CSE 160
Lecture 17

Hypercube algorithm
Gather/Scatter
Announcements

• Quiz #3 return
• Some tips about synchronization bugs
### Question 2

```c
(1) void sweep(int TID, int myMin, int myMax, double ε, double& err){
(2)     for (int s = 0; s < 100; s++) {
(3)         double localErr = 0;
(4)         for (int i = myMin; i < myMax; i++){
(5)             unew[i] = (u[i-1] + u[i+1])/2.0;
(6)             double δ = fabs(u[i] - unew[i]);
(7)             localErr += δ * δ ;
(8)         }
(9a)     BEGIN CRITICAL SECTION:
(9b)     err += localErr;
(9c)     END CRITICAL SECTION
(10)    if ((s > 0) && ( err < ε))
(11)        break;
(12)    if (!TID){double *t = u; u = unew; unew = t;} // Swap u ↔ unew
(13)    err = 0;
(14) } // End of s loop
(15) }
```
void sweep(int TID, int myMin, int myMax, double ε, double& err) {
    ... 
    BEGIN CRITICAL SECTION:
    err += localErr;
    END CRITICAL SECTION
    BEGIN CRITICAL SECTION:
    if ((s > 0) && (err < ε))
        break;
    END CRITICAL SECTION
    if (!TID) {
        double *t = u;
        u = unew;
        unew = t; // Swap u ← unew
        BEGIN CRITICAL SECTION:
        err = 0;
        END CRITICAL SECTION
    }
} // End of s loop
Happens Before Relationship?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) for (int s = 0; s &lt; 100; s++) {</td>
<td>(1) for (int s = 0; s &lt; 100; s++) {</td>
</tr>
<tr>
<td>(9a) BEGİN CRİTİCAL SECTİON:</td>
<td>(9a) BEGİN CRİTİCAL SECTİON:</td>
</tr>
<tr>
<td>(9b) err += localErr;</td>
<td>(9b) err += localErr;</td>
</tr>
<tr>
<td>(9c) END CRİTİCAL SECTİON</td>
<td>(9c) END CRİTİCAL SECTİON</td>
</tr>
<tr>
<td>(10) if ((s &gt; 0) &amp;&amp; (err &lt; \varepsilon))</td>
<td>(10) if ((s &gt; 0) &amp;&amp; (err &lt; \varepsilon))</td>
</tr>
<tr>
<td>(11) break;</td>
<td>(11) break;</td>
</tr>
<tr>
<td>(12) if (!TID) {/<em>u&lt;--&gt;unew</em>/}</td>
<td>(12) if (!TID) {/<em>u&lt;--&gt;unew</em>/}</td>
</tr>
<tr>
<td>(13) err = 0;</td>
<td>(13) err = 0;</td>
</tr>
<tr>
<td>(14) } // end of s loop</td>
<td>(14) } // end of s loop</td>
</tr>
</tbody>
</table>

(9b) ↔ (10)
(10) ↔ (13)
(12) i-th iteration ↔ i+1-th iteration
Minimize Synchronization

(1) void sweep(int TID, int myMin, int myMax, double $\epsilon$, double& err){

(2-8) ... 

(9a) BEGIN CRITICAL SECTION:
(9b) err += localErr;
(9c) END CRITICAL SECTION

BARRIER()

BEGIN CRITICAL SECTION:

(10) if ((s > 0) && (err < $\epsilon$))
(11) break;

END CRITICAL SECTION

BARRIER()

(12) if (!TID){double *t = u; u = unew; unew = t;} // Swap u $\leftrightarrow$ unew

BEGIN CRITICAL SECTION:

(13) err = 0;

END CRITICAL SECTION

BARRIER()

(14) } // End of s loop

(15) }
Some tips about Concurrent bugs

//NT is a global variable
//The function intends to print the number of threads
(1)ThreadFunc(int TID)
(2)    NT++;
(3)    if(TID==0)
(4)        cout<<NT<<endl;
(5)    }
(6) }

Brute force: try locks and barriers at each global access?

How to do it systematically?
Thread Synchronization

• Exclusion
  ‣ Critical sections
  ‣ Order not guaranteed
  ‣ Mechanisms (mutex, atomic variables, etc.)

• Synchronization
  ‣ Happen-before relationship
  ‣ Mechanisms (barrier, join, condition variable, spinning loop, etc.)

• Minimal synchronization
Exclusion

- Find all the critical section locations
  - shared variables accessed by different threads
  - at least one access is write

```cpp
//NT is a global vairable
//The function intends to print the //
number of threads
(1)ThreadFunc(int TID){
(2)  NT++;
(3)  if(TID==0){
(4)    cout<<NT<<endl;
(5)  }
(6)}
(10)
```
Check for thread synchronization

- Identify happens-before relationship requirements

```c++
ThreadFunc(int TID) {
    lock();
    NT++;
    unlock();
    if (TID == 0) {
        lock();
        cout << NT << endl;
        unlock();
    }
}
```

```
Thread 1
(1) ThreadFunc(int TID) {
(2)   lock();
(3)   NT++;
(4)   unlock();
(5)   if (TID == 0) {
(6)     lock();
(7)     cout << NT << endl;
(8)     unlock();
(9) } }
(10})
```

```
Thread 2
(1) ThreadFunc(int TID) {
(2)   lock();
(3)   NT++;
(4)   unlock();
(5)   if (TID == 0) {
(6)     lock();
(7)     cout << NT << endl;
(8)     unlock();
(9) } }
(10})
```

Happens-before
Minimize Synchronization

(1) ThreadFunc(int TID) {
(2)   lock();
(3)   NT++;
(4)   unlock();
(5)   barrier();
(6)   if(TID==0) {
(7)       lock();
(8)       cout<<NT<<endl;
(9)       unlock();
(10)  }
(11) }

Three Steps

• Critical section identification
• Happens-before relation
• Minimal synchronization
Today’s lecture

• More Collectives
  ‣ Inside MPI
  ‣ Hypercubes and spanning trees
  ‣ Gather/Scatter

• Parallel Print Function
Collective communication

• Basic collectives seen so far
  ▶ Broadcast: distribute data from a designated root process to all the others
  ▶ Reduce: combine data from all processes returning the result to the root process
  ▶ How do we implement them?

• Other Useful collectives
  ▶ Scatter/gather
  ▶ All to all
  ▶ Allgather

• Diverse applications
  ▶ Printing distributed data: “parallel print”
  ▶ Sorting
  ▶ Fast Fourier Transform
Underlying assumptions

• Fast interconnect structure
  ‣ All nodes are equidistant
  ‣ Single-ported, bidirectional links

• Communication time is $\alpha + \beta n$ in the absence of contention
  ‣ Determined by bandwidth $\beta^{-1}$ for long messages
  ‣ Dominated by latency $\alpha$ for short messages
Inside MPI-CH

- Tree like algorithm to broadcast the message to blocks of processes, and a linear algorithm to broadcast the message within each block
- Block size may be configured at installation time
- If there is hardware support (e.g. Blue Gene), then it is given responsibility to carry out the broadcast
- Polyalgorithms apply different algorithms to different cases, i.e. long vs. short messages, different machine configurations
- We’ll use hypercube algorithms to simplify the special cases when $P=2^k$, $k$ an integer
Recapping the Parallel Implementation of the Trapezoidal Rule

- Decompose the integration interval into sub-intervals
- Each core computes the integral on its subinterval
- All combine their local integrals into a global one
- Use a collective routine to combine the local values

\[ \int_{a}^{b} f(x) \, dx \]
Collective communication in MPI

- Collective operations are called by all processes within a communicator
- Broadcast: distribute data from a designated “root” process to all the others
  \[
  \text{MPI\_Bcast(in, count, type, root, comm)}
  \]
- Reduce: combine data from all processes and return to a designated root process
  \[
  \text{MPI\_Reduce(in, out, count, type, op, root, comm)}
  \]
- Allreduce: all processes get reduction: \text{Reduce + Bcast}
\textbf{Final version}

\begin{verbatim}
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
    integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce( &integral, &total, 1,
    MPI_FLOAT, MPI_SUM, WORLD)
\end{verbatim}
Broadcast

- The root process transmits of $m$ pieces of data to all the $p-1$ other processors
- With the linear ring algorithm this processor performs $p-1$ sends of length $m$
  - Cost is $(p-1)(\alpha + \beta m)$
- Another approach is to use the hypercube algorithm, which has a logarithmic running time
Today’s lecture

• More Collectives
  ‣ Inside MPI
  ‣ Hypercubes and spanning trees
  ‣ Gather/Scatter
• Parallel Print Function
Sidebar: what is a hypercube?

• A hypercube is a d-dimensional graph with $2^d$ nodes
• A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
• Each node has d neighbors
Properties of hypercubes

- A hypercube with \( p \) nodes has \( \lg(p) \) dimensions
- *Inductive construction*: we may construct a \( d \)-cube from two \( (d-1) \) dimensional cubes
- *Diameter*: What is the maximum distance between any 2 nodes?
- *Bisection bandwidth*: How many cut edges (mincut)
Bookkeeping

- Label nodes with a binary reflected grey code

- Neighboring labels differ in exactly one bit position
  \[001 = 101 \oplus e_2, \quad e_2 = 100\]
Hypercube broadcast algorithm with p=4

- Processor 0 is the root, sends its data to its hypercube “buddy” on processor 2 (10)
- Proc 0 & 2 send data to respective buddies
Reduction

- We may use the hypercube algorithm to perform reductions as well as broadcasts.
- Another variant of reduction provides all processes with a copy of the reduced result. 
  \texttt{Allreduce()}
- Equivalent to a \texttt{Reduce} + \texttt{Bcast}
- A clever algorithm performs an \texttt{Allreduce} in one phase rather than having perform separate reduce and broadcast phases.
Allreduce

- Can take advantage of duplex connections
Today’s lecture

• More Collectives
  ‣ Inside MPI
  ‣ Hypercubes and spanning trees
  ‣ Gather/Scatter

• Parallel Print Function
Scatter/Gather

P₀  P₁  Pᵢ₋₁

Gather

Scatter

Root
### Scatter

- **Simple linear algorithm**
  - Root processor sends a chunk of data to all others
  - Reasonable for long messages

\[
(p - 1)\alpha + \frac{p - 1}{p} n\beta
\]

- Similar approach taken for Reduce and Gather
- For short messages, we need to reduce the complexity of the latency ($\alpha$) term
Minimum spanning tree algorithm

- Recursive hypercube-like algorithm with $\lceil \log P \rceil$ steps
  - Root sends half its data to process $(\text{root} + p/2) \mod p$
  - Each receiver acts as a root for corresponding half of the processes
  - MST: organize communication along edges of a minimum-spanning tree covering the nodes
- Requires $O(n/2)$ temp buffer space on intermediate nodes
- Running time:
  $$[\lg P] \alpha + \frac{p - 1}{p} n \beta$$
Today’s lecture

• More Collectives
  ‣ Inside MPI
  ‣ Hypercubes and spanning trees
  ‣ Gather/Scatter

• Parallel Print Function
Parallel print function

• Application of Gather
• Problem: how to sort out all the output on the screen
• Many messages say the same thing
  
  Process 0 is alive!
  Process 1 is alive!
  ...
  Process 15 is alive!

• Compare with

  Processes[0–15] are alive!

• Parallel print facility
  http://www.llnl.gov/CASC/ppf
Summary of capabilities

• Compact format list sets of nodes with common output
  
  PPF_Print( MPI_COMM_WORLD, "Hello world" );
  0–3: Hello world

• %N specifier generates process ID information
  
  PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );
  Message from 0–3

• Lists of nodes
  
  PPF_Print( MPI_COMM_WORLD,
             (myrank % 2)
               ? "[%N] Hello from the odd numbered nodes!\n"
               : "[%N] Hello from the even numbered nodes!\n")

  [0,2] Hello from the even numbered nodes!
  [1,3] Hello from the odd numbered nodes!
Practical matters

- Installed in $(PUB)/lib/PPF
- Specify ppf=1 and mpi=1 on the “make” line
  - Defined in arch.gnu-4.7_c++11.generic
  - Each module that uses the facility must
    ```
    #include "ptools_ppf.h"
    ```
- Look in $(PUB)/Examples/MPI/PPF for example programs ppfexample_cpp.C and test_print.c
- Uses MPI_Gather()