Lecture 16

Parallel Matrix Multiplication
Announcements

• Assignment #5
  - Message passing on Triton
  - GPU programming on Lincoln

• Calendar
  - No class on Tuesday/Thursday Nov 16th/18th
  - TA Evaluation, Professor and Course Evaluation: Nov 23rd
Today’s lecture

• Some short topics on MPI
• Matrix Multiplication
• Working with Communication Domains
Correctness and fairness

• Iteration 1: 1 sends to 2 & 0, 0 sends to 1, 2 sends to 0 & 1
• 1 begins iteration 2: 1 sends to 2
• 0 sends to 2 (but for iteration 0)
• Problem: irecv in process 2 is receiving data from 1 in iteration 2 (of process 1) while it expects data from process

```
For i = 1 to n
MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
End for
```
Send_Recv

• Instead of Send and Recv, we can use `MPI_Sendrecv_replace()` to simplify the code and improve performance

• Sends then receives a message using a single buffer

```c
int MPI_Sendrecv_replace
    ( void *buf,
      int count,  MPI_Datatype datatype,
      int dest,   int sendtag,
      int source, int recvtag,
      MPI_Comm comm, MPI_Status *status )
```
Parallel Matrix Multiplication
Loop reordering

• The simplest formulation of matrix multiply the “ijk” formulation

\[
\text{for } i:= 0 \text{ to } n-1, \ j:= 0 \text{ to } n-1, \ k:= 0 \text{ to } n-1 \\
C[i,j] += A[i,k] * B[k,j]
\]

• The “kij” formulation is the basis for an efficient parallel algorithm

\[
\text{for } k:= 0 \text{ to } n-1, \ i:= 0 \text{ to } n-1, \ j:= 0 \text{ to } n-1 \\
C[i,j] += A[i,k] * B[k,j]
\]
Formulation

• The matrices may be non-square

\[
\text{for } k := 0 \text{ to } n_3 - 1 \\
\quad \text{for } i := 0 \text{ to } n_1 - 1 \\
\quad \quad \text{for } j := 0 \text{ to } n_2 - 1 \\
\quad \quad \quad C[i,j] += A[i,k] \times B[k,j] \\
\quad \quad \quad C[i,:] += A[i,k] \times B[k,:]
\]

• The two innermost loop nests compute \(n_3\) outer products

\[
\text{for } k := 0 \text{ to } n_3 - 1 \\
\quad C[ :, :) += A[ :, k] \bullet B[k,:]
\]

where \(\bullet\) is outer product
Outer product

- Recall that when we multiply an $m \times n$ matrix by an $n \times p$ matrix…
  we get an $m \times p$ matrix.

- Outer product of column vector $a^T$ and vector $b = \text{matrix } C$
  an $m \times 1$ times a $1 \times n$

  $$a[1,3] \cdot x[3,1]$$

  $$(a,b,c) \cdot (x,y,z)^T = \begin{pmatrix} ax & ay & az \\ bx & by & bz \\ cx & cy & cz \end{pmatrix}$$

  Multiplication table with rows formed by $a[:]$ and the columns by $b[:]$:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>60</td>
<td>90</td>
</tr>
</tbody>
</table>

- The SUMMA algorithm computes $n$ partial outer products:

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

- Each row $k$ of $B$ contributes to the $n$ partial outer products:

  \[
  \text{for } k := 0 \text{ to } n-1 \\
  C[::,::] += A[::,k] \cdot B[k,::]
  \]
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 $= b << N/\max(px,py)$

\[
\text{for } k := 0 \text{ to } n-1 \text{ by } b \\
\quad \text{multicast } A[ :, k:k+b-1 ] \quad \text{Along processor rows} \\
\quad \text{multicast } B[ k:k+b-1, : ] \quad \text{Along processor columns} \\
\quad C += A[:,k:k+b-1] * B[k:k+b-1,:, ] \quad \text{Local MM}
\]

- Each row and column of processors independently participate in a panel broadcast
- Owner of the panel changes with $k$
Parallel matrix multiplication

- Assume $p$ is a perfect square
- Each processor gets an $n/\sqrt{p} \times n/\sqrt{p}$ chunk of data
- Organize processors into rows and columns
- Process rank is an ordered pair of integers
- Assume that we have an efficient serial matrix multiply

```
 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)
```
What is the performance?

for k := 0 to n-1 by b

// Tree broadcast:  \( \lg \sqrt{p} (\alpha + b\beta n/\sqrt{p}) \)
// For long messages:  \( 2((\sqrt{p} - 1)/ p)b\beta n \)

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

// Built in matrix multiply:  \( 2(n/\sqrt{p})^2b \)
C += A[:, k:k+b-1] * B[k:k+b-1, :]

• Total running time:  \( \sim 2n^3/p + 4\beta bn/ \sqrt{p} \)
Highlights of SUMMA

• **Performance**
  - Running time = $2n^3/p + 4\beta bn/\sqrt{p}$
  - Efficiency = $O(1/(1 + \sqrt{p/n^2})$

• **Generality:** non-square matrices, non-square geometries

• Adjust $b$ to **tradeoff latency cost** against **memory**
  - $b$ small $\Rightarrow$ less memory, lower efficiency
  - $b$ large $\Rightarrow$ more memory, high efficiency

• **Low temporary storage**
  - grows like $2bn/\sqrt{p}$

• A variant used in SCALAPACK

R. van de Geign and J. Watts,
“SUMMA: Scalable universal matrix multiplication algorithm,”
[www.netlib.org/ lapack/lawns/lawn96.ps](http://www.netlib.org/ lapack/lawns/lawn96.ps)
Communication domains

- Summa motivates MPI *communication domains*
- Derive communicators that naturally reflect the communication structure along rows and columns of the processor geometry
Communication domains

• A communicator is name space specified by an MPI communicator
• Messages remain within their domain
• Communication domains simplify the code, by specifying subsets of processes that may communicate
• A processor may be a member of more than one communication domain
Splitting communicators

- Each process computes a key based on its rank
- Derived communicators group processes together that have the same key
- Each process has a rank relative to the new communicator
- If a process is a member of several communicators, it will have a rank within each one
Splitting communicators for Summa

- Create a communicator for each row and column
- Group the processors by row
  \[ \text{key} = \text{myid} \mod \sqrt{P} \]
- Thus, if \( P=4 \)
  - Processes 0, 1, 2, 3 are in one communicator because they share the same value of key (0)
  - Processes 4, 5, 6, 7 are in another (1), and so on
MPI support

- `MPI_Comm_split()` is the workhorse

  ```c
  MPI_Comm_split(MPI_Comm comm,
                 int splitKey,
                 int rankKey,
                 MPI_Comm* newComm);
  ```

- A collective call

- Each process receives a new communicator, `newComm`, which it shares in common with other processes having the same `splitKey` value
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
More on Comm_split

```c
MPI_Comm_split(MPI_Comm comm,
               int splitKey,
               int rankKey,
               MPI_Comm* newComm);
```

- Ranks are assigned arbitrarily among processes sharing the same `rankKey` value
- May exclude a process by passing the constant `MPI_UNDEFINED` as the `splitKey`
- A special `MPI_COMM_NULL` communicator will be returned
- Alternative is to enumerate all nodes in a set and call `MPI_Group_incl()` and `MPI_Comm_create()`
Panel Broadcast

- Each row/column calls Bcast, a multicast
- Contributing row/column circulates across and down ward

Foreach step in 0 to n by panel
  Ring Bcast(current column, comm_col)
  Ring Bcast(current row, comm_row)
DGEMM( )
RING_Bcast( double *buf, int count,
       MPI_Datatype type, int root,
       MPI_Comm comm )

MPI_Comm_rank( comm, &rank );
MPI_Comm_size( comm, &np );
if ( rank != root )
    MPI_Recv( buf, count, type, (rank-1+np) mod np,
           MPI_ANY_TAG, comm, &status );
if ( ( rank +1 ) mod np != root )
    MPI_Send(buf, count, type, (rank+1)%np, 0, comm );
Assignment #5

- Implement Dense Matrix Conjugate Gradient
- Groups of 2: MPI or CUDA
- Groups of 3: Both MPI and CUDA