Lecture 15

Parallel Programming Languages
Announcements

• Wednesday’s office hour is cancelled
Today’s lecture

• Revisiting processor topologies in MPI
• Parallel programming languages
  – Data parallelism
  – Global address space language study: UPC
MPI Topologies

- Convenient to organize processors into a topology
- Two ways in MPI
  - Create new communicators by splitting
  - Topology abstractions

```
p(0,0)      p(0,1)    p(0,2)
p(1,0)      p(1,1)     p(1,2) p(2,0)      p(2,1)     p(2,2)
```
Using Topologies in Cannon’s Algorithm

• Enumerate ranks using ordered pairs
• Circular shift
• See $\text{PUB/examples/Cannon\_new}$
Using Comm_split

MPI_Comm colComm;
int myRow = myid % √P;
MPI_Comm_split(MPI_COMM_WORLD, myCol, myid, &colComm);

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
Ring shift with Comm_split

MPI_Comm_rank(colComm,&myidRing);
MPI_Comm_size(colComm,&nodesRing);

int I = mycol, X = …, XC;

int next = (myidRng + 1 ) % nodesRing;
MPI_Send(&X,1,MPI_INT,next,0, colComm);
MPI_Recv(&XC,1,MPI_INT,
          MPI_ANY_SOURCE,
          0, colComm, &status);
Skewing: setting up the communication calls

```c
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, wrap, 1, &gridComm);
MPI_Comm_rank(gridComm,myGridRank);
MPI_Cart_coords (gridComm,myGridRank,coords); // int coords[2];
MPI_Cart_shift(gridComm,0,-coords[1],&srcR,&destR);
MPI_Cart_shift(gridComm,1,-coords[0],&srcC,&destC);
MPI_Sendrecv_replace(local_A->entries,n_bar*n_bar,grid.local_matrix_mpi_t,
                   destR, SK_ROW, srcR, SK_ROW, gridComm, &st1);
MPI_Sendrecv_replace(local_B->entries, n_bar*n_bar, grid.local_matrix_mpi_t,
                   destC, SK_COL, srcC, SK_COL, gridComm, &st1);
```
Parallel programming languages
Motivation for parallel programming languages

- Low levels models are difficult to use
- Manage too many low level details
  - Communication
  - Synchronization
  - Data partitioning
- We do need to think about new things, or think a bit differently
  - The distinction between remote, collective, and local data
  - Losses due to processor idleness can be dramatic
- Today we’ll look at two approaches to parallel programming which eliminates some concerns, simplifying the programming process
Data parallel programming

- Similar to SIMD parallelism
- Virtual processors execute the same operations on multiple data
- Implement the abstraction on MIMD architectures using SPMD parallelism
The data parallel model

- A parallel data structure, e.g. an array, list, sequence
- Apply an operation uniformly over all processors in a single step
- Assign each array element to a virtual processor
- Implicit barrier synchronization between each step
- Program executes as if in a global space
Practical data parallel languages

- APL (1962)
- Matlab
- Fortran 90, 95
- HPF (High Performance Fortran) - 1994
How do we express parallelism?

• Operations on whole arrays
• Forall, a parallel for loop
• FORALL ( triplet, triplet,… ) assignment statement

\[
\text{forall } (i=0:n-1) \; x[i] = (i\times2.0/n)-1.0
\]

\[
\text{forall } (i=0:n-1, \; j = 0:m-1) \; X[i, \; j] = 1.0/(i+j)
\]

• The head of the loop defines an index domain
• We think of each member of the index domain as defining a virtual processor
  – In the first example, we have an index domain of 0 to n-1, with n-1 virtual processors
  – In the second example we have an index domain of n \times m processors
Other uses of forall loops

• Indirect indexing
  
  ```c
  forall ( i = 0:n-1) D[i]= C[indx[i] ]
  ```

• Optional mask or guard
  
  ```c
  forall (i=1:n, j:1:n, i == j) X[i,j] = 0 // Guard
  ```

• A parallel loop nested inside a serial loop
  
  ```c
  for k = 1:n
  ```

• An illegal forall loop
  
  ```c
  forall (i=1:n, j=1:n, k=1:n)
  ```
Forall loop evaluation

- Evaluate entire RHS for all index values (in any order) and assign to a temporary
- Perform all assignments (in any order) using the temporary
- No more than one value for each element on the left hand side
- The following are equivalent

\[
\text{forall ( } i = 1:n \text{) } \text{unew}[i] = \frac{(u[i-1] + u[i+1])}{2.0}
\]

\[
\text{forall ( } i = 1:n \text{) } \text{tmp}[i] = \frac{(u[i-1] + u[i+1])}{2.0}
\]

\[
\text{forall ( } i = 1:n \text{) } \text{unew}[i] = \text{tmp}[i]
\]
Array Operations

Parallel Assignment, equivalent to a forall

```plaintext
double A[N,N,N], Z[N,N], D[N], C[N], T[5]
A = 0 // scalar extension,
     // all elements set to 0
forall(i=0:N-1,j=0:N-1,k=0:N-1) A[i] = 0
Z = 3.7    //forall(i=0:N-1,j=0:N-1) Z[i] = 3.7

D = C       // array copy
T = [1 2 3 4 5]     // An array literal
```

Binary array operators operate pointwise on *conforming* arrays

- same size and shape
- The arrays could be multidimensional
Extension to array operations

• Scalars can be combined with arrays

• There are also specialized intrinsics

\[
\begin{align*}
T & = [1 \ 4 \ 9 \ 16 \ 25] \\
U & = 3 + T \quad // \quad 4 \ 7 \ 12 \ 19 \ 28 \\
Z & = \sqrt{T} \quad // \quad \text{Built in intrinsic extended to array} \\
& \quad // \quad 1 \ 2 \ 3 \ 4 \ 5 \\
Y & = \max(T,10) \quad // \quad 10 \ 10 \ 10 \ 16 \ 25
\end{align*}
\]
Array Sections

- Portion of an array defined by a triplet in each dimension
- May appear wherever an array is used

- $B[:, 1]$ ! first column
- $B[j,:]$ ! jth row
Data Motion

CSHIFT( array, dim, shift)                      ! cyclic shift in one dimension
TRANSPOSE( matrix )                            ! matrix transpose

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
\end{array}
\rightarrow
\begin{array}{cccc}
4 & 1 & 2 & 3 \\
8 & 5 & 6 & 7 \\
\end{array}
\]
Motivating application: the N-body problem

• A classical problem
• Compute trajectories of a system of N bodies moving under mutual influence
• The bodies can be molecules, planets, stars, charged particles…
Trajectories

- Particles move continuously through space and time.
- On a computer we represent continuous values using a discrete approximation.
- Evaluate force field at discrete points in time, called timesteps
  \[ \Delta t, 2\Delta t, 3\Delta t, \ldots \]
  - \( \Delta t \) is called the time step (a parameter).
- “Push” the bodies according to Newton’s third law
  \[ F = ma = m \frac{du}{dt} \]
Solving the N body problem

while (current time < end time)
    forall bodies i ∈ 1:N
        compute force $F_i$ induced by all bodies $j \in 1:N$
        \[
        F_i = \sum_i F_{ij}
        \]
        update position $x_i$ by $F_i \Delta t$
    current time += $\Delta t$
end
Computing the force

- The running time of the computation is dominated by the force computation, so we ignore the rest.
- The simplest approach is to use the direct method, with a running time of $O(N^2)$

$$\text{Force on particle } i = \sum_{j=0}^{N-1} F(x_i, x_j)$$

- $F(\ )$ is the force law.

- One example is the gravitational force law

$$G \frac{m_i m_j}{r_{ij}^2} \text{ where } r_{ij} = \text{distance}(x_i, x_j)$$

$G$ is the gravitational constant.
A simple parallel algorithm

• Each processor is assigned N/P particles
• Processors circulate a copy of the particles in a processor ring configuration
• After P-1 steps every processor has seen every particle
Ring algorithm

Array mine[ ] contains position and mass of particles

Copy mine[ ] into array incoming[ ]

Repeat P times

Compute forces from incoming against mine

Transmit incoming particles to processor (myRank+1) MOD p

Receive incoming particles from processor (myRank-1+p) MOD p

End Repeat
Data parallel variant of the particle computation

• The particles are described by
  • the position and mass array \(\text{xyzm}[\ ]\)
  • shifted in copy \(\text{xyzmC}\)
  • Let the force law be a given function \(F(\text{xyzm},\text{xyzmC})\)

\[
\text{xyzmC} = \text{xyzm} \\
\text{do while} \ (t < t\_end) \\
\quad \text{for} \ i = 1 : n \\
\quad \quad \text{force} = \text{force} + F(\text{xyzm},\text{xyzmC}) \\
\quad \quad \text{xyzmC} = \text{CSHIFT}(\text{xyzmC},1) \ldots. \\
\quad \text{end for} \\
\text{end do}
\]
Reduction Operators

Reduce an array to a scalar under an associative binary operation

- sum, product
- minval, maxval

**do while** (maxdiff < epsilon)

unew[1:N] = (uold[0:N-1] + uold[2:N+1]) / 2.0

diff = unew – uold

absdiff = abs(diff)

maxdiff = maxval(absdiff)

**enddo**
Data Distribution

• Three kinds of distributions
  – Block
  – Cyclic
  – Block_cyclic

• Can we use one of these to express the others?
Layouts on Processor Grids

- [Block, *]
- [*, Block]
- [Block, Block]
- [Cyclic, *]
- [Cyclic, Cyclic]
- [Cyclic, Block]
Block cyclic

[Cyclic, *]

[Cyclic, Cyclic]

[Cyclic, *]

[Cyclic(2), *]

[Cyclic(2), Cyclic]

[Cyclic(2), Block]
Implicit Communication

Array section copies may induce communication

\[
A(1:7) = B(2:8)  \quad ! \text{A and B are distributed BLOCKwise}
\]
Subtleties of implicit communication

• Communication may occur if LHS and RHS have different layouts

  • Consider $A[1:7] = C[2:8]$ where $A$ is BLOCK distributed and $C$ is CYLIC

• Abstraction obscures performance penalty
Global Communication

\[
X = X[n:1:-1] \quad \text{! permutation (reverse)}
\]

\[
B = A[\text{Indx}[:]] \quad \text{! “gather”}
\]

\[
C[\text{Indx}[:]] = B \quad \text{! C = “scatter:”}
\]

! no duplicates on left!
Global Address Space Languages
Uniform Parallel C

• Partitioned Global Address Space Language
  – Shared memory model
  – Thread level parallelism
  – Global vs local data
  – Single sided communication
  – Performance tuning:
    data layout and message-like communication

• Commercial compilers as well as open source
• An extension of ANSI C
• Some materials come from the UPC community
  tutorials: Kathy Yelick and Tarek El-Ghazawi
UPC Example

- Monte Carlo algorithm: estimate $\pi$ by throwing darts at a unit square
- Randomly throw darts at $x,y$ positions
- Calculate % that fall in circle quadrant $= \pi/4$
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4 \times \text{ratio}$

\[
\frac{1}{4} \pi r^2 = \frac{\pi}{4}
\]
• Each thread independently estimates Pi

```cpp
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);
    }
}
```
Pi in UPC: Shared Array Version

• Each thread updates a separate counter in shared array
• One thread computes total sum

```c
shared int all_hits [THREADS];
main(int argc, char **argv) {
  ...
  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 estimate: %f.", 4.0*hits/trials);
  }
}
```

all_hits is shared by all processors
update element with local affinity
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

```plaintext
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

2D array logically blocked by columns

z is not
Pi in UPC: Traditional Shared Memory

```
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;  // create a lock
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    shared int hits;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);     // accumulate hits
    for (i=0; i < my_trials; i++)
        my_hits += hit();    // locally
    upc_lock(hit_lock);
    hits += my_hits;        // accumulate
    upc_unlock(hit_lock);    // across threads
    upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*hits/trials);
```
Pi in UPC: Data Parallel Style

• The previous version is not scalable:
  – On a large # of threads, the locked region will be a bottleneck

• Use a reduction for better scalability

```c
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ... 
    for (i=0; i < my_trials; i++)
        my_hits += hit(); 
    my_hits = bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*my_hits/trials);
}
```
# include <upc_relaxed.h>
const int N = 100*THREADS;
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
            sum[i]=v1[i]+v2[i];
    }

• Issues
  • How to layout data (here it is cyclic)
  • Which processor computes (here it is “owner computes”)
Work Sharing with upc_forall()

- Loop over all; work on those owned by this process
  
  `upc_forall(init; test; loop; affinity)
  statement;

  `upc_forall(i=0; i<N; i++; i)
       sum[i]=v1[i]+v2[i];`

- The iterations are independent
  - Undefined if there are dependencies across threads

- Affinity expression indicates which iterations to run on each thread
  - Integer: `affinity%THREADS` is `MYTHREAD`
  - Pointer: `upc_threadof(affinity)` is `MYTHREAD`

- Syntactic sugar for loop on previous slide
  - Some compilers may do better than this, e.g.,
    
    `for(i=MYTHREAD; i<N; i+=THREADS)`
  - Rather than having all threads iterate N times
Vector Addition with upc_forall

• The `vadd` example can be rewritten as follows

• The code would be correct but slow if the affinity expression were `i+1` rather than `i`

• Cyclic data distribution may perform poorly on some machines

```c
#define N 100*THREADES
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    upc_forall(i=0; i<N; i++; i )
        sum[i]=v1[i]+v2[i];
}
```
Blocked Layouts in UPC

- The cyclic layout is typically stored in one of two ways
  - Distributed memory: each processor has a chunk of memory
    - Thread 0 would have: 0, THREADS, THREADS*2,... in a chunk
  - Shared memory machine: each thread has a logical chunk
    - Shared memory would have: 0, 1, 2,... THREADS, THREADS+1,...
- What performance problem is there with the latter?
- What is this code was instead doing nearest neighbor averaging?
- Vector addition example can be rewritten as follows

```c
#define N 100*THREADS
shared int [*] v1[N], v2[N], sum[N];
void main() {
    int i;
    upc_forall(i=0; i<N; i++; &a[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 defined in terms of:
  - block size, a compile-time constant
  - and THREADS
- Element $i$ has affinity with thread
  $\frac{i}{\text{block\_size}} \mod \text{THREADS}$
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping
2D Array Layouts in UPC

- Array $a_1$ has a row layout, array $a_2$ has a block row layout
  
  ```cpp
  shared [m] int a1 [n][m];
  shared [k*m] int a2 [n][m];
  ```

- If $(k + m) \div \text{THREADS} = = 0$ then $a_3$ has a row layout
  
  ```cpp
  shared int a3 [n][m+k];
  ```

- To get BLOCK CYCILC layouts, one needs to add dimensions.
  
- Assume $r*c = \text{THREADS}$;
  
  ```cpp
  shared [b1][b2] int a5 [m][n][r][c][b1][b2];
  ```

- or equivalently
  
  ```cpp
  shared [b1*b2] int a5 [m][n][r][c][b1][b2];
  ```
UPC Matrix Vector Multiplication Code

- Matrix-vector multiplication with matrix stored by rows
- (Contrived example: problems size is PxP)

```c
shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void) {
    int i, j , l;
    upc_forall( i = 0 ; i < THREADS ; i++ ; i) {
        c[i] = 0;
        for ( l= 0 ; l < THREADS ; l++)
            c[i] += a[i][l]*b[l];
    }
}
```
UPC Matrix Multiplication Code

```c
#include <upc_relaxed.h>
const int N=4, P=4, M=4;
shared [N*P /THREADS] int a[N][P], c[N][M]; // a and c are row-wise blocked shared matrices
shared[M/THREADS] int b[P][M]; //column-wise blocking
void main (void) {
    int i, j, l; // private variables
    upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ;j++) {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
• 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>PP (<strong>p1</strong>)</td>
<td>PS (<strong>p3</strong>)</td>
</tr>
<tr>
<td>Shared</td>
<td>SP (<strong>p2</strong>)</td>
<td>SS (<strong>p4</strong>)</td>
</tr>
</tbody>
</table>

#### Code Examples

```c
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 private is not recommended**
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.
GASNet: Portability and High-Performance

8-byte Roundtrip Latency

- MPI ping-pong
- GASNet put+sync

GASNet better for latency across machines

11/7/06 Scott B. Baden/CSE 160/Fall 2006
GASNet: Portability and High-Performance

GASNet excels at mid-range sizes: important for overlap
Application study: NAS FT

- Performance of Exchange (Alltoall) is critical
  - 1D FFTs in each dimension, 3 phases
  - Transpose after first 2 for locality
  - Bisection bandwidth-limited
    - Problem as #procs grows
- Three approaches:
  - Exchange:
    - wait for 2nd dim FFTs to finish, send 1 message per processor pair
  - Slab:
    - wait for chunk of rows destined for 1 proc, send when ready
  - Pencil:
    - send each row as it completes
NAS FT Variants Performance

- Slab is always best for MPI; small message cost too high
- Pencil is always best for UPC; more overlap
Other features

- Storage allocation
- Single sided communication upc_mem{put,get}
- upc.lbl.gov