Recap of the last lecture (MPI)

- MPI is an industry standard model for parallel programming.

- MPI processes have own address space. Synchronization is made by explicit message passing.

- Users have finest control of data layout. Generally MPI is the fastest and most scalable parallel programming paradigm.
Outline

• Why PGAS Languages?
• UPC Execution Model
• UPC Memory Model
• Distributed Array in UPC
• UPC Synchronization
• Summary
Shared Memory vs. Message Passing

**Shared Memory**
- **Convenient:**
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- **Disadvantages**
  - No locality control
  - Does not scale
  - Race conditions

**Message Passing**
- **Scalable**
  - Locality control
  - Communication is all explicit in code (cost transparency)
- **Disadvantage**
  - Need to rethink entire application / data structures
  - Lots of tedious pack/unpack code
  - Don’t know when to say “receive” for some problems
Programming Challenges and Solutions

**Message Passing Programming**
Divide up domain in pieces
Each compute one piece
Exchange (send/receive) data

*PVM, MPI, and many libraries*

**Global Address Space Programming**
Each start computing
Grab whatever you need whenever

*Global Address Space Languages and Libraries*
PGAS Languages: Combines Best of Both

• **Global address space**: thread may directly read/write remote data
  • Hides the distinction between shared/distributed memory
• **Partitioned**: data is designated as local or global
  • Does not hide this: critical for locality and scaling

```
<table>
<thead>
<tr>
<th>l:</th>
<th>l:</th>
<th>l:</th>
<th>l:</th>
<th>l:</th>
</tr>
</thead>
<tbody>
<tr>
<td>g:</td>
<td>g:</td>
<td>g:</td>
<td>g:</td>
<td>g:</td>
</tr>
</tbody>
</table>
```

- **UPC, CAF, Titanium**: Static parallelism (1 thread per proc)
  • Does not virtualize processors
- **X10, Chapel and Fortress**: PGAS, but not static (dynamic threads)
History of UPC

• Initial Tech. Report from IDA in collaboration with LLNL and UCB in May 1999 (led by IDA).
  – Based on Split-C (UCB), AC (IDA) and PCP (LLNL)

• UPC consortium participants (past and present) are:
  – *UPC is a community effort, well beyond UCB/LBNL*

• Design goals: high performance, expressive, consistent with C goals, ..., portable

• UPC Today
  – Multiple vendor and open compilers (Cray, HP, IBM, SGI, gcc-upc from Intrepid, Berkeley UPC)
  – *“Pseudo standard”* by moving into gcc trunk
  – Most widely used on irregular / graph problems today
Outline

• Why PGAS Languages?
• UPC Execution Model
• UPC Memory Model
• Distributed Array in UPC
• UPC Synchronization
• Summary
UPC Threads

• A number of threads working independently in a SPMD fashion
  – Number of threads specified at compile-time or run-time; available as program variable \textbf{THREADS}
  – \textbf{MYTHREAD} specifies thread index
    \((0..\text{THREADS}-1)\)
  – \texttt{upc\_barrier} is a global synchronization: all wait
Hello World in UPC

• Any legal C program is also a legal UPC program
• If you compile and run it as UPC with P threads, it will run P copies of the program.
• Using this fact, plus the identifiers from the previous slides, we can parallel hello world:

```c
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

main() {
    printf("Thread %d of %d: hello UPC world\n", 
           MYTHREAD, THREADS);
}
```
Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} \pi r^2 = \pi/4$
- Randomly throw darts at $x,y$ positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - $\pi = 4 \times \text{ratio}$
Pi in UPC

• Independent estimates of pi:

```c
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;
    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);
    srand(MYTHREAD*17);
    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” separately
Helper Code for Pi in UPC

- Required includes:
  ```
  #include <stdio.h>
  #include <math.h>
  #include <upc.h>
  ```

- Function to throw dart and calculate where it hits:
  ```c
  int hit(){
      int const rand_max = 0xFFFFFFFF;
      double x = ((double) rand()) / RAND_MAX;
      double y = ((double) rand()) / RAND_MAX;
      if ((x*x + y*y) <= 1.0) {
          return(1);
      } else {
          return(0);
      }
  }
  ```
Outline

• Why PGAS Languages?
• UPC Execution Model
• UPC Memory Model
• Distributed Array in UPC
• UPC Synchronization
• Summary
Private vs. Shared Variables in UPC

• Normal C variables and objects are allocated in the private memory space for each thread.
• Shared variables are allocated only once, with thread 0

```c
shared int ours; // use sparingly: performance
int mine;
```

• Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static.
Pi in UPC: Shared Memory Style

• Parallel computing of pi, but with a bug

```c
shared int hits;
main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        hits += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
        printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

shared variable to record hits

divide work up evenly

accumulate hits

What is the problem with this program?
Shared Arrays Are Cyclic By Default

• Shared scalars always live in thread 0
• Shared arrays are spread over the threads
• Shared array elements are spread across the threads
  
  ```
  shared int x[THREADS]        /* 1 element per thread */
  shared int y[3][THREADS]     /* 3 elements per thread */
  shared int z[3][3]           /* 2 or 3 elements per thread */
  ```

• In the pictures below, assume THREADS = 4
  – Red elts have affinity to thread 0

Think of linearized C array, then map in round-robin

As a 2D array, y is logically blocked by columns

z is not
Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```c
shared int all_hits [THREADS];
main(int argc, char **argv) {
    ... declarations an initialization code omitted
    for (i=0; i < my_trials; i++)
        all_hits[MYTHREAD] += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
        for (i=0; i < THREADS; i++) hits += all_hits[i];
        printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

All _hits is shared by all processors, just as hits was update element with local affinity
Outline

• Why PGAS Languages?
• UPC Execution Model
• UPC Memory Model
• Distributed Array in UPC
• UPC Synchronization
• Summary
Blocked Layouts in UPC

• If this code were doing nearest neighbor averaging (3pt stencil) the cyclic layout would be the worst possible layout.
• Instead, want a blocked layout
• Vector addition example can be rewritten as follows using a blocked layout

```c
#define N 100*THREADS
shared int [*] v1[N], v2[N], sum[N];  // blocked layout

void main() {
    int i;
    upc_forall(i=0; i<N; i++; &sum[i]) {
        sum[i]=v1[i]+v2[i];
    }
}
```
Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - Empty (cyclic layout)
  - [*,] (blocked layout)
  - [0] or [] (indefinite layout, all on 1 thread)
  - [b] or [b1][b2]...[bn] = [b1*b2*...bn] (fixed block size)
- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element i has affinity with thread
  \[(i / \text{block\_size}) \mod \text{THREADS}\]
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping
2D Array Layouts in UPC

• Array a1 has a row layout and array a2 has a block row layout.

\[
\text{shared } [m] \text{ int } a1 [n][m]; \\
\text{shared } [k*m] \text{ int } a2 [n][m]; \\
\]

• If \((k + m) \% \text{THREADS} = 0\) then a3 has a row layout

\[
\text{shared int } a3 [n][m+k]; \\
\]

• To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.

• Assume \(r*c = \text{THREADS}\);

\[
\text{shared } [b1][b2] \text{ int } a5 [m][n][r][c][b1][b2]; \\
\]

• or equivalently

\[
\text{shared } [b1*b2] \text{ int } a5 [m][n][r][c][b1][b2]; \\
\]
Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
- Here is the vector addition example using pointers

```c
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    shared int *p1, *p2;  
    p1=v1; p2=v2;
    for (i=0; i<N; i++, p1++, p2++)
        if (i % THREADS == MYTHREAD)
            sum[i] = *p1 + *p2;
}
```
**UPC Pointers**

Where does the pointer point?

<table>
<thead>
<tr>
<th>Where does the pointer reside?</th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>Shared</td>
<td>p3</td>
<td>p4</td>
</tr>
</tbody>
</table>

int *p1;           /* private pointer to local memory */
shared int *p2;    /* private pointer to shared space */
int *shared p3;    /* shared pointer to local memory */
shared int *shared p4; /* shared pointer to shared space */

Shared to local memory (p3) is not recommended.
UPC Pointers

- `int *p1;` /* private pointer to local memory */
- `shared int *p2;` /* private pointer to shared space */
- `int *shared p3;` /* shared pointer to local memory */
- `shared int *shared p4;` /* shared pointer to shared space */

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.
Outline

• Why PGAS Languages?
• UPC Execution Model
• UPC Memory Model
• Distributed Array in UPC
• UPC Synchronization
• Summary
UPC Global Synchronization

• UPC has two basic forms of barriers:
  – Barrier: block until all other threads arrive
    `upc_barrier`
  – Split-phase barriers
    `upc_notify;` this thread is ready for barrier
    `do computation unrelated to barrier`
    `upc_wait;` wait for others to be ready

• Optional labels allow for debugging
  `#define MERGE_BARRIER 12`
  `if (MYTHREAD%2 == 0) {`
    `...`
    `upc_barrier MERGE_BARRIER;`
  } else {
    `...`
    `upc_barrier MERGE_BARRIER;`
  }`
Synchronization - Locks

- Locks in UPC are represented by an opaque type:
  ```c
  upc_lock_t
  ```

- Locks must be allocated before use:
  ```c
  upc_lock_t *upc_all_lock_alloc(void);
  ```
  allocates 1 lock, pointer to all threads
  ```c
  upc_lock_t *upc_global_lock_alloc(void);
  ```
  allocates 1 lock, pointer to one thread

- To use a lock:
  ```c
  void upc_lock(upc_lock_t *l)
  void upc_unlock(upc_lock_t *l)
  ```
  use at start and end of critical region

- Locks can be freed when not in use
  ```c
  void upc_lock_free(upc_lock_t *ptr);
  ```
Pi in UPC: Shared Memory Style

- Parallel computing of pi, without the bug

```c
shared int hits;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        my_hits += hit();
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*hits/trials);
}
```
Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi example
  - Private scalars (`my_hits`)
  - Shared scalars (`hits`)
  - Shared arrays (`all_hits`)
  - Shared locks (`hit_lock`)

![Diagram of Global address space](image)

- `hits`: Global address
- `hit_lock`: Global address
- `all_hits[0]`, `all_hits[1]`, ..., `all_hits[n]`: Global address
- `my_hits`: Global address, `Thread_0`, `Thread_1`, ..., `Thread_n`

where: `n = Threads - 1`
A Family of PGAS Languages

- UPC based on C philosophy / history
  - http://upc-lang.org
  - Free open source compiler: http://upc.lbl.gov
  - Also a gcc variant: http://www.gccupc.org
- Java dialect: Titanium
  - http://titanium.cs.berkeley.edu
- Co-Array Fortran
  - Part of Stanford Fortran (subset of features)
  - CAF 2.0 from Rice: http://caf.rice.edu
- Chapel from Cray (own base language better than Java)
  - http://chapel.cray.com (open source)
- X10 from IBM also at Rice (Java, Scala,...)
- Coming soon…. PGAS for Python, aka PyGAS
Summary

• UPC designed to be consistent with C
  – Some low level details, such as memory layout are exposed
  – Ability to use pointers and arrays interchangeably
• Designed for high performance
  – Memory consistency explicit
  – Small implementation
• Berkeley compiler (used for next homework)
  http://upc.lbl.gov
• Language specification and other documents
  http://upc.gwu.edu
• UC Berkeley CS267 on UPC
  – https://people.eecs.berkeley.edu/~demmel/cs267_Spr11