

Lecture 13: March 10

*Lecturer: Kathy Yelick**Scribes: Terry Filiba*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

13.1 Background

Most parallel systems are programmed as

- message passing clusters - scalable but difficult to program (programmable with languages like MPI)
- shared memory - easier to program (supported by languages such as OpenMP) but not very scalable

Global address space languages are the best of both

- They allow pointers across machines
- They provide a local vs global distinction. Global variables may be far away and have a high performance impact.
- A shared memory machine is not required, providing scalability of a message passing cluster

Unified Parallel C (UPC) is a global address space language

UPC design features:

- A parallel extension of ISO C
- UPC puts programmers close to the hardware (allows the programmer to get good performance but can be a source of bugs)
- The number of threads fixed throughout application execution
- UPC provides barrier (all wait) and parallel loop constructs

UPC is highly available with commercial Cray, SG1, HP compilers, and open source compilers available from LBNL and UC Berkeley

13.2 Execution Model

Compilation options

- static threads: the number of threads is determined at compile time, this may allow some optimizations (especially for powers of 2)

- dynamic threads: the compiler allows for varying numbers of threads

Any legal C program is a legal UPC program

Each thread gets the variable MYTHREADS, which has the thread number of the current thread, and THREADS, which is the total number of threads

13.3 Memory Model

Standard C declarations provide each thread with its own copy of that variable, shared declarations create one global instance of the variable on thread 0

Here is an example of local vs global declarations:

```
shared int ours; // allocated only once on thread 0
int mine; // allocated to each thread
```

Shared variables may not have a dynamic lifetime (if it appears in a function definition it must be static)

Shared variables can create a data race, or performance problems

Each thread gets a copy of argv and argc

Shared arrays are arrays are distributed across the threads

Here is an example of shared arrays (refer to slide 16 for a graphical representation of these arrays)

```
shared int x[THREADS]; //distributes 1 element to each thread
shared int y[3][THREADS]; //distributes 3 elements to each thread
shared int z[3][3]; //distributed by looping through each processor and wrapping

x[MYTHREAD]; //access own element
```

13.4 Synchronization

13.4.1 Barriers

UPC provides two types of barriers

blocking barrier - forces a thread to stop and wait on all threads to arrive (can be high latency)

- upc_barrier - stop execution and notify other threads it has reached this barrier

split phase barrier - allows a thread to notify but continue doing useful work

- upc_notify - thread ready for barrier, does not stop computation, do some unrelated computation
- upc_wait - need other threads to be finished, wait until they all reach notify

barriers may be labeled

```
upc_barrier MERGE_BARRIER //can be present in different places
```

13.4.2 Locks

Similar to POSIX threads

UPC provides an opaque lock type, `upc_lock_t`, and related functions

- `upc_all_lock_alloc(void)` - allocates 1 lock, pointer to all threads (all in the function name indicates that all threads should call this function)
- `upc_global_lock_alloc` - allocates 1 lock and 1 thread gets a pointer (can be broadcast later)
- `upc_lock`
- `upc_unlock`

UPC locks are implemented as spin locks (based on the assumption that no one can do useful work at the same time, like on a batch system)

UPC locks are not intensively used in practice

13.5 Collectives

UPC collectives provide functions for collective communication such as

- `TYPE bupc_allv_reduce(TYPE, TYPE value, int rootthread, upc_op_t reductionop)`
- `TYPE bupc_allv_reduce_all(TYPE, TYPE value, upc_op_t reductionop)`
- `TYPE bupc_allv_broadcast(TYPE, TYPE value, int rootthread)`

(refer to slide 26 for more collective functions)

all in the function name means all threads will perform the operation. otherwise only rootthread will perform the operation

UPC collective functions imply a barrier

13.6 Work Distribution

Distributed loops in UPC are something like OpenMP

UPC provides the `upc_forall` construct that distributes the loop throughout the threads

`upc_forall` requires an initialization, test, loop operation (such as counter increment), and affinity (if this thread should execute this iteration)

Loop iterations should be independent

13.7 Memory Layout

Cyclic wrapping (the default array layout) may not be optimal.

Alternative layouts can be specified in the loop declaration

```
shared int x[3][3]; //cyclic layout of array
shared int [0] y[3][3]; //indefinite layout, all elements will be in the address space of a single thread
shared [*] int z[3][3]; //blocked layout
```

UPC allows dynamic memory allocation to the global address space/
upc_global_alloc is the UPC analog to alloc, each thread that calls this function gets a different pointer and the data is allocated to the global address space
upc_all_alloc is called by all threads and each thread gets the same pointer back

13.8 Performance

UPC provides good performance compared to MPI
For multi-core nodes 1 MPI thread per core is inefficient. The shared memory should be put to use since it is more efficient than message passing.
UPC communication relies on GASNet communication
GASNet provides lower latencies than MPI and higher bandwidth for small to mid-size packets