CSE 262
Lecture 7

Parallel Matrix Multiplication
Announcements

• Projects
Today’s lecture

- Cannon’s Parallel Matrix Multiplication Algorithm
- Working with communicators
Recall Matrix Multiplication

- Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$
  
  $A$ is $m \times n$
  
  $B$ is $n \times p$

- Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix

- Different variants, e.g. $ijk$, etc.
**ijk variant**

```plaintext
for i := 0 to n-1
    for j := 0 to n-1
        for k := 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
```
Parallel matrix multiplication

- Organize processors into rows and columns
  - Process rank is an ordered pair of integers
  - Assume $p$ is a perfect square
- Each processor gets an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ chunk of data
- Assume that we have an efficient serial matrix multiply ($\text{dgemm}$, $\text{sgemm}$)
A simple parallel algorithm

- Conceptually, like the blocked serial algorithm
Cost

• Each processor performs $n^3/p$ multiply-adds
• Multiplies a wide and short matrix by a tall skinny matrix
• Needs to collect these matrices via collective communication
• High memory overhead
Observation

• But we can form the same product by computing $\sqrt{p}$ separate matrix multiplies involving $n^2/p \times n^2/p$ matrices and accumulating partial results
  
  for $k := 0$ to $n - 1$
  
  $C[i, j] += A[i, k] \times B[k, j]$;
A more efficient algorithm

- We can form the same product by computing $\sqrt{p}$ separate matrix multiplies involving $n^2/p \times n^2/p$ submatrices and accumulating partial results
  
  \[
  \text{for } k := 0 \text{ to } n - 1 \\
  C[i, j] += A[i, k] \times B[k, j];
  \]

- Move data incrementally in $\sqrt{p}$ phases within a row or column

- In effect, a linear time ring broadcast algorithm

- Modest buffering requirements
Canon’s algorithm

- Implements the strategy
- In effect we are using a ring broadcast algorithm
- Consider block $C[1,2]$

\[
\]
Skewing the matrices

\[
\]

- Before we start, we *preskew* the matrices so everything lines up
- Shift row \(i\) by \(i\) columns to the left using sends and receives
  - Do the same for each column
  - Communication wraps around
- Ensures that each partial product is computed on the same processor that owns \(C[I,J]\), using only shifts
Shift and multiply


- \( \sqrt{p} \) steps
- Circularly shift Rows 1 column to the left, columns 1 row up
- Each processor forms the partial product of local A & B and adds into the accumulated sum in C
Cost of Cannon’s Algorithm

forall \ i=0 \ to \ \sqrt{p} -1
    C\text{Shift-left} \ A[i; :] \ by \ i \hspace{1cm} // \ T= \alpha+\beta n^2/p
\end{array}
\begin{array}{c}
\text{forall} \ j=0 \ to \ \sqrt{p} -1 \\
\ C\text{shift-up} \ B[: , j] \ by \ j \hspace{1cm} // \ T= \alpha+\beta n^2/p
\end{array}
\begin{array}{c}
\text{for} \ \k=0 \ to \ \sqrt{p} -1
\end{array}
\begin{array}{c}
\text{forall} \ i=0 \ to \ \sqrt{p} -1 \ \text{and} \ j=0 \ to \ \sqrt{p} -1
\ C[i,j] += A[i,j]*B[i,j] \hspace{1cm} // \ T = 2n^3/p^{3/2}
\ C\text{Shift-left}A[i; :] \ by \ 1 \hspace{1cm} // \ T= \alpha+\beta n^2/p
\ C\text{shift-up} \ B[: , j] \ by \ 1 \hspace{1cm} // \ T= \alpha+\beta n^2/p
\end{array}
\end{array}
\begin{array}{c}
\text{end forall}
\end{array}
\text{end for}

\begin{array}{c}
T_P = 2n^3/p + 2(\alpha(1+\sqrt{p}) + (\beta n^2)(1+\sqrt{p})/p)
\end{array}
\begin{array}{c}
E_P = T_1 / (pT_P) = (1 + \alpha p^{3/2}/n^3 + \beta \sqrt{p}/n))^{-1}
\approx (1 + O(\sqrt{p}/n))^{-1}
\end{array}
\begin{array}{c}
E_P \to 1 \ as \ (n/\sqrt{p}) \ grows \ [\text{sqrt of data / processor}]
\end{array}
Implementation
Communication domains

- Cannon’s algorithm shifts data along rows and columns of processors
- MPI provides communicators for grouping processors, reflecting the communication structure of the algorithm
- An MPI communicator is a name space, a subset of processes that communicate
- Messages remain within their communicator
- A process may be a member of more than one communicator

Scott B. Baden /CSE 260/ Winter 2014
Creating the communicators

- Create a communicator for each row

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

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

- Each process obtains a new communicator according to the key
- Process rank relative to the new communicator
- Rank applies to the respective communicator only
- Ordered according to myRank
More on Comm_split

MPI_Comm_split(MPI_Comm comm, int splitKey, int rankKey, MPI_Comm* newComm)

• Ranks assigned arbitrarily among processes sharing the same rankKey value
• May exclude a process by passing the constant MPI_UNDEFINED as the splitKey
• Return a special MPI_COMM_NULL communicator
• If a process is a member of several communicators, it will have a rank within each one
Circular shift

• Communication with columns (and rows)
  MPI_Comm_rank(rowComm,&myidRing);
  MPI_Comm_size(rowComm,&nodesRing);
  int next = (myidRng + 1 ) % nodesRing;
  MPI_Ssend(&X,1,MPI_INT,next,0, rowComm);
  MPI_Recv(&XR,1,MPI_INT,
            MPI_ANY_SOURCE,  
            0, rowComm, &status);

• Processes 0, 1, 2 in one communicator because they share the same key value (0)

• Processes 3, 4, 5 are in another (key=1), and so on