Lecture 15

Advanced Collectives and their Applications
Parallel Print Function
Announcements

• Today’s office hours delayed 30 minutes:
  ‣ 4.30 to 5.30pm

• Makeup lecture on Friday during regular section time
Today’s lecture

• Finish up SUMMA
• Parallel Print Function
• Advanced collectives
Continuing with SUMMA
Recapping SUMMA

• Compute the sum of \( n \) outer products
• Each row & column \((k)\) of \( A \) & \( B \) generates a single outer product
  ‣ Column vector \( A[:,k] \) \((n \times 1)\) & a vector \( B[k,:] \) \((1 \times n)\)
  for \( k := 0 \) to \( n-1 \)
  \( C[:,] += A[:,k] \cdot B[k,:] \)
Recapping SUMMA

• Compute the sum of $n$ outer products
• Each row & column $(k)$ of $A$ & $B$ generates a single outer product
  ‣ $A[: , k+1] \cdot B[k+1, :]$
Recapping SUMMA

• Compute the sum of $n$ outer products
• Each row & column ($k$) of $A$ & $B$ generates a single outer product
  › $A[:,n-1] \cdot B[n-1,:]$

\[
\text{for } k := 0 \text{ to } n-1 \\
C[:, :] += A[:,k] \cdot B[k,:]
\]
Parallel algorithm

- Processors organized into rows and columns, process rank an ordered pair
- Processor geometry $P = px \times py$
- Blocked (serial) matrix multiply, panel size $= b << N/\max(px,py)$ for $k := 0$ to $n-1$ by $b$
  
  Owner of $A[:,k:k+b-1]$ Bcasts to ACol  // Along processor rows
  
  Owner of $B[k:k+b-1,:]$ Bcasts BRow  // Along processor columns
  
  $C += \text{Serial Matrix Multiply}(ACol,BRow)$

- Each row and column of processors independently participate in a panel broadcast
- Owner of the panel (Broadcast root) changes with $k$
What is the performance?

for k := 0 to n–1 by b

// Tree broadcast: \(\lg \sqrt{p} (\alpha + \beta bn/\sqrt{p})\)

multicast A[ :, , k:k+b−1 ] along rows
multicast B[ k:k+b−1, : ] along columns

// Built in matrix multiply: \(2(n/\sqrt{p})^2 b\)

\(C += A[:,k:k+b−1] * B[k:k+b−1,:]\)

• Total running time: \(\sim 2n^3/p + (\lg p/(2\sqrt{p})) \beta n^2\)
• How much time is spent communicating?
Back to Communication domains

- Create a communicator for each row and column
- Group the processors by row

\[
\text{key} = \text{myRank} \div \sqrt{P}
\]

\[
\begin{array}{c|cccc}
& X0 & X1 & X2 & X3 \\
\hline
Y0 & P0 & P1 & P2 & P3 \\
& (0,0) & (0,1) & (0,2) & (0,3) \\
\hline
Y1 & P4 & P5 & P6 & P7 \\
& (1,0) & (1,1) & (1,2) & (1,3) \\
\hline
Y2 & P8 & P9 & P10 & P11 \\
& (2,0) & (2,1) & (2,2) & (2,3) \\
\hline
Y3 & P12 & P13 & P14 & P15 \\
& (3,0) & (3,1) & (3,2) & (3,3) \\
\end{array}
\]
Establishing row communicators

```c
MPI_Comm rowComm;
MPI_Comm_split( MPI_COMM_WORLD, myRank / \sqrt{P}, myRank, &rowComm);
MPI_Comm_rank(rowComm,&myRow);
```

- Ranks apply only to the respective communicator
- Ordered according to myRank

<table>
<thead>
<tr>
<th>Y0</th>
<th>X0</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>Y0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y1</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
<td>P7</td>
</tr>
<tr>
<td>Y1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y2</td>
<td>P8</td>
<td>P9</td>
<td>P10</td>
<td>P11</td>
</tr>
<tr>
<td>Y2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Y3</td>
<td>P12</td>
<td>P13</td>
<td>P14</td>
<td>P15</td>
</tr>
<tr>
<td>Y3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Today’s lecture

• Finish up SUMMA
• Parallel Print Function
• Advanced collectives
Parallel print function

- Debugging output can be hard to sort out 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 set notation
  
  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
• Use a special version of the arch file called

    arch.gnu.mpi.ppf
• 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
Word problem
Prefix sum

- The prefix sum (also called a sum-scan) of a sequence of numbers $x_k$ is in turn a sequence of running sums $S_k$ defined as follows
  \[ S_0 = 0, \quad S_k = S_{k-1} + x_k \]
- Thus, scan $(3,1,4,0,2) = (3,4,8,8,10)$

- Design an algorithm for prefix sum on the hypercube
- Use the gray coding to define the layout of values across processors
Prefix sum algorithm

![Diagram of prefix sum algorithm with four steps: initial distribution of values, distribution of sums before second step, distribution of sums before third step, and final distribution of prefix sums.]

[..] Accumulated
(..) outgoing in next step

```
1. procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
2. begin
3. result := my_number;
4. msg := result;
5. for i := 0 to d - 1 do
6. partner := my_id XOR 2^i;
7. send msg to partner;
8. receive number from partner;
9. msg := msg + number;
10. if (partner < my_id) then result := result + number;
11. endfor;
12. end PREFIX_SUMS_HCUBE
```
Today’s lecture

• Finish up SUMMA
• Parallel Print Function
• Advanced collectives
Collective communication

- Diverse applications
  - Fast Fourier Transform
  - Sorting
- Collective operations are called by all processes within a communicator
- 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
- Other Useful collectives
  - Scatter/gather
  - All to all
Scatter/Gather family

$P_0$  $P_1$  $P_{p-1}$

Gather

Scatter

Root
Gather

MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>IN</td>
<td>recvcount</td>
<td>3 elements for any single receive</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
<td>data type of recv buffer elements</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
<td>rank of receiving process</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
Gatherv

MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>IN</td>
<td>recvcounts</td>
<td>integer array</td>
</tr>
<tr>
<td>IN</td>
<td>displs</td>
<td>integer array of displacements</td>
</tr>
<tr>
<td>IN</td>
<td>recvtype</td>
<td>data type of recv buffer elements</td>
</tr>
<tr>
<td>IN</td>
<td>root</td>
<td>rank of receiving process</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>

As if root made n calls:  

MPI_Recv(recvbuf +displs[i]*sizeof(recvtype), recvcounts[i], recvtype, i, ...)

The jth portion of recvbuf begins at recvbuff[ displs[j]]
Setting up Gatherv

- Processes transmit varying amounts of information to the root
- Prior to calling Gatherv(), all processes must tell the root how much data they are sending
- The root generates two arrays for Gatherv()
  - A receive list: how much it will receive from each process
  - A displacement list: where to situate the data
- Worked examples at
  [www.csd.uoc.gr/~hy555/mpi/mpi_complete_reference.pdf](www.csd.uoc.gr/~hy555/mpi/mpi_complete_reference.pdf)
Scatter algorithm

• 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 Gather (and Reduce)
• For short messages, we can 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:

$$[\lg P]_\alpha + \frac{p - 1}{p} n \beta$$
All-to-all Communication
All to all

• Also called *total exchange* or *personalized communication*: a transpose
• Each process sends a different chunk of data to each of the other processes
• Used in sorting and signal processing
Exchange algorithm

• \( n \) elements / processor (\( n \) total elements)
• \( p - 1 \) step algorithm
  ‣ Each processor exchanges \( n/p \) elements with each of the others
  ‣ In step \( i \), process \( k \) exchanges with processes \( k \pm i \)

\[
\text{for } i = 1 \text{ to } p-1 \\
src = (\text{rank} - i + p) \mod p \\
dest = (\text{rank} + i) \mod p \\
\text{sendrecv( from src to dest )}
\]
end for

• Good algorithm for long messages

• Running time:
  \[
  (p - 1)\alpha + \frac{p - 1}{p} n\beta \approx n\beta
  \]
Recursive doubling for short messages

• In each of $\lceil \log p \rceil$ phases all nodes exchange $\frac{1}{2}$ their accumulated data with the others

• Only $P/2$ messages are sent at any one time

  \[ D = 1 \]

  \[ \text{while } (D < p) \]

  Exchange & accumulate data with rank $\otimes D$

  Left shift $D$ by 1

  \[ \text{end while} \]

• Optimal running time for short messages

  \[ [\log P] \alpha + nP\beta \approx [\log P] \alpha \]
Flow of information
Flow of information
Flow of information
Summarizing all to all

• Short messages $[\log P] \alpha$

• Long messages $\frac{p - 1}{p} n \beta$