collective functions perform useful data manipulations in parallel. All of the functions defined in this section must be called by all threads, and all of the arguments must have the same value on all threads at the corresponding call.

All source modules that contain calls to collective functions must include the header `upc_collective.h`.

3.4.1 Synchronization Options

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

Heuristically, the names of the flags indicate how much synchronization the programmer wants the collective function to perform. NO indicates the function should perform no data synchronization; the programmer has taken care of it. ALL indicates that the function should synchronize all data; the programmer has done none. MY indicates that the programmer has taken care of synchronization of data for the current thread only, and the function should synchronize any data with affinity to other threads. IN and OUT specify the synchronization mechanism for input data and output data, respectively.

3.4.2 Relocalization Operations

The following are relocalization operations.

3.4.2.1 The upc_all_broadcast Function

Synopsis

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

Description

The `upc_all_broadcast` function copies an area of memory with affinity to a single thread to an area of shared memory on each thread. The number of bytes in each area is `nbytes`. The function treats the `src` pointer as if it pointed to a shared memory area with the type:

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

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

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

3.4.2.2 The upc_all_scatter Function

Synopsis

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

Description

The upc_all_scatter function copies the ith block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the ith thread. The number of bytes in each block is nbytes. The function treats the src pointer as if it pointed to a shared memory area with the type:

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

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

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

3.4.2.3 The upc_all_gather Function

Synopsis

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

Description

The upc_all_gather function copies a block of shared memory that has affinity to the ith thread to the ith block of a shared memory area that has affinity to a single thread. The number of bytes in each block is nbytes. The function treats the src pointer as if it pointed to a shared memory area of nbytes bytes on each thread and therefore had type:

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

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

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

3.4.2.4 The upc_all_gather_all Function

Synopsis

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

Description

The upc_all_gather_all function copies a block of memory from the shared memory area with affinity to the ith thread to the ith block of a shared memory area on each thread. The number of bytes in each block is nbytes. The function treats the src pointer as if it pointed to a shared memory area of nbytes bytes on each thread and therefore had type:

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

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

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

3.4.2.5 The upc_all_exchange Function

Synopsis

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

Description

The `upc_all_exchange` function copies the ith block of memory from each a shared memory area that has affinity to thread j to the jth block of a shared memory area that has affinity to thread i. The number of bytes in each block is `nbytes`. The function treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nbytes * THREADS` bytes on each thread and therefore had type:

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

3.4.2.6 The upc_all_permute Function

Synopsis

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

Description

The `upc_all_permute` function copies a block of memory from a shared memory area that has affinity to the ith thread to a block of a shared memory that has affinity to thread perm[i]. The number of bytes in each block is `nbytes`. `perm[0..THREADS-1]` must contain THREADS distinct values: 0, 1, ..., THREADS-1. The function treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type:

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

3.4.3 Computational Operations

This section describes the computational collective functions.

A variable of type `upc_op_t` can have the following values:

Variable	Value
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.

The operations represented by a variable of type upc op t (including userprovided operators) are assumed to be associative; a reduction or a prefix reduction whose result is dependent on the order of operator evaluation will have undefined results. Furthermore, all of these operations (except those provided using **UPC_NONCOMM_FUNC**) are assumed to be commutative; a reduction or a prefix reduction (using operators other than **UPC_NONCOMM_FUNC**) whose result is dependent on the order of the operands will have undefined results.

3.4.3.1 The upc_all_reduce and upc_all_prefix_reduce Functions

Synopsis

```
#include <upc.h>
#include <upc collective.h>
void upc_all_reduceT(shared void * restrict dst,
     shared const void * restrict src, upc_op_t op, size_t nelems,
     size_t blk_size, TYPE(*func)(TYPE, TYPE), upc_flag_t flags);
void upc_all_prefix_reduceT(shared void * restrict dst,
     shared const void * restrict src, upc_op_t op, size_t nelems,
     size_t blk_size, TYPE(*func)(TYPE, TYPE), upc_flag_t flags);
```

Description

These prototypes represent the 22 variations of the `upc_all_reduceT` and `upc_all_prefix_reduceT` functions where T and TYPE have the following correspondences:

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

On completion of the `upc_all_reduce` variants, the value of the TYPE shared object referenced by `dst` is

```
src[0] op src[1] op ... op src[nelems-1]
```

On completion of the `upc_all_prefix` reduce variants, the value of the TYPE shared object referenced by `dst[i]` is

```
src[0] op src[1] op ... op src[i]
```

for 0 <= i <= nelems-1.

4 Compiling and Running UPC Programs

This chapter describes the upc command, compiler options available to control UPC-specific behavior, and techniques for compiling, debugging, and running UPC programs.

4.1 The upc command

The upc command compiles UPC language source into machine-readable instructions. The desired output is specified with an option on the command line, and can be object files, translated C language source files, or symbolic assembly language.

The compiler produces one object file for each file compiled. If the linker is called, and only one source file is specified on the command line, the single object file is deleted after the linking operation.

The upc command invokes the UPC compiler, and possibly other components of the compilation system, depending on the specified command options and files. When you start the compiler, it normally performs the following tasks:

- Preprocessing
- Compiling
- Linking

Command syntax is

```
upc [ option ] ...  file ...)
```

Use the **-c** option to stop compiler operation before linking, resulting in one or more output files with the .o suffix. Use the **-E** option to run only the preprocessor.

The detection of an error in one source file does not affect compilation of other source files specified on the same command line.

The compiler recognizes the following file name suffixes as having special significance:

.a	Linker library
.c	UPC source code, to be compiled with the HP UPC compiler
.i	C source code already processed by the preprocessor
.lis	Source code listing from the **-source_listing** option. On HP-UX, **-source_listing** is transformed to **-K**, and a .int.c file is created.
.o	Linker object module
.sl	Shared object (shared library) on HP-UX
.so	Shared object (shared library) on Tru64 UNIX
.upc	UPC source code, to be compiled with the HP UPC compiler

The UPC compiler recognizes the command options described in the Options section. Some of the options affect the behavior of the preprocessor. The preprocessor is invoked through the `upc` command for UPC files, and is not invoked with a separate command. Any options not recognized by the UPC compiler are assumed to be linker options, and are passed through to the linker.

To facilitate setting default compiler flags, you can create an optional configuration file named comp.config.

- The comp.config file allows system administrators to establish a set of compilation flags that are applied to compilations on a system-wide basis. The compiler flags in **comp.config** must be specified on a single line. The **comp.config** file is located in the compiler target directory, `/usr/lib/cmplrs/upc` on Tru64 UNIX and `/opt/upc/bin` on HP-UX and XC Linux. You can use the **-comp.config** option to specify a different directory.
- The comp.config file can contain two distinct sets of compilation flags separated by a single vertical bar (|). The flags before the vertical bar are known as prologue flags and the flags after the bar are known as epilogue flags. The comp.config file can begin or end with a vertical bar, or have no vertical bar at all. If no vertical bar is present, the flags are treated as prologue flags by default. Any vertical bar found after the first vertical bar is treated as whitespace.

Compiler flags are processed in the following order during a compilation:

1. comp.config prologue flags
2. command line flags
3. comp.config epilogue flags

If **-v** is specified on the command line, the contents of **comp.config**, if present, is displayed.

If the environment variable TMPDIR is set, the value is used as the directory to hold any temporary files rather than the default `/tmp` directory.

You can use the the DEC_CC environment variable to establish a set of compilation options that are applied to subsequent compilation on a per-user basis. See Section 4.6 for details.

On Tru64 UNIX, if the environment variable `RLS_ID_OBJECT` is set and a link occurs, the value is used as the name of an object to link. The `RLS_ID_OBJECT` variable is always the last object specified to the linker.

4.2 Macro Names

The preprocessor recognizes the following standard macro names, which cannot be redefined or undefined. You can the upc **-v** option to display all predefined macros for a given compilation.

__LINE__	The current line number (a decimal integer).
__FILE__	The current file name (a character string).
__DATE__	The source file's compilation date (a character string literal of the form "Mmm dd yyyy", where the month names are the same as those generated by the asctime function, and the first character of "dd" is a space character if the value is less than 10.)
__TIME__	The source file's compilation time (a character string literal of the form "hh:mm:ss"), as in the time generated by the asctime function.

In addition, macro names that begin with two leading underscore characters (_ _) or a single underscore character followed by a capital letter (for example, **_WCHAR_T**) are reserved for the implementation.

Name	Function
_ _alpha[1]	System has an Alpha processor
_ _alpha_ _[1]	System has an Alpha processor
_ _ALPHA[1]	System has an Alpha processor
_ _arch64_ _[1]	System with 64-bit architecture
_ _compaq_ _[1]	Compiler is provided by Compaq Computer Corp.
_ _COMPAQ_UPC_VER[1]	Indicates the version of HP UPC being used; a value of 20020001 means V2.2-001
_ _DECC[1]	Compiler is compatible with Compaq C
_ _digital_ _[1]	System identification name on Tru64 UNIX
_ _EDG_ _	Compiler is built with EDG front end technology
_ _EDG_VERSION_ _	Version number of EDG front end technology
_ _hp_ _	Compiler is provided by Hewlett-Packard Company
_hppa[2]	Defined for compatibility with HP-UX system header files
_hpux[2]	Defined for compatibility with HP-UX system header files
_ _HP_UPC_VER	Indicates the version of HP UPC being used; a value of 20020001 means V2.2-001
_HPUX_SOURCE[2]	Defined for compatibility with HP-UX system header files
_ _IEEE_FLOAT[1]	The compiler supports IEEE floating point
_ _LANGUAGE_C_ _[1]	Compiler translates C language constructs and keywords
_ _LANGUAGE_UPC_ _	Compiler translates UPC language constructs and keywords
_LONGLONG[1]	Compiler supports "long long" type declarations
_LP64[2]	Defined for compatibility with HP-UX system header files
_ _LP64[2]	Defined for compatibility with HP-UX system header files
_ _LP64_ _[2]	Defined for compatibility with HP-UX system header files
_ _osf_ _	System is a Tru64 UNIX system
_HPUX_SOURCE[2]	Defined for compatibility with HP-UX system header files
_ _PRAGMA_ENVIRONMENT[1]	The compiler supports pragma environment
_ _SIGNED_CHARS_ _[2]	Ensures that signed chars are the default

[1]Tru64 UNIX only

[2]HP-UX only

Name	Function
__STDC__	Compiler supports ANSI C language constructs
__STDC_HOSTED__	C99 predefined macro. If set, indicates a hosted implementation.
__STDC_VERSION__	Version number of ANSI C support, in string form
__SYSTYPE_BSD__	Compiler supports BSD type system headers
__unix, __unix__	System is a UNIX system
__UPC__	Defined to 1 to indicate that the program is being compiled as a UPC program
__UPC_DYNAMIC_THREADS__	Defined to 1 when the program is being compiled in the dynamic THREADS environment, and is undefined otherwise
UPC_MAX_BLOCK_SIZE	Largest value that may be specified for a block size
__UPC_STATIC_THREADS__	Defined to 1 when the program is being compiled in the static THREADS environment, and is undefined otherwise
__UPC_VERSION__	Indicates the language specification version supported by the product. The version is given by date of acceptance. For version 1.1.1, approved in October 2003, the value of `__UPC_VERSION__` is 200310L.
UPCRTS_WIDE_SHARED_POINTER[2]	128-bit representation of shared pointer
_WCHAR_T	wchar_t is a keyword
__X_FLOAT[1]	System has default long double size of 128 bits

[1]Tru64 UNIX only

[2]HP-UX only

These names are useful in preprocessor **#ifdef** and **#if defined** directives to isolate code intended for one of the particular cases. The names can be used anywhere you use other defined names, including within macros.

4.3 Compiler Options

Table 4–1 lists the C compiler options that are supported by HP UPC on all platforms. platforms. For detailed information about these options, see the C documentation or the reference page for C (cc(1)).

Table 4–1 C Compiler Options Supported on All Platforms

Group	Options
Output file control	-c, -o, -S, -K
Preprocessor control	-C, -D, -I, -E, -M, -P, -U
Language dialect accepted	-std
Suppress linking	-c
Debug symbol table generation	-g
Control of error messages	-w, -w0
Driver behavior	-v
Compiler optimization	-O, -assume
Integer control	-signed, -unsigned

Command option descriptions are grouped as follows:

- UPC-Specific Options
- General Options
- Language Mode Options
- Message Information and Control Options
- Optimization Options
- Preprocessor Options

4.3.1 UPC-Specific Options Supported on All Platforms

The following options are supported on all platforms.

4.3.1.1 The -frelaxed Option

The **-frelaxed** option specifies that the compiler is allowed to overlap communication operations (fetches and stores) of shared quantities. When using the **-frelaxed** option), the programmer must explicitly insure the consistency of shared data using barrier class statements. See **-fstrict** for more information on this topic. In the absence of an **-frelaxed** or **-fstrict** option, the "strict" or "relaxed" type qualifiers and #pragma upc determine the memory consistency for shared quantities.

4.3.1.2 The -fstrict Option

The **-fstrict** option specifies that each inter-processor communication (fetch or store) using shared quantities will be completed prior to beginning any later such communication. See **-frelaxed** for more information on this topic. In the absence of an **-fstrict** or **-frelaxed** option, the "strict" or "relaxed" type qualifiers and #pragma upc determine the memory consistency model for shared quantities.

4.3.1.3 The -fthreads *num* Option

num is an integer constant. The **-fthreads** option specifies that the UPC compiler should compile the source programs with the assumption that the UPC language symbol **THREADS** will always have the value of *num* for all executions of the program. Specifying this option defines the macro `__UPC_STATIC_THREADS__`.

4.3.2 UPC-Specific Options Supported Only on Tru64 UNIX

The following options are supported only on Tru64 UNIX systems.

4.3.2.1 The -narrow Option

Specifes a pointer to shared structure size of 64 bits (the default size). Use the **-wide** option to specify a size of 128 bits. All modules must be compiled using the same size.

4.3.2.2 The -assume [no]nested_upc_forall Option

This option specifies that all **upc_forall** statements are to be presumed possibly nested or never nested, respectively. **-assume nested_upc_forall** is equivalent to placing a `#pragma upc nested` at the beginning of the compilation unit, and is the default setting. **-assume nonested_upc_forall** is equivalent to putting a `#pragma upc unnested` at the beginning of the compilation unit.

4.3.2.3 The -[no]smp Option

Informs the compiler that the program will be run in a fully shared memory environment, allowing the compiler to make further optimizations in shared data access. This option also implies `-smp_local`.

4.3.2.4 The -[no]smp_local Option

Enables an optimized use of shared memory when the program is run on a cluster of symmetric multiprocessor (SMP) systems. When this option is specified, shared data on different threads running on the same SMP node are accessed via local memory, without going through the context of the other thread.

Specify **-nosmp_local** to disable the optimization.

4.3.2.5 The -wide Option

HP UPC uses a structure to contain pointer to shared information. The size of the pointer to shared structure is normally 64 bits. Alternatively, you can select a 128-bit pointer to shared structure by specifying the `-wide` option. Use the **-narrow** option to reset the size to 64 bits. All modules must be compiled using the same size. Specifying **-wide** defines the macro `_UPCRTS_WIDE_SHARED_POINTER`.

Note that wide shared pointers are the default on HP-UX and XC Linux systems.

The **phase** field of the pointer to shared structure places limits on the allowable block size for shared data. The values for block sizes for shared data must not be too large to fit into this field, which for the narrow pointer to shared structure is 10 bits, thereby permitting block sizes of up to 1024 array elements. For the wide pointer to shared structcture, the field is 32 bits, allowing block sizes of more than four billion elements.

A disadvantage of the wide pointer to shared structure is the overhead involved in passing pointers to shared to functions, or receiving pointers to shared returned from functions. In addition, you must be careful to ensure that modifications to shared pointers to shared are done atomically.

4.3.3 General Options Supported on All Platforms

The `upc` command supports the following general options on all platforms.

-B*string*
Specifies a suffix to add to the normal names of any components specified with the **-t** option. If *string* is omitted, the usual component names are used.

-c
Suppress the linking phase of the compilation and force production of one or more object file(s) with the .o suffix. If the file name argument already has the .o suffix, do nothing at all.

-comp.config *directory*
Specifies a directory for the comp.config file. Normally, the file is stored in the compiler target directory. This option is interpreted only if specified on the command line.

-g
Produce symbol table information for full symbolic debugging, but do not perform optimizations that limit full symbolic debugging. Same as **-g2**. Using the **-g2** option resets the optimization level to -O0.

-h *dir*
Use the specified directory instead of the directory where the name specified by the **-t** flag is normally found.

-K
On HP-UX systems, controls preservation of the `.int.c` file.

-o *file*
Name the final output file, rather than use the default a.out. The given name is used regardless of the type of file (executable, object, or other type).

-S
Compile the specified source programs and produce symbolic assembly language in corresponding files with a .s suffix. Do not assemble these files. If the file name argument already has a .s suffix, do nothing at all.

-source_listing
Produce a source code listing file of the same name as the source file, but with the suffix .lis. On HP-UX, this option is transformed to **-K**, thus preserving the intermediate C source in the *filename*.int.c file.

-unsigned
Cause all `char` declarations to have the same machine representation and value set as unsigned char declarations. The **-unsigned** option overrides **-signed** if both are specified on the command line.

-v
Print the names of compiler phases as they execute, along with their arguments and input and output files. On Tru64 UNIX, this option also prints resource usage in the following format: user time, system time, total elapsed time, percentage use of CPU cycles.

-V
Enable printing of the compiler version.

-w
Suppress warning and informational level compiler messages, but not back-end messages. Same as **-w2**.

-w0
Display all levels of compiler messages.

-w1
Suppress information-level compiler messages. This is the default.

-w2
Suppress warning- and informational-level compiler messages, but not back-end messages. Same as **-w**.

-W*c*[*c*...],*arg1*[,*arg2*...]
Passes the argument, or arguments (*arg1*), to the compiler pass, or passes (*c*[*c*...]). Each *c* character can be one of the following: **[abflLnqQryz]** (see the **-t** option for an explanation). The *c* variable selects the compiler pass in the same way as the **-t** option.

-writable_strings
Make string literals writable.

4.3.4 General Options Supported Only on Tru64 UNIX

The `upc` command supports the following options only on Tru64 UNIX systems.

-call_shared
Produce a dynamic executable file that uses shareable objects during run-time. This is the default. The loader uses shareable objects to resolve undefined symbols. The run-time loader is invoked to bring in all required shareable objects and to resolve any symbols that remained undefined during static link time.

-[no]compress
-compress causes the output object file file to be produced in compressed object file format, resulting in a substantially smaller object file. To produce uncompressed files, specify **-nocompress**. The default is **-compress**.

-denorm
Force denormalized constant numbers to zero when IEEE support is enabled.

-error_limit *n*, -noerror_limit
Specify the number of errors allowed before compilation stops for a single source file. The default value is 30.

-fast
The **-fast** option defines the following compiler options and symbols to improve run-time performance. You can adjust the optimizations by specifying the negation of any given option after the **-fast** option.

```
-ansi_alias, -assume trusted_short_alignment,
[no]nomath_errno, -D_INTRINSICS, -D_INLINE_INTRINSICS,
-fp_reorder, -intrinsics, -O3, -readonly_strings,
-gen_feedback
```

The `-D_INTRINSICS` and `-D_INLINE_INTRINSICS` options are available only on Tru64 UNIX.

The **-fast** option can produce different results for floating-point arithmetic and math functions, although most although most programs are not sensitive to these differences. The **-fast** option is passed as is to the HP-UX C compiler for interpretation.

-fprm *option*
Specify rounding mode. The following options are supported:

c	Round toward 0 (chop).
d	Set rounding mode for IEEE floating-point instructions dynamically, as determined from the contents of the floating-point control register. The dynamic rounding mode can be changed or read at execution time by a call to write_rnd(3) or read_rnd(3). If you specify this option, the IEEE floating-point rounding mode defaults to round to nearest.
m	Round toward minus infinity.
n	Set normal rounding mode (unbiased round to nearest). This is the default.

This option also defines the macro **__FLT_ROUNDS**

-fptm *option*
Specify trapping mode. Only one option can be used on a command line. The following options are supported:

n	Generate instructions that do not trigger floating-point underflow or inexact trapping modes. Any floating point overflow, divide-by-zero, or invalid operation will unconditionally generate a trap. This is the default.
u	Generate intstructions that trap floating-point underflow, overflow, divide-by-zero, and invalid operations.

-g0
Do not produce symbol table information for symbolic debugging. This is the default.

-g1
Produce symbol table information for accurate, but limited, symbolic debugging of partially optimized code.

-g2
Produce symbol table information for full symbolic debugging, but do not perform optimizations that limit full symbolic debugging. Same as **-g**. This resets the optimization level to -O0.

-g3
Produce symbol table information for full symbolic debugging of fully optimized code. This option can affect debugger accuracy.

-G *num*
Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. The *num* argument is interpreted as a decimal number. If *num* is zero, data is not accessed from the global pointer. The default value for num is 8 bytes.

-H*c*
Halts compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* character can be one of the following: [**fablyzx**] (see the **-t** option for an explanation). The **-H***c* option selects the compiler pass in the same way as the **-t** option. If this option is used, the

symbol table file produced and used by the passes is given the name of the last component of the source file with the suffix changed to **.T**, and the file is always retained after the compilation is halted.

-ieee
Support all portable features of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754- 1985), including the treatment of denormalized numbers, NaNs, infinities, and the handling of error cases. This flag also sets the **_IEEE_FP** macro.

-machine_code
List the generated machine code in the listing file. To produce the listing file, you must also specify the **-source_listing** option. The default is not to list the generated machine code.

-p
Perform profiling by periodically sampling the value of the program counter. This option affects only linking. When linking is done, this option replaces the standard run-time startup routine with the profiling run-time startup routine (mcrt0.o) and searches the level 1 profiling library (libprof1.a). When profiling is completed, the startup routine calls the monstartup routine and produces a mon.out file that contains execution-profiling data for use with the postprocessor prof. Same as **-p1**.

-p0
Do not perform profiling. This is the default.

-p1
Perform profiling by periodically sampling the value of the program counter. This option affects only linking. When linking is done, this option replaces the standard run-time startup routine with the profiling run-time startup routine (mcrt0.o) and searches the level 1 profiling library (libprof1.a). When profiling is completed, the startup rou- tine calls the monstartup routine and produces a mon.out file that contains execution-profiling data for use with the postprocessor prof. Same as **-p**.

-[no]pg
Perform call graph profiling using the **gprof** tool. Specify **-nopg** to disable selectively profiling for individual modules when using the **-pg** graph profiling option.

For more information on profiling UPC code, see the *Tru64 UNIX Programmer's Guide* and the `gprof(1)` reference page.

-preempt_symbol
Provide full symbol preemption. Full symbol preemption allows the replacement of the definition of an individual external symbol (function or data) in a shared library. This "replacement" occurs at runtime when the dynamic loader uses a definition of the symbol from the main object or some other shared library in preference to the definition in the original shared library. This behavior may be important when a particular symbol is defined in both the main object and in a shared library (or in multiple shared libraries).

-readonly_strings
Make all string literals readable only. This is the default. See also **writable_ strings**.

-rpath *string*, -norpath
Set the rpath to the specified string. This option is meaningful for shared linkage only.

-shared
Produce a shared object. This includes creating all of the tables for run-time linking, and resolving references to other specified shared objects. The object created can be used by the linker to make dynamic executables.

-show *keyword[,keyword,...]*
Specify one or more items to be included in the listing file. When specifying multiple keywords, separate them by commas (with no intervening blanks). To use any of the **-show** keywords, you must also specify the **-source_listing** option. The **-show** keywords are as follows:

all	Enable all options
[no]header	Specify whether to produce header lines at top of each page of listing
[no]include	Specify whether to place contents of **#include** files in the program listing
none	Negate all options
[no]source	Specify whether to place source program statements in the program listing

The default is **-show header,source**.

-signed
Cause all char declarations to have the same machine representation and value set as signed char declarations. This is the default. This option is ignored if specified on the same command line as **-unsigned.**

-r
Retain relocation entries in the output file. Relocation entries must be saved if the output file is to become an input file in a subsequent linker run. This option prevents final definitions from being given to common symbols; it also suppresses the diagnosis of undefined symbols.

-t*c*[*c*...]
The **-t**, **-h**, and **-B** options are used together to specify a location and/or name for one or more compiler passes, tools, libraries, or include files, other than their normal locations or names.

The **-t** option specifies to which compiler passes (or components) the **-h** and **-B** options that follow apply. On Tru64 UNIX, the *c* characters can be one or more of the following:

Character	Name of pass, tool, or component
a	as0
b	as1
k	C compiling
l	ld
q	HP-UX C compiler
Q	UPC preprocessor

Character	Name of pass, tool, or component
y	ftoc
z	cord
r	[m]crt0.o
n	libprof1.a
L	om
M	_main.o
s	stdlib

On HP-UX, only the passes specifed by q, Q, and l are supported.

To relocate the comp.config file, use the **-comp.config** option.

The **-t** and **-h** options are not processed until the next **-B** option is processed. If more than one **-t** option or more than one **-h** option appear on the command line before the next **-B** option, only the last of the previous **-t** and **-h** option arguments are used.

If you specify the **-p** option, it must precede the **-tr -h -B** options because the processing of the latter depends on a value which is modified by the **-p** option.

-trapuv
Forces all uninitialized stack variables to be initialized with 0xfff58005fff58005. When this value is used as a floating-point variable, it is treated as a floating-point NaN and causes a floating-point trap. When it is used as a pointer, an address or segmentation violation usually occurs.

For programs compiled without the **-trapuv** option, the debugger stops only on executable statements in which the value of a specified variable changes. With the **-trapuv** option, the debugger stops on these statements and also stops on all local variable declarations. (The debugger treats the local variable declarations as assignment statements because the variables are initialized by the compiler.)

-verbose
Include identifiers with diagnostic messages. These identifiers can be used with #pragma message directives, or as arguments to **-msg_disable** options.

-volatile
Compile all variables as volatile.

4.3.5 Optimization Options Supported on All Platforms

The `upc` command supports the following optimization options on all platforms.

-assume *option*
Allow the compiler to make assumptions regarding certain alignments or code transformations. On Tru64 UNIX, the following options are supported. On HP-UX and XC Linux, only the **nested_upc_forall** and **nonested_upc_forall** options are supported.

[no]nested_upc_forall
Specifies that all **upc_forall** statements are to be presumed possibly nested or never nested, respectively. **-assume nested_upc_forall** is equivalent to placing a `#pragma upc nested` at the beginning of the compilation unit, and is the default setting. **-assume nonested_upc_forall** is equivalent to putting a `#pragma upc unnested` at the beginning of the compilation unit.

4.3.6 Optimizatization Options Supported Only on Tru64 UNIX

The `upc` command supports the following options only on Tru64 UNIX systems.

-[no]ansi_alias
Directs the compiler to assume the ANSI C aliasing rules. By so doing, the compiler has the freedom to generate better optimized code.

If a program does not access the same data through pointers of a different type (and for this purpose, signed and qualified versions of an otherwise same type are considered to be the same type), then assuming ANSI C aliasing rules allows the compiler to generate better optimized code.

If a program does access the same data through pointers of a different type (for example, by a "pointer to int" and a "pointer to float"), you must not allow the compiler to assume ANSI C aliasing rules, because these rules can result in the generation of incorrect code.

The default is **ansi_alias**.

-arch *option*
Specifies the version of the Alpha architecture for which to generate instructions. All Alpha processors implement a core set of instructions and, in some cases, the following extensions: BWX (byte- and word-manipulation instructions), MVI (multimedia instructions), FIX (square root and floating-point convert extension), and CIX (count extension). (*The Alpha Architecture Reference Manual* describes the extensions in detail.)

The option specified by the **-arch** flag determines which instructions the compiler can generate:

generic	Generate instructions that are appropriate for all Alpha processors.
host	Generate instructions for the processor on which the compiler is running (for example, EV56 instructions on an EV56 processor, and EV4 instructions on an EV4 processor).
ev4,ev5	Generate instructions for the EV4 processor (21064, 20164A, 21066, and 21068 chips) and EV5 processor (some 21164 chips), respectively. (Note that the EV5 and EV56 processors both have the same chip number - 21164.) Applications compiled with this option do not incur incur any emulation overhead on any Alpha processor.
ev56	Generate instructions for EV56 processors (some 21164 chips). This option is the default. This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX extension. Applications compiled with this option can incur emulation overhead on EV4 and EV5 processors.
ev6	Generate instuctions for EV6 processors (21264 chips). This option permits the compiler to generate any EV56 instruction, plus any instructions contained in the MVI and FIX extensions. Applications compiled with this option can incur emulation overhead on EV4, EV5, and EV56 processors.

ev67	Generate instuctions for EV67 processors (21264a chips).
	This option permits the compiler to generate any EV6 instruction, plus any instructions contained in the CIX extension.
	Applications compiled with this option can incur emulation overhead on EV4, EV5, EV56, and EV6 processors.
pca56	Generate instructions for PCA56 processors (21164PC chips).
	This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX or MAX extensions.
	Applications compiled with this option can incur emulation overhead on EV4, EV5, and EV56 processors.

A program compiled with any of the options runs on any Alpha processor. Beginning with Version 4.0 of the operating system and continuing with subsequent versions, the operating system kernel includes an instruction emulator. This capability allows any Alpha chip to execute and produce correct results from Alpha instructions, even if some of the instructions are not implemented on the chip. Applications using emulated instructions run correctly, but can incur significant emulation overhead at run time.

On Tru64 UNIX, the **psrinfo -v** command can be used to determine which type of processor is installed on any given Alpha system.

-assume *option*
Allow the compiler to make assumptions regarding certain alignments or code transformations. On Tru64 UNIX, the following options are supported. On HP-UX and XC Linux, only the **nested_upc_forall** and **nonested_upc_forall** options are supported.

[no]accuracy_sensitive
Specify whether certain code transformations that affect floating-point operations are allowed. These changes may affect the accuracy of the program's results.

The **accuracy_sensitive** option directs the compiler to use only certain scalar rules for calculations. This setting can prevent some optimizations. This is the default.

The **noaccuracy_sensitive** option frees the compiler to reorder floating-point operations based on algebraic identities (inverses, associativity, and distribution). This allows the compiler to move divide operations outside loops, for improved performance.

[no]aligned_objects
The **aligned_objects** option causes the compiler to assume that a dereferenced object's alignment matches or exceeds the alignment indicated by the pointer to the object. This is the default. On Tru64 UNIX systems, dereferencing a pointer to a longword- or quadword-aligned object is more efficient than dereferencing a pointer to a byte- or word-aligned object. Therefore, when the compiler assumes that a pointer object of an aligned pointer type does point to an aligned object, it can generate better code for pointer dereferences of aligned pointer types.

The **noaligned_objects option** flag causes the compiler to generate longer code sequences to perform indirect load and store operations in order to avoid hardware alignment faults for arbitrarily aligned addresses. Although this flag may generate less efficient code than the **aligned_objects option**, by avoiding hardware alignment faults, it speeds the execution of programs that reference unaligned data.

[no]math_errno
Controls the compiler's assumption about a program's dependence on the setting of errno by math library routines:

By default (**-assume math_errno**), the compiler assumes that the program might interrogate errno after any call to a math library routine that is capable of setting errno. The definition of the ANSI C math library allows programs to depend on this behavior, which unfortunately restricts optimization because this causes most math functions to be treated as having side effects.

Specifying **-assume nomath_errno** instructs the compiler to assume that the program does not look at the value of errno after calls to math functions. This assumption allows the compiler to reorder or combine computations to improve the performance of those math functions that it recognizes as intrinsic functions. In practice, robust floating-point code seldom relies on errno to detect domain or range errors, so **-assume nomath_errno** can often be safely used to improve performance.

[no]nested_upc_forall
Specifies that all **upc_forall** statements are to be presumed possibly nested or never nested, respectively. **-assume nested_upc_forall** is equivalent to placing a `#pragma upc nested` at the beginning of the compilation unit, and is the default setting. **-assume nonested_upc_forall** is equivalent to putting a `#pragma upc unnested` at the beginning of the compilation unit.

[no]ptrs_to_globals
Controls whether the compiler can safely assume that global variables have not had their addresses taken in code that is not visible to the current compilation.

The default is **ptrs_to_globals**, which directs the compiler to assume that global variables have had their addresses taken in separately compiled modules and that a pointer dereference in the module could be accessing the same memory as a global variable. This is often a significant barrier to optimization.

While the **ansi_alias** option allows some resolution based on data type, **ptrs_to_globals** provides significant cant additional resolution and improved optimization in many cases.

noptrs_to_globals tells the compiler that any global variable accessed through a pointer in the compilation unit must have had its address taken within that compilation unit. The compiler can see any code that takes the address of an extern variable. If it does not see the address of the variable being taken, the compiler can assume that no pointer within this compilation unit points to the variable.

[no]trusted_short_alignment
Control the alignment assumptions for code generated for indirect load and store instructions.

The **trusted_short_alignment** option indicates that the compiler should assume any short accessed through a pointer is naturally aligned. This generates the fastest code, but can silently generate the wrong results when an unaligned short object crosses a quadword boundary.

The **notrusted_short_alignment** option tells the compiler that short objects might not be naturally aligned. The compiler generates slightly larger (and slower) code that will give the correct result, regardless of the actual alignment of the data. This is the default.

The **notrusted_short_alignment** option does not override the **__unaligned** type qualifier or the **-assume noaligned_objects** flag.

[no]whole_program

Asserts to the compiler that except for "well-behaved library routines", the whole program consists only of the single object module being produced by this compilation. The optimizations enabled by **whole_program** include all those enabled by **nopointer_to_globals**, and possibly other optimizations.

The default is **nowhole_program**.

-inline *keyword*, -noinline *keyword*

Specifies whether to provide inline expansion of functions. The **-noinline** flag disables the inlining optimization that would otherwise be performed by default under the following compiler optimization **-O[n]** flags:

- When **-O2**, **-O3**, or **-O4** is specified.

You can specify one of the following as the keyword to control inlining:

all	Inline every call that can be inlined while still generating correct code. Recursive routines, however, do not cause an infinite loop at compile time.
manual	Inline only those function calls explicitly requested for inlining by an inline keyword. This is the default when compiling with the **-O1** flag.
none	Do not do any inlining. This is the default when compiling with the **-O0** optimization level.
size	Inline all of the function calls in the manual category, plus any additional calls that the compiler determines would improve run-time performance without significantly increasing the size of the program. This is the default when compiling with either the **-O2** or the **-O3** flag.
speed	Inline all of the function calls in the manual category, plus any additional calls that the compiler determines would improve run-time performance, even where it may significantly increase the size of the program.

For optimization level 0 (**-O0**), the **-inline** flag is ignored and no inlining is done.

-[no]intrinsics

The **-intrinsics** option causes the compiler to recognize intrinsic functions wherever it can automatically, based only on name and call signature. On Tru64 UNIX, unlike **-D_INTRINSICS**, this option can treat library function calls as intrinsic even when the appropriate header file is not included. Any function declaration or call site (in the case of implicit declaration) with a name matching the name of an intrinsic function is examined to see if its parameters and return result are consistent with the intrinsic function of that name. If so, calls are treated as being intrinsic. If not, a diagnostic is issued and calls are treated as ordinary external function calls.

When the compiler identifies a function as an intrinsic function, it is then free to make code optimizations (transformations) based on what it knows about the operations performed by the standardized version of that function—given an optimization level (**-O***n*) that enables the intrinsic treatment of that particular function.

The optimization level determines which functions can be treated as intrinsics:

-O0 or **-O1**	No intrinsic functions. The **-intrinsics** option has no effect at this optimization level.
-O2 (or **-O**)	Memory and string functions: **alloca**, **bcopy**, **bzero**, **memcpy**, **memmove**, **memset**, **strcpy**, **strlen** Math functions: **abs**, **fabs**, **labs**, **atan**, **atan2**, **atan2f**, **atand**, **atand2**, **atanf**, **ceil**, **ceilf**, **cos**, **cosd**, **cosf**, **floor**, **floorf**, **sin**, **sind**, **sinf**,
-fast	(due to its supplying **-assume nomath_errno**, and **-O3**,) **acos**, **acosf**, **asin**, **asinf**, **cosh**, **coshf**, **exp**, **expf**, **log**, **log10**, **log10f**, **logf**, **log2**, **pow**, **powf**, **sqrt**, **sqrtf**, **sinh**, **sinhf**, **tan**, **tand**, **tanf**, **tanh**

The **-intrinsics** option is in effect by default. To disable the default, specify the **-nointrinsics** option. To disable the intrinsic treatment of individual functions, specify the function names in a **pragma function** directive in your source code.

Although on Tru64 UNIX **-intrinsics** is the default (and it generally treats calls to [**f**]**printf** as intrinsic), to have the low-level support routines for intrinsic [**f**]**printf** inlined, the compilation must include and also specify both **-D_INTRINSICS** and **-D_INLINE_INTRINSICS** on the command line.

-D_INTRINSICS
Affects the compilation of some system header files, causing them to compile **#pragma intrinsic** directives for certain functions that they declare. The exact functions affected can vary depending on the language mode and other macro definitions. See the header files **math.h**, **stdio.h**, **stdlib.h**, **string.h**, and **strings.h** for details. The exact effect of each **#pragma intrinsic** varies by function, by optimization options, and by other compile-time options. The basic effect is to inform the compiler that the function specified in the pragma is the one by that name whose behavior is known to the compiler (that is, it is a standard C or commonly-used library function rather than a user -written external function). This gives the compiler license to perform additional checks on the usage of the function and issue diagnostics, and to optimize and/or rewrite calls to it based on the compiler's understanding of what the function does. Some possible optimizations include generating complete inline code, generating partial inline code with calls to one or more different functions, or just using characteristics of the function to move the call site or avoid some of the overhead triggered by an external call.

-D_INLINE_INTRINSICS
Affects the compilation of **stdio.h** in two ways:

- Whenever the header file would otherwise define **getc** and **putc** as preprocessor macros expanding into code to access the **_cnt** and **_ptr** members of the referenced FILE object directly, instead these macros are defined to invoke inlined static functions defined in the header file. The use of an inlined static function instead of a simple macro prevents the argument from being evaluated more than once (so arguments containing side effects do not cause a problem), and the function generally will produce better code because it uses local declarations to avoid aliasing assumptions that the compiler has to make when analyzing the traditional macro expansions of **getc** and **putc**. Note that **getc** and **putc** are not expanded inline when i/o locking is required, as is normally the case for reentrant or thread-safe compilations.

- If **-D_INTRINSICS** was also specified, making **printf** and **fprintf** intrinsic functions, then certain of the low-level runtime support routines that may be called for special cases of format strings are defined as inline static functions in the header file, avoiding external calls to these routines in **libc**.

-O[*n*]
Determine the level of optimization, as follows:

Option	Optimization
-O0	No optimization.
-O1	Optimization with space as the primary criterion. This is the default if no optimization option is specified.
-O2	Optimization with time as the primary criterion. This is the default if you specify **-O** without a level number. Optimization levels -O2 and higher enable specific UPC optimizations, such as thread local fetch and store optimizations.
-O3	Enables inline expansion of C global functions.
-O4, **-O5**	Additional global optimizations that improve speed at the cost of extra code size. **-O4** and **-O5** have the same effect.

In addition to affecting the generated code, the **-O** level is passed on to **ld** and is used by both **ld** and **-om** (if **-om** is specified).

The general guidelines for optimization are as follows:

- If the speed of the generated code is more important than code size, specify **-O** (same as **-O2**). In some cases, **-O4** may produce faster code. Using inline all at **-O** or **-O4** can inline more calls (particularly calls to constructors) and improve speed, but this option may increase the code size for some programs beyond an acceptable limit. If you are potentially interested in this option, you should build your program both with and without the option and compare the code size.
- If the size of the generated code is more important than speed, some experimentation may be necessary to determine the best optimization option. Whereas **-O1** (default if an optimization level is not specified) is intended to optimize for code size, in some cases code that is compiled with **-O** (or **-O2**) to optimize for speed may actually be smaller. Also try **-O -unroll 1** to see whether a smaller size is generated. Using **-unroll 1** disables a loop unrolling optimization and generally reduces the code size when **-O** is used. You might also try compiling with the **-noinline** option both with and without **-O** to see whether a reduction in code size occurs.

Other options that can affect run-time size and speed are **-non_shared** and **-om**.

When you use **-g** for best debugging, optimizations are suppressed. Thus, when comparing the effects of different optimization levels, you should not specify **-g** or **-g2**. For such comparisons, you specify **-g0**, which suppresses debugging information.

-om
Perform code optimization after linking, including nop (NoOperation) removal, .lita removal, and reallocation of common symbols. This flag also positions the global pointer register so the maximum addresses fall in the global pointer-accessible window. The **-om** flag is supported only for programs compiled with the **-non_shared** flag. The following options can be passed directly to **-om** by

using the **-WL** compiler flag. Note that some flags that improved performance on older Alpha processors now primarily degrade performance on EV56 and later Alpha processors.

-WL,-om_compress_lita
Remove unused **.lita** entries after optimization, and then compress the **.lita** section.

-WL,-om_dead_code
Remove dead code (unreachable instructions) generated after applying optimizations. The **.lita** section is not compressed by this flag.

-WL,-om_ireorg_feedback,*file*
Use the pixie-produced information in **file.Counts** and **file.Addrs** to reorganize the instructions to reduce cache thrashing.

-WL,-om_no_inst_sched
Disable instruction scheduling.

-WL,-om_no_align_labels
Disable alignment of labels. Normally, the **-om** flag quadword-aligns the targets of all branches to improve loop performance.

-WL,-om_Gcommon,*num*
Set size threshold of "common" symbols. Every "common" symbol whose size is less than or equal to num will be allocated close to each other. This flag can be used to improve the probability that the symbol can be accessed directly from the global pointer register. Normally, the **-om** flag causes the compiler to try to collect all "common" symbols together.

-tune *option*
Select processor-specific instruction tuning for implementations of the operating system architecture. Using the **-tune** option causes the generated code to run correctly on all implementations of the architecture. Tuning for a specific implementation may improve run-time performance; however, code tuned for a specific target may run slower on another target. For a list of options, see the description of -arch.

Note that **-tune ev[*x*]** does not imply **-arch ev[*x*]**. Unlike **-arch**, **-tune** does not cause instruction emulation or illegal instructions on any Alpha architecture.

A program compiled with any of the options runs on any Alpha processor. Beginning with Version 4.0 of the operating system and continuing with subsequent versions, the operating system kernel includes an instruction emulator. This capability allows any Alpha chip to execute and produce correct results from Alpha instructions, even if some of the instructions are not implemented on the chip. Applications using emulated instructions run correctly, but can incur significant emulation overhead at run time.

The **psrinfo -v** command can be used to determine which type of processor is installed on any given Alpha system.

-unroll *n*
Control loop unrolling done by the optimizer. *n* signifies the number of times to unroll loop bodies. Specifying zero for *n* tells the optimizer to use its own default unroll amount. This is the default. Note that the argument *n* is only a suggestion to the optimizer.

4.3.7 Preprocessor Options Supported on All Platforms

Preprocessor options control the action of the preprocessing phase. The `upc` command supports the following preprocessor options on all platforms.

-cpp
Run the preprocessor on the source files before compiling. This is the default. If both **-cpp** and **-nocpp** are included on the command line, the option specified last is in effect.

-C
Prohibit the preprocessor from removing comments in the source file. (Use with the **-E** or **-Em** option.)

-D*name=def*, -D*name*
Define name to the preprocessor, as if the line **#define** *name def* were prepended to the UPC source file. No space is allowed space between the option and *name*. If *name* or *def* contains a dollar sign ($), it must be surrounded by apostrophes ('). If no =*def* is given, the name is defined as 1.

-E
Run only the preprocessor on the source files (regardless of whether a suffix exists), and send the result to the standard output. This sets the **-cpp** option.

-I
Do not search for **#include** files in the standard directories. Because **-I** can be followed by a directory, do not place a nondirectory file name on the command line immediately following **-I**. If **-I** is being used without a directory, follow it with another option or place it at the end of the command line to avoid misinterpretation.

-I*dir*
Define the directory name *dir* to the preprocessor for use in searching for quoted and angle-bracketed include files. There can be a space between the option and dir. The preprocessor searches for include file names that do not begin with a slash (/) in the following order:

Quoted file names:

1. In the directory containing the source file with the **#include** directive.
2. In the directories specified by the **-I** option.
3. In the **/usr/include/c** and **/usr/include**

Angle-bracketed files are searched for in the list of directories specified on the command line, then in the **/usr/include/c** directory, and finally, in the **/usr/include** directory.

If **-nocurrent_include** is specified, the preprocessor does not search in the directory containing the source file (#1 above).

-P
Run only the preprocessor and put the result for each source file in a corresponding **.ixx** file, without including line numbers. If specified after **-E**, a .ixx file is not created. If specified before **-E**, it overrides **-E**. On HP-UX output is sent to `stdout`.

4.3.8 Preprocessor Options Supported Only on Tru64 UNIX

The `upc` command supports the following options only on Tru64 UNIX systems.

-M
Run only the compiler on the source files (regardless of whether a suffix exists), and produce makefile dependencies instead of the usual output. For each source file, the preprocessor creates one makefile entry naming the object file and listing all included files as dependencies. To identify correctly the template implementations on which the source depends, the compiler parses and analyzes the source; the source must be a valid UPC program. To disable this analysis, specify the **-noimplicit_include** option. Same as **-Em**.

-MD
Produce a dependency file, which has the suffix **.d** appended to the object file name. This dependency file is created even if the object file is not. The information and the format in the dependency file is identical to that produced by the **-M** flag. The **-MD** flag allows dependency information to be generated at the same time that a compilation is occurring. If multiple files are specified, only the last file's information is saved.

-nocurrent_include
Do not search for quoted include files in the directory con- taining the source file with the **#include** line. The preprocessor searches the directories specified by the **-I** option and in the standard directory. See also the **-I***dir* option.

-U*name*
Cancel any command-line definition of *name* to the preprocessor, as if the line **#undef** *name* were prepended to the UPC source file. There can be a space between the option and *name*. If name contains a dollar sign ($), *name* must be surrounded by apostrophes ('). The undefine operation occurs after any definitions produced by the **-D** options. Symbols defined by default are listed in the Description section.

4.3.9 Language Mode Options

The `upc` command supports the following language mode options on all platforms.

-std

-std0
Support the pre-ANSI (K & R) C dialect for cases in which the syntax conflicts with ANSI C syntax or semantics.

-std1
Use this option if you want the compiler to enforce the ANSI UPC standard strictly. The default ANSI mode permits some common extensions and provides less strict error checking than the STRICT_ANSI mode. This option also defines the macros **__STD_STRICT_ANSI**, **__STDC__**, and **__STDC_VERSION__**.

4.3.10 Message Information and Control Options Supported on Tru64 UNIX

On Tru64 UNIX systems, the `upc` command supports the following message information and control options. The options apply only to discretionary, warning, and informational messages. The *tag* variable can be the keyword **all**, a tag obtained from the **-msg_display_tag** option, or a number obtained from the **-msg_display_number** option.

-msg_inform *tag*,...
Reduce message(s) severity to informational.

-msg_warn *tag*,...
Reduce message(s) severity to warning.

-msg_error *tag*,...
Increase message(s) severity to error.

-msg_enable *tag*,...
Enable specific messages that would normally not be issued. You can also use this option to re-enable messages disabled with **-msg_disable.**

-msg_disable *tag*,...
Disable message. Can be used for any non-error message.

-msg_quiet
Fewer messages are issued using this option. The **-msg_enable** option can be used with this option to enable specific messages normally disabled using **-msg_quiet.**

The `upc` command supports the following message information options. Both are off by default.

-msg_display_number
Displays the error number at the beginning of each message. Note that "D" (meaning discretionary) indicates that the severity of the message can be controlled from the command line. The message number can be used as the tag in the message control options. If "D" is not displayed with the message number, any attempt to control the message is ignored.

-msg_display_tag
Displays a more descriptive tag at the end of each message. "D" indicates that the severity of the message can be controlled from the command line. The tag displayed can be used as the tag in the above message control options. Note that you can also change the severity of a diagnostic message if the message is discretionary. For example, **-msg_inform 110** changes the severity of message 110 to an informational. These options interact with **-w0, -w1,** and **-w2.**

4.4 Examples

This section provides usage examples.

```
% upc -g -DUSE_CONCENTRATOR -o netmud netmud.c
```

This example creates an executable file named netmud with symbol table information for full symbolic debugging (**-g**). The **-D** option defines the macro name USE_CONCENTRATOR for the preprocessor.

```
% upc -o gfview -I/usr/kafka/src -I/usr/barnes/include gfview.c
```

This command line creates an executable file from the **gfview.c** source file. The **-o** option names the output file gfview. The **-I** option directs the preprocessing phase to search the specified directories for include files not found in the current working directory.

```
% upc -c io_module.c
```

This example preprocesses and compiles the source file **io_module.c** and produces an object file named **io_module.o**. Processing stops after creating the object file.

```
% upc -o newsreader io_module.o ui.c -L/users/dave -lnews
```

This example creates an executable file, and the **-o** option names the file newsreader. The source file **ui.c** is preprocessed and compiled, and then linked with the object file **io_module.o**. The link operation uses the library specified by **-l** (**libnews.a**). The linker first looks for the library in the current working directory, then in the directory specified by **-L** (**/users/dave**), and finally in $PATH.

```
% upc -pg a.c b.c -nopg c.c
```

On Tru64 UNIX systems, this command line enables gprof profiling for files **a.c** and **b.c** and disables profiling for file **c.c**.

4.5 Diagnostics

The error messages produced by the UPC compiler are self-explanatory. The line number and file name where the error occurred are printed along with the diagnostic.

SEE ALSO
upcrun(1), **upcrund**(8), **cc**(1), **csh**(1), **gcc**(1) (<REFERENCE>(platform_linux)), **sh**(1), **ladebug**(1) (<REFERENCE>(platform_linux)), **ld**(1) (<REFERENCE>(platform_linux)), **what**(1), the *Tru64 UNIX Programmer's Guide*, the *HP-UX Programmer's Guide*, and the *HP XC System Software User's Guide*

4.6 Use of the comp.config File

The `comp.config` file allows system administrators to establish a set of compilation options that are applied to compilations on a *system-wide* basis. The compiler options are specified on a single line in the `comp.config` file. The file is placed in the the compiler target directory, `/usr/lib/cmplrs/upc`.

Options from `comp.config` are processed first, and so may be overridden by options specified on the command line. It is possible to control this by use of a vertical bar in the `comp.config` line; switches specified to the left of the vertical bar are placed before command line switches, and those to the right are placed after the command line switches. For example, if the `comp.config` file were as follows:

```
        -signed|-intrinsics
```

the `-signed` switch would appear before command line switches, and the `-intrinsics` switch would appear afterward.

You can use the the DEC_CC environment variable to establish a set of compilation options that are applied to subsequent compilation on a per-user basis.

The `DEC_CC` environment variable can contain two distinct sets of compilation options separated by a single vertical bar (|). The options before the vertical bar are known as prologue options, and the options after the bar are know as epilogue options.

The `DEC_CC` environment variable can begin or end with a vertical bar, or have no vertical bar at all. If no vertical bar is present, the options are treated as prologue options by default. Any vertical bar found after the first vertical bar is treated as whitespace and a warning is issued.

Compiler options are processed in the following order during a compilation:

1. comp.config prologue options
2. `DEC_CC` prologue options
3. command line options
4. `DEC_CC` epilogue options
5. comp.config epilogue options

If **-v** is specified on the command line, the contents of `DEC_CC` and comp.config, if present, are displayed.

4.7 Header Files

All UPC programs should include `upc.h`. You can do this by including either `upc_strict.h` or `upc_relaxed.h`; these files set the default coherence to strict or relaxed, respectively, and then include the common files `upc.h`, `upcstrict.h`, and `upc_relaxed.h`. Alternatively, you could include `upc.h` file directly and specify the **-fstrict** or **-frelaxed** option with the compile command. If the source file references the collective functions (see Section 3.4), you must inlude the `upc_collective.h` file.

4.8 Linking UPC Programs

Unless the `-c` compiler option is used, the `upc` command automatically links the compiled object modules into an executable UPC program file. This file is named `a.out` unless the `-o` option is used to change the file name. The `upc` command also includes the UPC Run-Time System (UPCRTS) in the link step automatically. The program must not be linked non-shared On Tru64 UNIX, the UPCRTS also references the Quadrics Elan and RMS libraries provided as a part of the AlphaServer SC system software.

4.9 Debugging UPC Programs

In Tru64 UNIX clusters, you can use the *TotalView debugger* from Etnus, Inc. to debug UPC programs. *TotalView* is a full-featured, GUI-based debugger specifically designed to meet the needs of parallel applications running on many processors. The *TotalView* documentation set is available directly from Etnus, Inc. However, *Totalview* is not included with the HP UPC software and is not supported. If you install and use *TotalView* and have problems with it, contact Etnus, Inc. UPC support in Totalview may not yet be complete.

You can also debug UPC programs using the UPCRTS family of display facilities (see Chapter 7 or Appendix A). Note that an application that exits with `upc_global_exit` may not generate output for certain run-time statistics.

4.10 Running UPC Programs

You can run UPC programs in one of three ways:

- Single-threaded execution (Section 4.10.1)
- Multithreaded execution using the `prun` or `srun` command (Section 4.10.2)
- Multithreaded execution on an SMP system (or on a single-CPU system) using the UPC Run-Time Environment (Section 4.10.3)

4.10.1 Running Programs in Single Thread Mode

If a UPC program has been compiled with `-fthreads` set to 1, or without having specified a value for `-fthreads`, then it may be run in *single thread mode* by simply executing the program image directly, as with any normal UNIX program.

Single thread mode is useful for basic testing and debugging of UPC programs. Single thread mode can be used on HP-UX, on XC Linux, or on Tru64 UNIX Version 5.1 or later systems; the AlphaServer SC software is not needed.

4.10.2 Running Programs in Multithread Mode

The `prun` (parallel run) command is included as part of the the software provided with the AlphaServer SC system, and provides access to the *Resource Management Services* (RMS) databases and system management services. For detailed information on the `prun` command, refer to the reference page for `prun` and the RMS User Manual, provided with the AlphaServer SC software documentation.

On XC Linux systems, use the `srun` command provided with the Simple Linux Utility for Resource Management (SLURM). For detailed information, see the XC Linux documentation and the reference page for `srun`.

Examples:

```
prun -n 4 upc_loadtest3
srun -n 4 upc_loadtest3
```

These commands run the program `upc_loadtest3` in the default parallel partition, using four threads on four CPUs. You can use all `prun` or `srun` command options to run UPC programs on any configuration of CPUs and nodes. Output from each thread's `stdout` is buffered and displayed at the terminal where the command is executed. Diagnostics output to `stderr` is not buffered; it is immediately displayed at the terminal.

If any inconsistencies are found, the UPCRTS issues a diagnostic message describing the problem and exits before running the `main` function. Inconsistencies might be found in the following areas:

- The declaration of a shared array in different modules
- The value specified for `-fthreads` in diverse modules
- The value specified for `-fthreads` and the value used for the `-n` option to the `prun` or `srun` command
- Some modules are compiled with the **-smp** or **-smp_local** while others are not

If inconsistencies are found, you must modify one or more modules to correct the problem. You must then recompile one or more modules with a consistent value for `-fthreads` and reissue the `prun` or `prun` command with a `-n` value consistent with the value used for `-fthreads`. If all modules compile successfully without specifying `-fthreads`, you can use any value for the `-n` option of the `prun`or `srun` command.

4.10.3 Running Programs in Multithread Mode Using the UPC Run-Time Environment

HP UPC provides its own job control mechanism for SMP machines that do not have Quadrics RMS software, or as an alternative to Quadrics RMS software for jobs that can run on a single SMP node. This job control mechanism is called the UPC Run-Time Environment (RTE) and comprises the UPC job control daemon `upcrund` and the `upcrun` command. For detailed information about the RTE, see Chapter 5.

For a program to run on an SMP system (where all processors share the same physical memory), the shared data must be made visible to all threads. Use the `-smp_local` or the `-smp` compiler option. The `-smp` option provides better performance. Note that you can include the desired option in the `comp.config` file (see Section 4.6).

On HP-UX, the default is to run in SMP mode. It is not necessary to specify the **-smp** option.

5 The UPC Run-Time Environment

For a program to run on an SMP system (where all processors share the same physical memory), the shared data must be made visible to all threads. To do this, use the `-smp_local` or the `-smp` compiler option. The `-smp` option provides better performance. Note that you can include this option in the comp.config file (see Section 4.6).

On HP-UX, the default is to run in SMP mode. It is not necessary to specify the **-smp** option.

HP UPC provides its own job control mechanism for SMP machines that do not have Quadrics RMS software, or as an alternative to Quadrics RMS software for jobs that can run on a single SMP node. This job control mechanism is called the UPC Run-Time Environment (RTE). The RTE comprises the UPC job control daemon (upcrund) and the upcrun command.

5.1 The UPC Job Control Daemon

The UPC installation procedure automatically configures and starts the UPC job control deamon on Tru64 UNIX and HP-UX. The daemon is normally restarted at boot time when the system executes its `init` script, called **/sbin/init.d/upcrund**. The process runs with root privileges. On XC Linux systems, the daemon is not needed, because these systems use the `srun` command provided with the Simple Linux Utility for Resource Management (SLURM).

The UPC job control daemon determines the number of CPUs on the system and uses that value as the maximum number of UPC programs that can be run at one time. On a four-CPU system, for example, a user can specify a maximum **-n** value of 4 to the upcrun command. However, specifying the **-O** option allows more than one UPC thread per processor.

The UPC daemon starts the UPC programs and waits for them to exit. It forwards output from the programs to the user and writes whatever the user types to each program's **stdin**. If the upcrun(1) command receives a **INT**, **QUIT**, or **TERM** signal, the UPC daemon kills the programs with that signal.

After a successful initialization, the UPC daemon writes all messages to the system log maintained by the **syslogd** daemon.

5.1.1 upcrund Options

The upcrund command supports the following options:

-S
Disable the CPU Resource Manager. Not currently supported.

-D
Reserved for use by HP.

-a
Use the **-a** option to specify an alternate socket port number when another service is using the same number as the UPC RTE. The *alternate* port number must be the same value used by the upcrun(1) command, which has a corresponding **-a** option. The default port number for UPC RTE is 9112.

-h
Display help information.

-v
Display the version of the UPC RTE.

5.1.2 upcrund Environment Variables

The UPC daemon uses the following environment variables, but none are required by the user.

UPC_RTE_HOME
This is the working directory of the UPC daemon. By default, UPC_RTE_HOME is set to **/etc/upc_rte**. The UPC daemon executes chdir to this directory as part of its initialization.

UPC_RTE_CONF
This variable may be used in future versions to control UPC node configuration.

5.1.3 upcrund Files

The UPC daemon uses the following files:

/sbin/init.d/upcrund
Init script for the UPC daemon.

/usr/sbin/upcrund (Tru64 UNIX), /opt/upc/bin/upcrund (HP-UX)
The UPC daemon executable

/etc/upc_rte
The default HOME directory for the UPC daemon.

/etc/upc_rte/crm_sock@nodename
A socket created by the UPC daemon.

5.2 The upcrun Command

To execute UPC programs within the UPC RTE, you use the upcrun command. In this release, upcrun is restricted to SMP programs on a single system. For proper execution, UPC programs must be compiled using the **-smp_local** or **-smp** option. The **-smp** option is preferred.

On HP-UX, the default is to run in SMP mode. It is not necessary to specify the **-smp** option.

The upcrun command executes multiple copies of the specified program. The set of processes started by a upcrun command is called a job. Each job is assigned a unique *job id*.

The programs start in the same directory from which the upcrun command is issued and execute as if the owner ran them directly. The user's entire environment is made available to the running program.

The upcrun command exits when all the processes have exited or when one of the processes exits upon receiving a signal. Normally, the exit values for the processes are summed and the result value becomes the exit value of the upcrun command. However, if a process is killed because of a signal, the UPC RTE displays a job failure message containing the number of the signal that killed the job.

Once a job is started, the upcrun command catches the signals **INT**, **QUIT**, and **TERM**, and forwards them to all the programs. Normally these signals cause programs to exit, but programs may choose to act on the signals or block them.

5.2.1 upcrun Options

The upcrun command supports the following options:

-a *port*
Use the **-a** option to specify an alternate socket port number when another service is using the same number as the UPC RTE. The *alternate* port number must be the same value used by the upcrund(8) daemon, which has a corresponding **-a** option. The default port number for UPC Run-Time Environment (RTE) is 9112.

-h
Display help information.

-I
Normally, upcrun starts jobs as long as CPU resources are available. When resources are exhausted, running jobs terminate immediately, and pending jobs are queued for later execution. Use the **-I** option to prevent the queueing of jobs when there are insufficient CPU resources.

-n *procs*
Specifies the number of processes to start. The value of **procs** must be less than or equal to the number of CPUs on the system. The default value is 2. On Tru64 Unix, each process runs on its own CPU and has sole use of that CPU.

-N *nodes*
Specifies the number of nodes on which to run the program. By default, upcrun uses as many nodes as needed to fulfill the **-n** *procs* specification. In the current release, only -N 1 is supported; that is, all threads must run on CPUs in a single multiprocessor node.

-0
Allows more than one UPC thread per processor. Used in conjunction with the **-n** option of the `upc` command.

-q
Displays information about jobs that are running or are queued.

-t
Prefixes each line of output with the value of **MYTHREAD** (the UPC thread number). You can use the **-t** option to tag output from programs (**stdout** and **stderr**) and thus determine which UPC process wrote which line of output. The UPC RTE does not buffer any output it reads from the UPC program. Any data available at the time of the read request is written to the output of the upcrun command. Separation of **stderr/stdout** data is maintained. Read requests are made for 8K blocks.

Programs may read from **stdin**. The upcrun program forwards data read from **stdin** to all of the programs. The upcrun program can be executed in the background.

-v
Display the version of the UPC RTE.

5.2.2 upcrun Operands

program [args...]
The name of the UPC program optionally followed by its arguments. The program can be specified with a full or relative path name or found using the user's path environment variable.

5.2.3 upcrun Warnings

For the UPC RTE to manage multiple jobs, processes are placed in process groups. A UPC program cannot change its process group.

5.2.4 upcrun Exit Status

The upcrun command exit code is a bitwise OR of all the exit codes of the parallel processes. An exit value of 1 indicates that upcrun could not fulfill the users request.

5.2.5 upcrun Examples

In the following example, upcrun is used to execute 2 copies of the following program:

```
main()
{
  printf("MYTHREAD is %d\n", MYTHREAD);
}
```

```
$ upcrun -n2 a.out
upcrun: starting job #1
MYTHREAD is 1
MYTHREAD is 0
```

In the following example, upcrun executes four copies of the shell; each shell calls echo to print its pid.

```
$ upcrun -n4 sh -c 'echo My PID is $$'
upcrun: starting job #56
My PID is 2984
My PID is 2985
My PID is 2986
My PID is 2987
```

In the following example, an error occurs because too many resources are requested:

```
$ upcrun -n8 my_upc_prog
Unable to fulfill your request to run 8 processes.
There are only 4 CPUs available for UPC.
```

The following behavior occurs if there is a temporary CPU resource shortage:

```
$ upcrun -n4 my_upc_prog
upcrun: queuing job #64
upcrun: starting queued job #64
```

To prevent the job from being queued, use the '-I' option:

```
$ upcrun -n4 -I my_upc_prog
Unable to allocate CPU resources immediately.
```

If you specify an alternate socket port which is not the same as that being used by the UPC daemon, upcrun displays an error message appears like the following:

```
$ upcrun -a9001 -n2 my_upc_prog
upcrun: starting job #71
Connect to port 9001 on node upc failed
Proceeding to shutdown job ...
Shutdown of job is complete
```

In the following example, the user program contains an error that causes one of the processes (with MYTHREAD = 1) to get a segmentation violation:

```
$ upcrun -n2 my_bad_prog
Job failure on node upc: my_bad_prog (UPC_PEER_PID=1) was killed
with signal '11' on Fri Sep  7 15:41:41 2001
Proceeding to shutdown job ...
Shutdown of job is complete
```

If the UPC daemon is not running, upcrun displays the following error messages:

```
$ upcrun -n2 my_upc_prog
Failed to connect to the CPU Resource Manager.
Connect failed in init_client_unix_sock
path /etc/crm_sock@upc: No such file or directory

Check if the UPC Daemon is running and that it can
respond to CPU resource requests (no '-S' switch).
```

5.2.6 upcrun Environment Variables

Each thread receives an environment variable UPC_PEER_PID with a value of that particular thread's thread number. In addition, all user environment variables are made available to the programs.

6 Programming Techniques

This chapter describes UPC programming techniques.

6.1 Sharing Data Across Multiple Threads

Variables that are declared with the **shared** type qualifier are visible to all threads. Shared arrays are distributed across all threads, while shared scalars have affinity to thread 0.

6.1.1 Granularity

Adjacent small data types (anything smaller than an `int`, but see discussion of bit fields below) may be accessed by different threads without interference *when referenced as shared data*. There may be a performance penalty, however, in making such references. It is suggested that small data types, if needed, be referenced primarily on the thread to which the data have affinity.

A pointer to a shared small data type may be cast to a pointer to private. If the small datum is stored via the pointer to private, the compiler may find it more efficient to read, modify, and write out a larger number of bytes than the specified data type. *This has the potential of overwriting modified data if some of the adjacent bytes have been written by a different thread.* There are several ways to avoid this problem:

- Avoid small data types;
- Avoid having adjacent small data written concurrently by multiple threads;
- Use locks to synchronize access to small data;
- Declare any pointers to private to small shared data **volatile**, and specify the `-strong_volatile` option on the command line. (This option is available only on Tru64 UNIX.)

6.1.2 Bit Fields

If a shared structure contains a bit field, and that bit field is modified, the entire containing structure is fetched, the bit field modified, and the entire structure written back. This operation can be slow, and can cause overwriting of other data modifications if any part of the structure is referenced by other threads. **Bit fields in shared structures should be avoided**, and where possible accessed via pointers to private.

6.2 Synchronizing Access To Shared Data

References to shared data may be synchronized via several methods:

- Locks;
- Strict references and fences;
- Barrier statements.

Each has its own characteristics. Strict references and fences guarantee that a thread will see any writes to shared data that have occurred, but do not guarantee that any particular write has taken place. Barriers provide uniform synchronization across all threads, guaranteeing that all have reached a particular point in the program. Locks permit finer-grain synchronization for any subset of threads. Use of locks is usually for the purpose of resource allocation—to allow a single thread to allocate memory, perform I/O, or protect access to a commonly used data structure.

6.3 Improving Performance Of Shared Data Access

References to shared memory locations can be several orders of magnitude slower than references to private memory. See Chapter 7 for a discussion of several features in the HP UPC Run-Time System that can alleviate these delays. UPC programmers can further reduce these overheads by paying attention to allocation of shared data and using access techniques that minimize these delays.

Shared data references are ultimately always performed using pointers to shared. Such references may be performed on the thread to which the target has affinity (call this case a *local target*), or on a different thread (call this case a *remote target*). Alternatively, in the case of a *local target*, the pointer to shared may be cast to a pointer to private and the target referenced via the pointer to private. In decreasing speed order:

1. Referencing a *local target* via a pointer to private.
2. Referencing a *local target* via a pointer to shared.
3. Referencing a *remote target* via a pointer to shared.

If a large amount of work can be done by the threads on their own data (data with affinity to the current thread), *without the data being modified by other threads*, it is best to do the work using pointers to private. If this is not possible, it is best to have the threads do most of the modifications of their own data. (See also the discussion of granularity in Section 6.1.1.)

References to fields of structures that are shared are turned into direct references to the field type, and so are handled efficiently, with the exception of bit fields (see Section 6.1.2).

Even with relaxed references, remote data fetches must be completed in order to provide a value used in an expression. On the other hand, relaxed remote stores may be delayed. The programmer may take advantage of this fact by assigning work to threads accordingly. For example:

```
shared int A[THREADS], B[THREADS];
upc_forall(i = 0; i < THREADS; i++; &A[i])
  A[i] = B[(i+1)%THREADS];
```

In this loop, the store to A is performed locally, as specified in the affinity expression of the forall statement. Meanwhile, the remote fetch of the array B must complete in order to have a value to assign to A. It would be better to switch the thread assignments by changing the affinity expression:

```
upc_forall(i = 0; i < THREADS; i++; &B[(i+1)%THREADS])
```

In the modified loop, the fetch of B is performed locally, and the remote store to A can be delayed.

6.4 Using the upc_forall Statement

As discussed in Section 2.12.1, the **upc_forall** statement is a UPC-specific statement designed to allow distribution of executed loop iterations based on an *affinity* expression. The following sections describe some possible uses for this statement. See also the examples is Section 6.3.

6.4.1 Null Affinity Expressions

Even without an affinity expression, the **upc_forall** statement is slightly different from an ordinary **for** statement in the C language. The difference is that, because **upc_forall** is a parallel statement, the programmer is asserting that no premature loop exits exist, and that no iteration modifies the values computed for any iteration executed on a different thread. All threads see the same sequence of iterations.

These properties allow the compiler to make useful assumptions about the operations in the loop body, such as, that any functions called do not modify values needed for future iterations. These properties thus allow the compiler to perform memory-access optimizations such as pre-fetching and store delaying with the assurance that these speedups will not cause the loop to compute incorrect results.

6.4.2 Integer Affinity Expressions

Next, it is important to note that the affinity expression can be either an integer or the address of a pointer to shared value. The simplest form of a **upc_forall** that partitions the iteration space into **THREAD** independent streams using the loop index for its affinity expression, for example:

```
upc_forall (i=0; i<LOOPLIMIT; i++; i) { body }
```

In this case, assuming $LOOPLIMIT > THREADS$, all iterations where the relationship `i%THREADS == MYTHREAD` holds will have their body statement executed by the current thread. This construct partitions the iteration space evenly among the threads in a round-robin fashion. The HP UPC compiler optimizes such constructs in such a way that the skipped iterations do not actually take extra time.

If all computations in the loop body involve only private memory values, or shared values whose affinity is to the current thread, then such loops will run at approximately the same speed as their counterparts in an ordinary C program.

Such a construct is also appropriate if all shared array elements referenced in the loop are not declared with block size specifiers, and use only simple indexes, such as `a[i]`. In this case, the array index matches the affinity of the individual element, and the performance hit of a remote-affinity shared reference need not be paid.

Integer affinity expressions are also appropriate if the user desires to partition the iteration space into non-uniform chunks, or if the dominant affinity of the body statement changes in a non-uniform manner throughout the iteration space. By using an affinity expression that is an integer expression or function call, any arbitrary pattern of iteration references can be produced. Such an expression need only return **MYTHREAD** for the iterations where the body must be executed, and a non-**MYTHREAD** value for other iterations. UPC requires that each iteration be executed by exactly one thread.

6.4.3 Shared Address Affinity Expressions

The most common form of affinity expression is the address of a shared array element referenced within the loop. The main benefit of this form is to provide a simple way to partition the iteration space in a way that takes advantage of the alignment of the affinities of several shared arrays used within the loop. For example, in a simple summation loop:

```
shared int a[100*THREADS], b[100*THREADS], c[100*THREADS];
. . .
upc_forall (i=0; i<100*THREADS; i++; &a[i]) {
 [i] = a[i] + b[i];
}
```

the iteration space is partitioned in such a way that every iteration is executed on exactly those threads for which the a[i], b[i], and c[i] values have local affinity. No remote shared references are needed to execute this loop, so it will run almost as fast as a simple C **for** loop with similar form. This method works well regardless of the block size values used for the array declarations.

For more complicated loops, the design goal is to choose an affinity expression that minimizes the aggregate overhead of remote references, and partitions them evenly among the threads. Often, that goal will be achieved by choosing the address of an array element that appears within the loop body and which has the same affinity as other array elements used in the body. However, more complicated situations may require the use of a “surrogate” array, which is not used in the body at all, but which is only declared to provide an affinity pattern to follow.

7 Run-Time Library Configuration and Control

The HP UPC Run-Time System (UPCRTS) is a general purpose interprocess communication system with calling interfaces optimized for UPC programs compiled by the HP UPC compiler. In addition to its primary function of supporting the UPC language shared fetch and store operations, the UPCRTS supports an extensive error checking system, the UPC language functions for memory management, synchronization, and locking. The UPCRTS provides binary control interfaces for initialization, communication, synchronization, and resource management, as well as an extensive user interface in the form of a variety of environment variables and display services. The values of any recognized environment variables (prefixed by `UPCRTS_`) are examined at start-up time and used to control operating modes, which run-time optimizations to attempt, and what display services to activate. The default modes are chosen to support correct UPC program execution with minimal interference with the underlying Quadrics™ Elan™ services (when applicable). Optional modes offer higher performance for UPC programs that can tolerate certain restrictions in resource usage.

RTS behavior differs according to its implementation in the Quadrics or SMP environment.

7.1 Quadrics Environment

The Quadrics Elan software interface supports a high-speed Remote Direct Memory Access (RDMA) facility and relies on the Quadrics Resource Management Services (RMS) facilities for parallel startup and I/O to `stdin`, `stdout`, and `stderr` for all threads. The UPCRTS provides several significant performance optimization systems targeting the performance characteristics of the Quadrics Elan Network Interfaces and Elite™ switches.

The properties that make such enhanced operating modes possible are:

- The latency and bandwidth characteristics of the Quadrics interconnect. While the first word of a Quadrics RDMA store or fetch takes 2-6 microseconds, each additional word requires only a few additional nanoseconds.
- By combining several UPC language fetch or store operations into a single Quadrics operation, aggregate shared access performance can be increased by 10-fold or greater.
- The Quadrics Elan interfaces support overlapping several shared memory access operations at a time on each thread. By supporting binary interfaces and resource management capabilities that exploit this feature, the UPCRTS can hide a significant amount of the base latencies of the individual operations. Aggregate performance improvement can be as much as five-fold from this feature.

- Several of the system configurations that are supported as individual nodes in AlphaServer SC systems are themselves shared memory multiprocessors. Shared memory access operations performed between CPUs in such systems using true shared memory operations can be two orders of magnitude faster than using the Quadrics interconnect for such references. By supporting the use of the `-smp_local` operating mode, the UPCRTS allows UPC programmers access to this high performance feature.
- The Quadrics Elan network interconnect adapters are themselves programmable. Certain operations, such as global synchronization and lock management, can use this feature to deliver high performance implementations of UPC operations that correspond to these primitives.

Because access operations involving shared memory can be 2 to 4 orders of magnitude slower than similar operations involving local private memory, it is not unusual for UPC programs to have shared memory access delays as their primary performance bottlenecks. However, by exploiting the enhanced modes and optimizations in the UPCRTS, the HP UPC programmer and user can shift the overall bottlenecks of many applications away from interconnect performance and back to individual CPU performance, thus achieving the ultimate goal of parallel programming: linear performance increases up to high processor counts without the high programming complexity of explicit message-passing calls.

7.2 SMP Environment

HP UPC provides its own job control mechanism for SMP machines that do not have Quadrics RMS software, or as an alternative to Quadrics RMS software for jobs that can run on a single SMP node. This job control mechanism is called the UPC Run-Time Environment (RTE) and comprises the UPC job control daemon (upcrund) and the upcrun command. For detailed information about the RTE, see Chapter 5.

For a program to run on an SMP system (where all processors share the same physical memory), the shared data must be made visible to all threads. To do this, use the `-smp_local` or the `-smp` compiler option. The `-smp` option provides better performance. Note that you can include the desired option in the `comp.config` file (see Section 4.6).

On HP-UX, the default is to run in SMP mode. It is not necessary to specify the **-smp** option.

7.3 UPC RTS Memory Usage

The RTS uses the following memory zones:

all alloc
Contains all shared data dynamically allocated using the upc_all_alloc memory allocator.

global alloc
Contains all shared data dynamically allocated using the upc_global_alloc memory allocator.

local alloc
Contains all shared data dynamically allocated using the upc_alloc memory allocator.

static alloc
If **-smp_local** is specified (note that **-smp** implies **-smp_local**), statically declared shared data (data declared at file scope or with the static storage class) are dynamically allocated in the static alloc memory zone. Otherwise, the compiler allocates space for the data, and the `static alloc` memory zone is empty.

The RTS uses the following mechanism to determine the zone sizes:

- During application initialization, the RTS learns the following from the compiler:
  - Total size of the **static alloc** zone
  - Whether the application makes any calls to allocate from the **all alloc** zone.
  - Whether the application makes any calls to allocate from the **local alloc** zone.
  - Wheter the application makes any calls to allocate from the **global alloc** zone.
- When running in the Quadrics environment, the RTS detects the size of Elan-Visible space. This size is the maximum amount of memory that can be considered shared. The size of Elan-visible space can be modified using the following Environment Variables:

  LIBELAN_GALLOC_SIZE
  LIBELAN_ALLOC_SIZE

  Note that Galloc size is a portion of the Alloc size, so the Alloc setting must be larger than the Galloc setting.
- The RTS detects the maximum shared memory segment size from the ipc subsystem. When running under the `prun` environment, the minimum of this value and the size of the Elan-Visible space determines the maximum amount of memory that can be shared.
- Once the RTS has gathered this data, it is able to partition the shared memory segment into appropriately sized zones. It allocates a shared memory segment based on the computations described earlier and reserves space for the static alloc memory zone, if applicable. It then partitions the remaining portion of the segment into the dynamic zones. Note that the `global alloc` and `all alloc` zones require a minimum amount of space for internal UPC housekeeping, so they will never be 0, but they will be a minimum size if there are no calls to those allocators in the application. If there are no application calls to allocate from the `local alloc` zone, then no memory will be reserved for the `local alloc` zone. Thus, the RTS strives to reserve the largest zones possible based on the perceived usage of the zone.

Refer to the Quadrics Elan documentation for more information.

7.4 Environment Variables

The UPC Run-Time System (UPCRTS) supports diverse behavior controls using the environment variables described in this section. See the Section A.3 for information on environment variables that affect the behavior of UPC applications running in an SC environment.

Some of the variables in this section are described as **boolean**. Such variables are considered *FALSE* if they are not defined, or if they are defined to *FALSE*, *F*, *OFF*, or *0*. They are considered *TRUE* if they are defined with no value, or if they are defined to *TRUE*, *T*, *ON*, or *1*.

7.4.1 Performance Control Variables

By default, the UPCRTS caches remote operations; the caching mechanism causes the UPCRTS to expand remote fetch operations to a cache block sized region containing the requested address. Subsequent fetches from nearby remote addresses may be satisfied from the cached results of the expanded remote fetch, eliminating the need for remote operations. A four-way set associative cache method is used. Note that the cache is only used if the cache block size is greater than zero and there are more than zero cache blocks. Note that caching is not used in SMP environments.

Table 7–1 Caching Control Variables

Variable	Type	Default	Description
UPCRTS_USE_CACHE	Boolean	True	Turns the use of the cache on or off.
UPCRTS_CACHE_SETS	Integer	128	Indicates how many cache sets (lines) to use per thread. Must be greater than zero.
UPCRTS_CACHE_BLOCK_SIZE	Integer	64	Indicates the size in bytes of a cache block. Permitted values are the powers of 2 from 64 bytes up to 8192 bytes.
UPCRTS_CACHE_ASSOCIATIVITY	Integer	4	Indicates the number of cache blocks per cache set. Must be in the range 1 to 128.
UPCRTS_DISP_CACHE_STATISTICS	Boolean	False	Displays information about cache performance after the program is run. Only operates if the cache is turned on.

Note that the total number of cache bytes available per thread is:

```
SETS * BLOCK_SIZE * ASSOCIATIVITY
```

7.4.2 Diagnostic Library and Output Control Variables

The Standard Library provided with the RTS, `/usr/lib/cmplrs/upc/libupc.so` on Tru64 UNIX and `/opt/upc/lib/libupc.sl` on HP-UX and XC Linux, cannot emit debugging diagnostic information. Another library, `/usr/lib/cmplrs/upc/diagnostics/libupc.so` on Tru64 UNIX and `/opt/upc/lib/diagnostics.sl` on HP-UX and XC Linux, maintains this capability at a small performance cost. If you need the diagnostic library, include `/usr/lib/cmplrs/upc/diagnostics` in the definition of the environment variable `LD_LIBRARY_PATH`.

By default, the output stream for UPCRTS statistics reporting and logging is `stdout`, and for diagnostics is `stderr`. These output streams can be modified by setting the appropriate variable from Table 7–2 to a file name. When run with Quadrics RMS software on an AlphaServer SC system, these files have a *.threadnumber* appended to their name, one for each separate thread number. An application that exits with `upc_global_exit` might not generate output for certain run-time statistics.

Table 7–2 Diagnostic Library and Output Control Variables

Variable	Type	Default	Description
UPCRTS_DISP_CACHE_STATISTICS	Boolean	False	Displays information about cache performance after the program is run. Only operates if the cache is turned on.
UPCRTS_STATISTICS_STREAM	File	None	Changes statistics reporting output stream from `stdout` to the specified file.
UPCRTS_LOGGING_STREAM	File	None	Changes logging output stream from `stdout` to the specified file.
UPCRTS_DIAGNOSTICS_STREAM	File	None	Changes diagnostic output stream from `stderr` to the specified file.
UPCRTS_SUPPRESS_LOCAL_ALLOC_ MESSAGE	Boolean	None	Suppresses display of the informational message indicating the obsolesense of `upc_local_alloc`.

7.4.3 Configuration Variables

These variables control aspects of UPCRTS normal operation.

Table 7–3 Configuration Variables

Variable	Type	Default	Description
UPCRTS_BUFFERS	Integer	1024	Number of memory buffers for controlling the remote memory operations supported by the Quadrics switch software. The default value has proven adequate for small load tests. However, larger loads may benefit from allocating a larger numbers of buffers at UPCRTS startup time.
UPCRTS_LOCK_TYPE	Lock Type	GREEDY	Two algorithms may be used for obtaining locks: `FAIR` or `GREEDY`.

7.4.4 Display Variables

The UPCRTS can display information summarizing its behavior after the program is run. All display variables are **boolean** variables. Setting any of these variables to *TRUE* causes the UPRTS to write summary information to `UPCRTS_STATISTICS_STREAM`.

Note

Use of these variables requires the diagnostic library.

Table 7–4 Display Variables

Variable	Description
UPCRTS_DISP_TOTALTIME	Total system, user, and elapsed time.
UPCRTS_DISP_TOTALCOMM	Total number of local and remote GET and PUT operations.
UPCRTS_DISP_TOTALSYNCS	Number of **upc_wait** and **upc_barrier** operations performed.
UPCRTS_DISP_TOTALGETS	Number of GET operations.
UPCRTS_DISP_TOTALPUTS	Number of PUT operations.
UPCRTS_DISP_LOCALGETS	Number of GET operations for data with affinity to the requesting thread.
UPCRTS_DISP_LOCALPUTS	Number of PUT operations for data with affinity to the requesting thread.
UPCRTS_DISP_REMOTEGETS	Number of GET operations for data with affinity to other threads.
UPCRTS_DISP_REMOTEPUTS	Number of PUT operations for data with affinity to other threads.
UPCRTS_DISP_BARRIERTIMES	Documents barrier operations as they are performed, rather than just at the end of the program run.
UPCRTS_DISP_TOTALGETSIZES	Displays local bytes fetched, remote bytes fetched, remote GET operations per second, and total GET operations per second.
UPCRTS_DISP_TOTALPUTSIZES	Displays local bytes written, remote bytes written, remote PUT operations per second, and total PUT operations per second.
UPCRTS_DISP_MAXPUTS	Peak number of PUTs issues between PutAllSyncs (fences) or barriers. In strict mode, this should never exceed one.
UPCRTS_DISP_BUFFER_WAITS	Number of times the UPCRTS had to wait for a remote operation to complete in order to obtain a buffer for an operation.
UPCRTS_DISP_CACHE_STATISTICS	Only has effect when caching is turned on.
UPCRTS_DISP_ALLSTATISTICS	Activates all the other statistics display variables. Also causes global allocation overhead information to be displayed.

A
Recovering from Errors

This appendix provides information to help you deal with failures or errors that might occur during product installation or product use. See Section A.3 for additional information on troubleshooting memory problems.

If you find an error in the documentation or would like to provide suggestions for improvement, please send mail to the following Internet address:

hp_upc@hp.com

Include the title of the document, section, and page number where appropriate.

A.1 Failures During Product Installation

If errors occur during the installation, the system displays failure messages. For more information, see the UPC Installation Guide.

A.2 Failures During Product Use

If an error occurs while HP UPC is in use and you believe the error is caused by a problem with the product, please report it to HP.

Please provide as much detail as possible (problem description, compiler version, OS version, hardware, and so on).

Customers with support contracts should seek support for problems through local customer support centers.

Customers who do not have support contracts are encouraged to mail problem reports to compaq_upc.support@hp.com. Although these reports will certainly be used as a source of input for fixing problems for new releases, we cannot give the reports individual attention. We can take remedial action only on a best-effort basis.

A.3 UPC RTS Run-Time Errors

This section describes UPC RTS Run-time errors.

The RTS detects the following run-time error conditions and their causes. You can take the suggested corrective actions.

Condition: Static shared data exceeds the maximum shared memory segment size

Cause: The amount of statically declared shared data exceeds the current system configuration limits for shared memory maximum.

RTS Message:

```
Static shared data (size: XXX) exceeds maximum shared memory segment size (YYY)
```

Corrective action: On Tru64 UNIX systems, you can increase the shared memory max size (`shm_max`) using the `sysconfigdb` utility.

Condition: Static shared data exceeds the Elan Galloc memory size

Cause: The amount of statically declared shared data exceeds the current Galloc region within `Libelan`.

RTS Message:

```
Static shared data (size: XXX) exceeds Elan galloc size (YYY)
Try increasing global memory size via the LIBELAN_GALLOC_SIZE environment variable
```

Corrective action: As the message suggests, you can increase the Galloc region with the `LIBELAN_GALLOC_SIZE` environment variable. Note: Because the Galloc region is a partition of the Alloc region, you might also need to increate the `LIBELAN_ALLOC_SIZE` accordingly.