COMS/CPRE 425
Spring 2005
Lecture 30

Ricky A. Kendall
rickyk@cs.iastate.edu
Logistics

• **G4 cluster issues (4pack)**
  - If you have trouble with the system email help@scl.ameslab.gov and copy me on the email!
  - Try to submit only 1 job at a time
    - Remember SSS is “beta” software

• **Homework 5 issues**
  - Any of them??

• **Presentation Schedule**
  - Is now online!
## Presentation Schedule

<table>
<thead>
<tr>
<th>Group</th>
<th>Members</th>
<th>Date</th>
<th>Time</th>
<th>Reviewing</th>
<th>Reviewing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Seth Bacon</td>
<td>Monday 04/25/2005</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Katie Becker</td>
<td>Wednesday 04/20/2005</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>David Rodgers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Nichole Burgett</td>
<td>Friday 04/22/2005</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Emily Ericson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Bill Maples</td>
<td>Monday 04/25/2005</td>
<td>2</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>Sam Miller</td>
<td>Monday 04/18/2005</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Jeff Russell</td>
<td>Wednesday 04/27/2005</td>
<td>1</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Jake Johnson</td>
<td>Friday 04/22/2005</td>
<td>2</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>David Gregg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Weston Lahr</td>
<td>Friday 04/29/2005</td>
<td>1</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Matt Myers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Brice Lambi</td>
<td>Monday 04/18/2005</td>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>James Kurtz</td>
<td>Wednesday 04/27/2005</td>
<td>2</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Luis Martinez</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>David Gillingham</td>
<td>Friday 04/29/2005</td>
<td>2</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Michael McCown</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>David Bireta</td>
<td>Wednesday 04/27/2005</td>
<td>3</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Shaun Lindsay</td>
<td>Wednesday 04/20/2005</td>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Unified Parallel C

• **UPC**
  - An extension to C-99 ANSI standard
    - Extensions are explicit parallelism
    - Familiar syntax and control is high-level or low-level as needed!
  - A distributed shared memory programming model
    - Language based not library based like Global Arrays or SHMEM
  - Current support is limited but available in some fashion on many systems
    - SMP on linux, Solaris, etc.
    - Cluster based on Linux
    - Vendor support
      - Cray, HP
  - Designed to deliver performance with ease of use!
  - Research supported in part by the Programming Models effort. [http://www.pmodels.org](http://www.pmodels.org)
DSM Memory Model (processes)

Shared
Global

Private
Local

Process  Process  Process  Process  Process
DSM Memory Model (Threads)
DSM Memory Model (SMP cluster)
UPC execution model

- A collection of threads with independent instruction streams
  - MYTHREAD “macro” is the thread ID
  - Number of threads is the THREADS “macro”
    - Numbered from 0 to THREADS-1
  - SPMD programming model
    - Not as “easy” as OpenMP but much more straightforward than Pthreads!
  - Synchronization mechanisms
  - Memory Consistency mechanisms
    - Different kind of synchronization
  - Cluster based implementations “hide” the runtime exercise of moving data from one “node” to another!
    - Same code runs SMP, Distributed, Cluster of SMP
    - Performance is still an issue!!
UPC Memory Model

- **Variables are**
  - “shared” or
  - Private

- **Pointers are**
  - “to Shared” or private
  - Pointers to shared can point to any address in the shared space
    - Part of the shared space has some “affinity” to a given thread
  - Private pointers can point to:
    - Local addresses or the “shared” space of the thread
  - Dynamic and static allocation of memory is valid for both shared and local addresses
#include “upc.h”
#define LEN THREADS*1000
shared int v1[LEN], v2[LEN], diff[LEN];
int main (int argc, char *argv[])
{
    int j;
    for (j = 0; j<LEN; j++)
        if (j % THREADS == MYTHREAD)
            diff[j] = v1[j] - v2[j];
}
**upc forall example**

```c
#include "upc.h"
#define LEN THREADS*1000
shared int v1[LEN], v2[LEN], diff[LEN];
int main (int argc, char *argv[])
{
    int j;
    upc_forall (j = 0; j<LEN; j++; j)
        diff[j] = v1[j] – v2[j];
}
```
The shared qualifier

• **When applied to scalars**
  - The variable is shared with affinity to thread 0
  - **shared double timeval**
    - One variable among all threads lives in thread 0 shared space
  - **double timelocal**
    - Each thread has a “timelocal”

• **When applied to vectors elements are distributed among threads**
  - **shared double X[THREADS]**
    - One element per thread
  - **shared double Y[100][THREADS]**
    - 100 elements per thread
The shared qualifier [2]

• Can be applied to pointers!
  ➢ shared double *p1
  ➢ This is a pointer to a double somewhere in the shared address space!
    ▪ Called a pointer to shared
Examples of private and shared data

THREADS = 3 (assumption)
shared int A;
shared double B[THREADS];
int C;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B[0]</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Examples of private and shared data

shared double B[2][3*THREADS];
Shared arrays can be blocked

- Shared arrays are distributed
  - With a block per thread
  - With arbitrary block size
    - Default is with a block size of 1
  - Round robin per thread
- shared [blocksize] type array[length]
  - Blocksize is optional
  - shared [3] double xyz [99];
- Affinity to the thread is determined by the block size and THREADS
  \[
  \left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS}
  \]
Shared data distribution

Assume THREADS = 3
shared [2] double B[3][THREADS];

Thread 0

B[0][0]
B[0][1]
B[2][0]
B[2][1]

Thread 1

B[0][2]
B[1][0]
B[2][2]

Thread 2

B[1][1]
B[1][2]
UPC Pointers

- There are essentially 4 kinds of pointers
  - int *ppp;
    - Private pointer pointing to data that is private to the thread
  - shared int *pps;
    - Private pointer pointing to data that is in shared space
  - int *shared psp;
    - Shared pointer pointing to data in thread local space
  - shared int *shared pss;
    - Shared pointer pointing to data in shared space
View of these pointers
Standard C and UPC Pointers

- Virtual Address is the local address of the block
- Phase is the offset into the block
UPC Pointers

• **Pointer arithmetic works for shared pointers**
  ➢ Regardless of the distribution

• **Casting a pointer-to-shared to a private pointer**
  ➢ Is allowed
    ▪ The reverse is not allowed!
  ➢ Well defined only if the shared address has affinity with the thread that is performing the cast
  ➢ Thread number information can be lost!
Pointer Arithmetic

shared double *pd=&B[4];  Assume THREADS = 3
shared double B[12];  shared double *pdoff;
pdoff = pd + 4;
**Pointer Functions for shared pointers**

- **size_t upc_threadof(shared void *ptr);**
  - Returns the thread id of the thread that the has the affinity of the address pointed to by ptr

- **size_t upc_addrfield(shared void *ptr);**
  - Returns the address of the block of memory in shared space pointed to by ptr

- **size_t upc_phaseof(shared void *ptr);**
  - Returns the index within the block of the shared space block of memory pointed to by ptr;

- **size_t upc_resetphase(shared void *ptr);**
  - Zeros the phase component of the ptr
  - Basically resets the full ptr back to the beginning of the shared space block of memory
Byte based routines

• **memcpy-like routines**
  - `upc_memcpy(dst,src,size)`
    - Copy from shared space to shared space
  - `upc_memput(dst,src,size)`
    - Copy from private space to shared space
  - `upc_memget(dst,src,size)`
    - Copy from shared space to private space

• **memset-like routine**
  - `upc_memset(dst,char,size)`
    - Initialize a shared memory segment with the value of char
Work Sharing with upc_forall

- Upc_forall is a parallel loop work sharing construct
  - Similar to OpenMP omp for or do
  - upc_forall is a collective operation
- The loop syntax is
  upc_forall( j = 0 ; j < 1000; j++; affinity)
  {
      work
  }
- The first three components of the forall are identical to the C standard for syntax.
- **affinity** is either a keyword *continue* or and expression that is of either type *pointer-to-shared or integer*
What is the for_all affinity

- **If** `affinity` **is continue**
  - Then the loop body is executed on all threads
  - The programmer may control (based on MYTHREAD) which iteration is executed
    - Manual control of concurrency
- **If** `affinity` **is an integer expression**
  - The loop body is executed such that the value
    `MYTHREAD == (affinity \mod THREADS)` controls which iteration is executed by which thread
- **If** `affinity` **is a pointer-to-shared**
  - The loop body is executed such that the value
    `MYTHREAD == \text{upc\_threadof}(affinity)` controls which iteration is executed by which thread.
  - An “owner-compute” like construct
upc_forall restrictions

• In nested upc_forall constructs the outer most loop that has a non “continue” affinity is the loop that is executed concurrently across threads.
  ➢ All other loops are “serialized”

• Some constructs are not allowed in the upc_forall loop body
  ➢ upc_barrier
  ➢ break
  ➢ goto

• cannot branch/jump into a loop body either
upc_forall examples

ASSUME THREADS = 4

shared double a[100], b[100], c[100];    int j;
upc_forall(j=0;j<100;j++;j)
    c[j] = a[j] + b[j];

• This will put the iterations round-robin on each thread
  ➢ Thread 0 will get iterations 0, 4, 8, 12, ..., 96
  ➢ Thread 1 will get iterations 1, 5, 9, 13, ..., 97
  ➢ Thread 2 will get iterations 2, 6, 10, 14, ..., 98
  ➢ Thread 3 will get iterations 3, 7, 11, 15, ..., 99
upc_forall examples[2]

ASSUME THREADS = 4
shared double a[100], b[100], c[100];   int j;
upc_forall(j=0;j<100;j++;&c[j])
   c[j] = a[j] + b[j];

• This will put the iterations round-robin on each thread because the distribution of c is that way!
  ➢ Thread 0 will get iterations 0, 4, 8, 12, ..., 96
  ➢ Thread 1 will get iterations 1, 5, 9, 13, ..., 97
  ➢ Thread 2 will get iterations 2, 6, 10, 14, ..., 98
  ➢ Thread 3 will get iterations 3, 7, 11, 15, ..., 99
upc\_forall examples[3]

ASSUME THREADS = 4
shared double a[100], b[100];
shared 10 double c[100]; int j;
upc\_forall(j=0;j<100;j++;&c[j])
c[j] = a[j] + b[j];

• This will put the iterations in round-robin chunks on each thread based on the the distribution of c
  ➢ Thread 0 will get iterations 0..9, 40..49, 80..89
  ➢ Thread 1 will get iterations 10..19, 50..59, 90..99
  ➢ Thread 2 will get iterations 20..29, 60..69
  ➢ Thread 3 will get iterations 30..39, 70..79
  ▪ Note the load imbalance!
upc_forall examples[4]

ASSUME THREADS = 4
shared double a[100], b[100], c[100]; int j;
upc_forall(j=0;j<100;j++;j/10)
c[j] = a[j] + b[j];

• This will put the iterations in chunks round-robin on each thread!
  ➢ Thread 0 will get iterations 0..9, 40..49, 80..89
  ➢ Thread 1 will get iterations 10..19, 50..59, 90..99
  ➢ Thread 2 will get iterations 20..29, 60..69
  ➢ Thread 3 will get iterations 30..39, 70..79
    ▪ Note the load imbalance!
    ▪ Note the number of non-thread-local calls per thread based on round robin C distribution
upc_forall examples[5]

ASSUME THREADS = 4
shared double a[100], b[100], c[100]; int j;
upc_forall(j=0; j<100; j++;(j*THREADS/100) )
c[j] = a[j] + b[j];

• This will also put the iterations in “even” chunks on each thread!
  ➢ Thread 0 will get iterations 0..24
  ➢ Thread 1 will get iterations 25..49
  ➢ Thread 2 will get iterations 50..74
  ➢ Thread 3 will get iterations 75..99
  ▪ Note the same as the default OpenMP construct
Dynamic Memory Allocation

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

  - Non collective
  - Shared space allocated “contiguous” chunk of memory
    - One region per call
  - Equivalent to `shared [nbytes] char[nblocks*nbytes]`
    - For each allocated memory segment
  - Behavior is just like `malloc` accept for where memory allocated lives
    - Each call allocates a region
    - Separate pointers.
Dynamic Memory Allocation [2]

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

  - A collective call
  - All participating threads must call
  - Returns one pointer to one shared space contiguous memory segment
    - All threads have the same pointer
  - Equivalent to `shared [nbytes] char[nblocks*nbytes]`
    - Only one segment shared!
Dynamic Memory Allocation [3]

- **shared void *upc_alloc(size_t nbytes)**
  - Non-collective
  - Allocates a shared space contiguous memory segment with affinity to the calling thread
  - So it is “malloc” for “local shared” address space

- **void upc_free(shared void *ptr)**
  - Frees memory allocated by upc_global_alloc, upc_all_alloc or upc_alloc
  - Subtle behavior is that only one upc_free should be called for each segment of shared space memory
    - Everyone calls upc_all_alloc only one call needed from one of the participating threads needed to free it.
Synchronization mechanisms

• UPC offers 2 kinds of barriers, fence operations and locks
  • upc_barrier [integer expression]
  • upc_fence
    ➢ Ensures that all shared references issued before the fence are completed before any shared references after the fence are issued
    ➢ Just like the Global Array fence operation!
Synchronization mechanisms [2]

- A split phase barrier
  - `upc_notify [integer expression]`
    - The first half of a barrier
    - Non blocking call
  - `upc_wait [integer expression]`
    - Second half of a barrier
    - Blocks until all threads make the matching `upc_notify` call
  - Think about this as the inverse of a critical section
    - There is an implied fence before the notify and after the wait!
UPC Locks

- Very similar to Mutex Locks in Pthreads
- UPC locks of upc_lock_t

  `upc_lock_t *upc_global_lock_alloc(void);`
  - Non collective call and allocates a shared space lock.
    - One lock created per call
  - Initialized and created in the unlocked state.

  `upc_lock_t *upc_all_lock_alloc(void);`
  - Collective call and allocates a shared space lock
    - One lock created all threads have the same lock pointer
  - Initialized and created in the unlocked state

- `upc_lock_free(upc_lock_t *lock)`
  - Frees locks and all resources associated with the lock (dynamics ones).
UPC Locks [2]

- To protect a critical section of code

  - `void upc_lock(upc_lock_t *lock);`
    - Obtains the lock.
    - Waits if another thread has the lock
  
  - `void upc_unlock(upc_lock_t *lock);`
    - Releases the lock
  
  - `int upc_lock_attempt(upc_lock_t *lock);`
    - Attempts to obtain the lock.
      - If it gets the lock returns 1
    - If the lock is unavailable returns 0 immediately
      - Similar to pthread_mutex_trylock
UPC Memory Consistency Model

- The consistency model controls the ordering of shared operations in a global context.
  - Relaxed
    - Shared operations can be reordered by the compiler and/or runtime system
  - Strict
    - Enforces sequential ordering of shared operations
      - All shared data movements complete before new ones begin!
  - Programmers are responsible for using the correct model.
  - Can change during the execution of the code
  - Specified via
    - Declarations
    - Pragma statements
    - Use of barriers, fences, global operations etc.
Consistency model

• **Declarations**
  - strict shared int y
  - relaxed shared double x

• **Include files**
  - #include <upc_strict.h>
  - #include <upc_relaxed.h>

• **Pragmas**
  - #pragma upc strict
  - #pragma upc relaxed
  - Single statement or block of code like the pragmas within the OpenMP programming model.
Our favorite parallel problem

- **Matrix Multiplication** \((C = AB)\)
- **You can control the patch distribution**
  - For \(A\) decompose it row-wise
    - Multiple rows of \(A\) are distributed among threads
    - Dynamic distribution via allocation!
  - For \(B\) decompose it column-wise
    - Multiple Columns of \(B\) are distributed among threads
  - The work is a patch of rows of \(A\) multiply a patch of columns of \(B\) to give the patch of \(C\) being computed!

- **Optimizations**
  - Use `upc_forall` with affinity for \(C\) so there is no “extra” communication for the result
  - Use `upc_memget` to pull \(B\), \(A\), or both local and compute the contribution
  - Can be blocked, daxpy rearranged etc.