Lecture 6

Scaling
Parallel Print function
Advanced Communicators
Matrix Multiplication
Announcements

Assignment #1 return
Assignment #2 due
Assignment #3 has been posted
Quiz #1 on Friday
Performance modeling of stencil methods

• Recall the **stencil operators** in an iterative method

\[
u_{i,j} = \frac{(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{j,i-1})}{4}\]

![Stencil diagram](image)
Model assumptions and definitions: 2D case

- \( T(1,(m,n)) \) = running time of the best serial algorithm on a problem of size \( m \times n \)
- \( T(P,(m,n)) \) = running time on \( P \) processors
- \( T_\gamma(P,(m,n)) \) = **grind time** on \( P \) processors
  - \( T_\gamma(P,(m,n)) = T(P,(m,n))/(m \cdot n \cdot Niter) \)
  - Ideally \( T_\gamma \) is independent of \( m, n, \) and \( P \)
Performance

• 2 components
  – Local computation
  – Ghost cell communication
    – (Convergence check, including global communication)
Local Computation time

- \( T(1,(m,n)) = m \times n \times T_\gamma \) where
  \( T_\gamma = \) grind time on one processor
- For the Blue Horizon IBM SP3 system at the San Diego Supercomputer Center (with Power3 CPUs)
  \( T_\gamma \approx 16 \beta \)

\[
  u_{i,j} = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{j,i-1})/4
\]
Completing the Performance model

- Message passing time \( T(N) = \alpha + 8\beta N \)
  (8-byte double precision numbers)
- Processor geometry is \( p \times q \)
  - Strips or box-like partitions
- \( T(P,(N,N)) = T(1,(m,n)) + T_{local}^{\text{comm}} \)
  \( m = N/p, n = N/q \)
Communication performance for 1D

- \( P \) divides \( N \) evenly
- \( N/P > 2 \)
- For horizontal strips, data are contiguous

\[ T_{\text{comm}} = 2(\alpha + 8\beta N) \]
2D Processor geometry

- Assume $\sqrt{P}$ divides $N$ evenly and $N/\sqrt{P} > 2$
- Ignore the cost of packing message buffers
- $T_{\text{comm}} = 4(\alpha + 8\beta N/\sqrt{P})$
Summing up the performance models

• 1-D decomposition

\[(16N^2 \beta / P) + 2(\alpha + 8\beta N)\]

• 2-D decomposition

\[(16N^2 \beta / P) + 4(\alpha + 8\beta N / \sqrt{P})\]
Comparative performance

• Strip decomposition will outperform box decomposition—resulting in lower communication times—when
  \[ 2(\alpha+8\beta N) < 4(\alpha+8\beta N/\sqrt{P}) \]
• Assuming \( P \geq 2 \):
  \[ N < \left( \frac{\sqrt{P}}{\sqrt{P} - 2} \right) \left( \frac{\alpha}{8\beta} \right) \]
• On SDSC’s IBM SP3 system “Blue Horizon”
  \[ \alpha = 24 \text{ us} \]
  \[ \beta = 1/(390 \text{ MB/sec}) \]
• \( N < 1170 \left( \frac{\sqrt{P}}{\sqrt{P} - 2} \right) \)
• For \( P = 16 \), strips are preferable when \( N < 2340 \),
Parallel speedup and efficiency

• 1-D decomposition

\[ S_p = \frac{T_1}{T_P} = \frac{16N^2\beta}{(16N^2\beta/P + 2(\alpha+8\beta N))} \]
\[ E_p = \frac{S_p}{P} = \frac{16N^2\beta}{(16N^2\beta + 2P(\alpha+8\beta N))} \]
\[ = 1 / (1 + (\alpha+8\beta N)P/(8N^2\beta)) \]

• 2-D decomposition

\[ S_p = \frac{T_1}{T_P} = \frac{16N^2\beta}{(16N^2\beta/P+4(\alpha+8\beta N/\sqrt{P}))} \]
\[ E_p = \frac{S_p}{P} = \frac{16N^2\beta}{((16N^2\beta)+4(\alpha P+8\beta N\sqrt{P}))} \]
\[ = 1 / (1 + (\alpha P+8\beta N\sqrt{P})/(4N^2 \beta)) \]
Putting these formulas to work

• 1-D decomposition

• Let’s plot $E_p$ as a function of $N$, varying $P$ as a parameter

$$E_p = \frac{1}{1 + (\alpha + 8\beta N)P / (8N^2\beta)}$$

• Let’s also plot the fraction of time spent communicating
Parallel speedup and efficiency

N = 1024

N = 128
Communication fraction

\[
\text{N} = 128
\]

\[
\text{N} = 1024
\]
Surface to volume ratio affects performance

• The *surface to volume ratio* of a geometry is the maximum number of points on the surface (perimeter) over all partitions divided by the volume

• As we increase N while leaving P fixed, we decrease the surface to volume ratio, which gives us a measure of the relative cost of communication

• As volume increases, S/V drops
Parallel print function

- Parallel print facility designed by the ptools effort; see [http://www.llnl.gov/CASC/ppf/](http://www.llnl.gov/CASC/ppf/)
- Installed in `${PUB}/lib/PPF`
- Use a special version of the arch file called `arch.valkyrie_ppf`
- Each module that uses the facility must
  
  ```
  #include "ptools_ppf.h"
  ```

- Look in `${PUB}/examples/PPF` for example programs `ppfexample.c` and `test_print.c`
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!
  ```
Matrix Multiplication

• Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$

• Second dimension of $A$ must equal first dimension of $B$
  
  $A$ is $m \times n$
  
  $B$ is $n \times p$

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

function MM(Matrix A, Matrix B, Matrix C)

for i := 0 to n – 1
    for j := 0 to n – 1 do
        C[i, j] = 0;
        for k := 0 to n - 1
            C[i, j] += A[i, k] * B[k, j];
        end for
    end for
end MM
Memory locality

- Data access order affects performance
- 2.7GHz Power PC G5, gcc -O, N=1024 (sgl prec.)

IJ loop: 0.0049 secs. JI loop: 0.27 s.

for (i=0; i<N; i++)
for (j=0; j<N; j++)
    a[i][j] += b[i][j];

```
  0  1  2  3
  4  5  6  7
  8  9 10 11
 12 13 14 15
```
Reuse

• Memory access times are much slower than cache

• An L2 cache miss might cost 50 to 100 cycles, and is increasing

• The success of caching depends on the ability to re-use previously cached data
  – Such re-use exhibits temporal locality
  – Re-use depends on the ability of the application to live within the capacity of the cache

• Blocked algorithms, similar to the parallel algorithm:
  http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect 02.html
Parallel matrix multiplication

- Assume \( p \) is a perfect square
- Each processor gets an \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \) 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

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
A simple parallel algorithm

- Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element.
- Thus, $A[i,k] \times B[k,j]$ is a matrix multiply in $C[i, j] += A[i, k] \times B[k, j]$.
A simple parallel algorithm

- Apply the basic algorithm but treat each element \(A[i,j]\) as a block rather than a single element.
- Thus, \(A[i,k] \times B[k,j]\) is matrix multiply in \(C[i, j] += A[i, k] \times B[k, j]\)

![Diagram showing matrix multiplication and parallel algorithm](image-url)
Cost

- Each processor performs $\frac{n^3}{p}$ multiply-adds
- But a significant amount of communication is needed to collect a row and a column of data onto each processor
- Each processor broadcasts a chunk of data of size $\frac{n^2}{p}$ within a row and a column of $\sqrt{p}$ processors
- Disruptive - distributes all the data in one big step
- High memory overhead
  - needs $2\sqrt{p}$ times the storage needed to hold $A$ & $B$
Observation

- In the broadcast algorithm each processor multiplies two skinny matrices of size $n^2/\sqrt{p}$
- 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

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases, using smaller pieces than with the broadcast approach
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$
Canon’s algorithm

- \( C[1,2] += A[1,k] \times B[k,2] \) for \( k=0, 1, 2 \)

- We want \( A[1,0] \) and \( B[0,2] \) to reside on the same processor initially

- Shift rows and columns so the next pair of values \( A[1,1] \) and \( B[1,2] \) line up

- And so on with \( A[1,2] \) and \( B[2,2] \)
Skewing the matrices

- We first *skew* the matrices so that everything lines up
- Shift each row \( i \) by \( i \) columns to the left using sends and receives
- Communication wraps around
- Do the same for each column
Shift and multiply

• Takes $\sqrt{p}$ steps
• Circularly shift
  – each row by 1 column to the left
  – each column by 1 row to the left
• Each processor forms the product of the two local matrices adding into the accumulated sum

$$C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2)$$
Cost of Canon’s algorithm – Pre skewing

forall \( i=0 \) to \( \sqrt{p} - 1 \)
\[
\text{CShift-left } A[i; :] \text{ by } i \quad \text{// } T = \alpha + \beta n^2/p
\]

forall \( j=0 \) to \( \sqrt{p} - 1 \)
\[
\text{Cshift-up } B[:, j] \text{ by } j \quad \text{// } T = \alpha + \beta n^2/p
\]
Cost of computational loop

for k=0 to $\sqrt{p} - 1$

forall i=0 to $\sqrt{p} - 1$ and j=0 to $\sqrt{p} - 1$

\[ C[i,j] += A[i,j]*B[i,j] \quad \text{// } T = \frac{2n^3}{p^{3/2}} \]

forall i=0 to $\sqrt{p} - 1$

CShift-leftA[i; :] by 1 \quad \text{// } T= \alpha+\beta\frac{n^2}{p}

forall j=0 to $\sqrt{p} - 1$

Cshift-up B[: , j] by 1 \quad \text{// } T= \alpha+\beta\frac{n^2}{p}

end for
Cost of Canon’s algorithm

- $T_p = \frac{2n^3}{p} + 4\sqrt{p} \alpha + 4\beta \frac{n^2}{\sqrt{p}}$
- $E_p = \frac{2n^3}{p \cdot T_p}$
  - $= \frac{1}{1 + 2\alpha (\sqrt{p/n})^3 + 2 \beta \sqrt{p/n}}$
  - $= \frac{1}{1 + O(\sqrt{p/n})}$
Implementing Cannon’s algorithm

- Cannon’s algorithm provides a good motivation for using MPI \textit{communication domains}
- A 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
- We define communicators that naturally reflect the structure of communication along rows and columns of the processor geometry
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 Cannon’s algorithm

• Let’s create a communicator for each row and one for each column
• Consider a grouping of processors by row  
  key = myid \text{ div } \sqrt{P}
• Thus, if P=9, then  
  – Processes 0, 1, 2 are in one communicator because they share the same value of key (0)  
  – Processes 3, 4, 5 are in another (1)  
  – Processes 6, 7, 8 are in a third (2)
MPI support

• **MPI_Comm_split()** is the workhorse

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

• A collective call

• Each process receives a new communicator, which it shares in common with other processes having the same key value
Comm_split

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

• Each process receives a unique rank within its respective communicator according to the value of rankKey

• The relative values of the ranks follows the ordering of the rankKeys across the processes

• I.e. if A give a rank key of 1, and B a rank key of 10, then A’s rank < B’s rank
More on Comm_split

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 from a communicator by passing the constant MPI_UNDEFINED as the splitKey
• A special MPI_COMM_NULL communicator will be returned
Splitting into rows

```c
MPI_Comm rowComm;
int myRow = myid / \sqrt{P};
MPI_Comm_split(MPI_COMM_WORLD, myRow, myid, &rowComm);
```
A ring shift

MPI_Comm_rank(rowComm,&myidRing);
MPI_Comm_size(rowComm,&nodesRing);

int I = myrow, X = …, XR;

int next = (myidRng + 1) % nodesRing;
MPI_Send(&X,1,MPI_INT,next,0,rowComm);
MPI_Recv(&XR,1,MPI_INT,MPI_ANY_SOURCE,0,rowComm,&status);
Other kinds of communication domains

• Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar

• But we can accomplish a good deal of what we want using the splitting method
The Code

• Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar
• But we can accomplish a good deal of what we want using the splitting method
• We use another routine MPI_Sendrecv_replace() to simplify the coding
• Sends then receives a message using one buffer
**MPI_Sendrecv_replace()**

- Sends then receives a message using one 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 )
```