Lecture 14

Shared memory programming
MPI Topologies
Announcements

• Midterm return today
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads
  - Kernel scheduler multiplexes threads
Barrier synchronization

- Ensures that no process can move on until all have arrived
- Don’t overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- Don’t read locations updated by other processes in the previous iteration until they have been produced (true dependence)
- A barrier can be built with locks
Building a linear time barrier with locks

Mutex arrival=UNLOCKED, departure=LOCKED;
int count=0;

void Barrier( )
    arrival.lock( ); // atomically count the
    count++; // waiting threads
    if (count < n$proc) arrival.unlock( );
    else departure.unlock( ); // last processor
        // enables all to go
    departure.lock( );
    count--; // atomically decrement
    if (count > 0) departure.unlock( );
    else arrival.unlock( ); // last processor resets state
Motivating application

- Jacobi’s method for solving Poisson’s equation in two dimensions

```c
for j = 1 : N
    for i = 1 : M
        u\text{new}[i,j] = \frac{(u[i-1,j] + u[i+1,j] + u[l,j-1] + u[l,j+1])}{4};
```

![Diagram of a grid with a central point and surrounding points connected with lines]
The code

1. procedure Solve (sharedArray2D<float> A) // A is an (n + 2)-by-(n + 2) array
2. begin
3. int done = FALSE;
4. float diff;
5. while (!done) do // outermost sweep loop
6.     diff = 0;
7.     for i ← 1 to n do // sweep over interior points of grid
8.         for j ← 1 to n do
10.                A[i, j+1] + A[i+1, j]);
11.             diff += abs(A[i, j] – A'[i, j]);
12.         end for
13.     end for
14.     A[: ,:] = A'[: ,:] // Set old solution = new solution
15.     if (diff/(n^2) < TOL) done = TRUE;
16. end while
17. end procedure

• Interior n x n points updated in each sweep
• Compute error, taking differences against old solution
• Update old solution from new solution
• Continue sweeping until solution has converged
Exposing the parallelism

1. procedure Solve (SharedArray2D<float>A)  // A is an (n + 2)-by-(n + 2) array
2. begin
3.   int done = FALSE;
4.   while (!done) do
5.     float diff = 0;
6.     forall i ← 1 to n do // sweep over interior points of grid
7.       forall j ← 1 to n do
10.        diff += abs(A[i,j] - A'[i,j]);
11.     end for
12.   end for
13.   diff = REDUCE_SUM(diff)
14.   A[:,] = A'[:,]
15.   if (diff/(n^2) < TOL) done = TRUE;
16. end while
17. end procedure
Shared memory parallel code

1a. Lock diff_lock;   // declaration of lock to enforce mutual exclusion
1b. Barrier bar1;    // barrier declaration for global synchronization between sweeps
2. Array2D<float> A; // Shared array
2b. float diff;
3. main()
4. begin
5. read(n); read(nprocs); bar1.init(nprocs);
6. A ← new Array2D<float>(n+2,n+2)
7. FORK (nprocs−1, Solve, A);
8. Solve(A);          /*main process becomes a worker too*/
9. JOIN (nprocs−1);   /*wait for all child processes created to terminate*/
10. end main

• Variables declared out of the scope of any function are global to all processes, all others are local to the process
Threaded code

\[
\text{mymin} = 1 + (TID \times n/NT), \quad \text{mymax} = \text{mymin} + n/NT -1;
\]

\textbf{while (done) do}

\begin{align*}
\text{mydiff} &= \text{diff} = 0; \\
\textit{omp} & \textbf{barrier} \\
\textbf{for} i = \text{mymin} \textbf{ to } \text{mymax} \textbf{ do} \\
\textbf{for} j = 1 \textbf{ to } n \textbf{ do} \\
& A'[l,j] = \ldots \\
& \text{mydiff} += \ldots
\end{align*}

\textit{omp critical}

\begin{align*}
\{ \text{diff} += \text{mydiff} \}
\end{align*}

\textit{omp barrier}

\textbf{if (diff / (n*n) < Tolerance) done = TRUE;}

\textit{omp barrier}

\textbf{end while}

- Don’t read locations updated by other processes in the previous iteration until they have been produced (true dependence)
- Don’t overwrite values used by other processes in the current iteration until they have been consumed (anti-dependence)
Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally when the loop is over
- What does this remind us of?

```c
#pragma omp parallel reduction(+:sum)
for (int i=i0; i< i1; i++)
    sum += x[i];
```
Coding with pthreads

#include <pthread.h>
void *PrintHello(void *threadid){
cout << "Hello World from " << (int)threadid << endl; pthread_exit(NULL); }

int main(int argc, char *argv[]){
    int NT; cin >> NT; pthread_t th[NT];
    for(int t=0;t<NT;t++)
        assert(!pthread_create(&th[t], NULL, PrintHello, (void *)t));

    pthread_exit(NULL);
Computing the dot product

- Dot product: dotprod_mutex.c in
  www.llnl.gov/computing/tutorials/pthreads/samples

```c
pthread_t thrd[NT];
double *x = (double*) malloc (NT*N*sizeof(double));
for(i=0;i<NT;i++)
    pthread_create( &thrd[i], &attr, dotprod, (void *)i);
for(i=0;i<NT;i++) {
    pthread_join( thrd[i], (void **)&status);
```

```c
pthread_create( &thrd[i], &attr, dotprod, (void *)i);
for(i=0;i<NT;i++) {
    pthread_join( thrd[i], (void **)&status);
```
The computation

```c
void *dotprod(void *arg){
    int TID = (int)arg;
    int i0 = TID*N, i1 = i0 + N;
    double mysum = 0;
    for ( i=i0; i<i1; i++)
        mysum += x[i] * y[i];
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit((void*) 0);
}
```
Measuring shared memory system performance

• Dot product: 2 mems / 2 flops
• DOT_PROD = SUM(x[:,]*y[:])
• On valkyrie, 1M array per processor
  – NT=1: 44.6 MFlops
  – NT=2: 68.0 MFlops
• On a dual 1.8GHz Opteron: 250 and 500 MFlops
• SDSC Datastar, 1.5GHz Power4: 289&586 MFlops
Reducing memory impact

- Insert operations that use registers only

```c
for (int i=0; i < N; i++){
    result += x[i]*y[i];
    for (int j=0; j < K; j++) result *= c;
}
```

- On valkyrie, 1M array per processor

<table>
<thead>
<tr>
<th>NT</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.6</td>
<td>121</td>
<td>71</td>
<td>92</td>
<td>113</td>
<td>150</td>
<td>172</td>
<td>165</td>
</tr>
<tr>
<td>2</td>
<td>68.8</td>
<td>157</td>
<td>114</td>
<td>151</td>
<td>198</td>
<td>270</td>
<td>319</td>
<td>322</td>
</tr>
</tbody>
</table>

11/3/06 Scott B. Baden/CSE 160/Fall 2006
Workload decomposition

• Static assignment: BLOCK decomposition
• Dynamic assignment: self scheduling
  – get a row index, work on the row, get a new row, repeat
• Static assignment into rows reduces concurrency from $n$ to $p$, but reduces synchronization overhead
• Unlike message passing, workload assignment is specified by the subsets of the global loop bounds
  
  ```c
  #pragma omp parallel private(i) shared(n)
  {
    #pragma omp for
    for(i=0; i < n; i++)
      work(i);
  }
  ```
How do does OpenMP distribute the loops?

• With static decomposition, the process is straightforward: each process gets a unique range of indices based on its thread ID
• But with irregular problems, or when processor performance is not predictable, we can’t use static decomposition
• OpenMP provides a dynamic option
• Relies on processor self-scheduling
How does processor self-scheduling work?

• Processors “schedule” themselves
• Sample a shared counter or work queue to obtain work
• Adjust work granularity (“chunk size”) to trade off the overhead of sampling the queue against increased load imbalance
• Also used with work queues

![Graph showing running time, load imbalance, and high overheads as functions of increasing granularity.](image)
Details of self scheduling

$omp parallel
while (!done) do
    mydiff = diff = 0;
$omp barrier
    for i = 1 to n do
        while (getNextChunk(&mymin,&mymax )) do
            for j = mymin to mymax do
                A'[i,j] = .... ;
            end for
        end do
    end for
    ReduceSum(diff,mydiff);
$omp barrier
    if (diff/(n*n) < TOL) done = TRUE;
$omp barrier
end do
More details of self scheduling

SelfScheduler S(n,P,CSize);

Boolean getNextChunk(int * mymin, int * mymax ){

    #omp critical
    {
        k = S.counter += S.ChunkSize;
    }
    mymin = k;
    mymax = k + S.chunkSize;
}
Hybrid MPI + thread programming

- Take advantage of faster shared memory on-node
- Higher complexity than “flat MPI”
- KeLP2:
  www.cse.ucsd.edu/groups/hpcl/scg/Research/MT.html
Multi-tier model

• Programs have 3 levels of control
  – **Collective** level operations performed on all nodes
  – **Node** level operations performed on one node
  – Processor level operations performed on a single CPU
Managing locality

- Well performed programs use tricks from message passing
- Alobal address space is helpful
- Single sided communication
  - The best of both worlds
  - put and get
End of threads programming
MPI_PROC_NULL

• We may specify a rank of MPI_PROC_NULL to handle edge cases
• Consider ghost cell communication

if (myrank == 0) up = MPI_PROC_NULL;
else up = myrank - 1;
if (myrank == p-1) down = MPI_PROC_NULL
else down = myrank+1;

MPI_Sendrecv(B(1,1),n, MPI_FLOAT, up, tagUP, 
A(1,0),n, MPI_FLOAT, up, tagUP, comm, 
status, ierr)

MPI_Sendrecv(B(1,m),n, MPI_FLOAT, down, tagDN, 
A(1,m+1),n, MPI_FLOAT, down, tagDN, comm, 
status, ierr)
MPI Topologies

• Convenient to organize processors into a topology
• Two ways in MPI
  – Create new communicators by splitting
  – Topologies

\[
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2)
\end{array}
\]
Using Topologies in Cannon’s Algorithm

- Enumerate ranks using ordered pairs
- Circular shift
- See $PUB/examples/Cannon_new
Using Comm_split

MPI_Comm colComm;
int myRow = myid % \sqrt{P};
MPI_Comm_split(MPI_COMM_WORLD, myCol, myid, &colComm);

\begin{tabular}{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{tabular}
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);
Using Cartesian Topologies

```c
MPI_Cart_create(COMM_WORLD, 2, dims, wrap, 1, &gridComm);
MPI_Comm_rank(gridComm, myGridRank);
MPI_Cart_coords(gridComm, myGridRank, coords);  // int coords[2];
```

```
<table>
<thead>
<tr>
<th></th>
<th>p(0,0)</th>
<th>p(0,1)</th>
<th>p(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1,0)</td>
<td>p(1,1)</td>
<td>p(1,2)</td>
<td></td>
</tr>
<tr>
<td>p(2,0)</td>
<td>p(2,1)</td>
<td>p(2,2)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>B(0,0)</th>
<th>B(1,1)</th>
<th>B(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
<td></td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
<td></td>
</tr>
</tbody>
</table>
```
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, &st2);
```

```
<table>
<thead>
<tr>
<th></th>
<th>p(0,0)</th>
<th>p(0,1)</th>
<th>p(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(0,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(1,0)</td>
<td>p(1,1)</td>
<td>p(1,2)</td>
<td></td>
</tr>
<tr>
<td>p(2,0)</td>
<td>p(2,1)</td>
<td>p(2,2)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>B(0,0)</th>
<th>B(1,1)</th>
<th>B(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(0,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
<td></td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
<td></td>
</tr>
</tbody>
</table>
```