Communication Performance of stencil methods with message passing
Matrix Multiplication
Today’s lecture

• Communication performance of Stencil methods in MPI
• Parallel Print Function
• Matrix Multiplication with Cannon’s Algorithm
Stencil methods under message passing

- Recall the image smooth algorithm

```plaintext
for iter = 1 : nSmooth
    for (i,j) in 0:N-1 x 0:N-1
        Img\text{new}[i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])/4
    end
    Img = Img\text{new}
end
```

Original

100 iter

1000 iter
Parallel Implementation

- Partition data, assigning each partition to a unique process
- Different partitionings according to the *processor geometry*
- Dependences on values found on neighboring processes
- “Overlap” or “ghost” cells hold a copies off-process values
Managing ghost cells

- Post `IReceive()` for all neighbors
- **Send** data to neighbors
- **Wait** for completion
Performance is sensitive to processor geometry

- Aliev- Panfilov method running on triton.sdsc.edu (Nehalem Cluster)
- 256 cores, n=2047, t=10 (8932 iterations)

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GFlops</th>
<th>Gflops w/o Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 8</td>
<td>573</td>
<td>660</td>
</tr>
<tr>
<td>8 x 32</td>
<td>572</td>
<td>662</td>
</tr>
<tr>
<td>16 x 16</td>
<td>554</td>
<td>665</td>
</tr>
<tr>
<td>2 x 128</td>
<td>508</td>
<td>658</td>
</tr>
<tr>
<td>4 x 64</td>
<td>503</td>
<td>668</td>
</tr>
<tr>
<td>128 x 2</td>
<td>448</td>
<td>658</td>
</tr>
<tr>
<td>256 x 1</td>
<td>401</td>
<td>638</td>
</tr>
</tbody>
</table>
Communication costs for 1D geometries

• Assumptions
  ‣ P divides N evenly
  ‣ N/P > 2

• For horizontal strips, data are contiguous
  \[ T_{\text{comm}} = 2(\alpha + 8\beta N) \]
2D Processor geometry

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

• Substituting $T_\gamma \approx 16 \beta$

• 1-D decomposition

\[
\frac{(16N^2 \beta)}{P} + 2(\alpha + 8\beta N)
\]

• 2-D decomposition

\[
\frac{(16N^2 \beta)}{P} + 4(\alpha + 8\beta N/\sqrt{P})
\]
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 \(P \geq 2\): \(N < (\sqrt{P}/(\sqrt{P} - 2))(\alpha/8\beta)\)

• On SDSC’s Triton System
  \(\alpha = 3.2 \text{ us}, \beta = 1/(1.2 \text{ GB/sec})\)
  » \(N < 480(\sqrt{P}/(\sqrt{P} - 2))\)
  » For \(P = 16\), strips are preferable when \(N < 960\)

• On SDSC’s IBM SP3 system “Blue Horizon”
  \(\alpha = 24 \text{ us}, \beta = 1/(390 \text{ MB/sec})\)
  » \(N < 1170 (\sqrt{P}/(\sqrt{P} - 2))\)
  » For \(P = 16\), strips are preferable when \(N < 2340\)
Debugging tips

- Bugs?! Not in my code!
- MPI can be harder to debug than threads
- MPI is a library, not a language
- Command line debugging
- The seg fault went away when I added a print statement
- Garbled printouts
- 2D partitioning is much more involved than 1D
- Indices are swapped

for (int j=1; j<=m+1; j++)
  for (int i=1; i<=n+1; i++)
    \[ E[j][i] = E_{\text{prev}}[j][i] + \alpha (E_{\text{prev}}[j][i+1] + E_{\text{prev}}[j][i-1] - 4E_{\text{prev}}[j][i] + E_{\text{prev}}[j+1][i] + E_{\text{prev}}[j-1][i]); \]
Multidimensional arrays

- Remember that in C, a 2D array is really a 1D array of 1D arrays

```c
double **alloc2D(int m, int n){
    double **E;;
    E = (double**);
    malloc(sizeof(double*)*m + sizeof(double)*n*m);
    assert(E);
    for(int j=0; j<m; j++)
        E[j] = (double*)(E+m) + j*n;
    return(E);
}
```
Today’s lecture

• Communication Performance of Stencil methods in MPI
• Parallel Print Function
• Matrix Multiplication with Cannon’s Algorithm
Parallel print function

- Debug 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 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 on the “make” line
  ‣ Defined in arch.gnu.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 gather(), which we’ll discuss next time
Today’s lecture

• Communication Performance of Stencil methods in MPI
• Parallel Print Function
• Matrix Multiplication with Cannon’s Algorithm
Matrix Multiplication

• An important core operation in many numerical algorithms

• 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
Simplest Serial Algorithm

“ijk”

\[
\text{for } i := 0 \text{ to } n-1
\]
\[
\quad \text{for } j := 0 \text{ to } n-1
\]
\[
\quad \quad \text{for } k := 0 \text{ to } n-1
\]
\[
C[i,j] += A[i,k] \times B[k,j]
\]
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 (\text{dge}\text{mm}, \text{sg}\text{emm})
Canon’s algorithm
- Move data incrementally in $\sqrt{p}$ phases
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Consider iteration $i=1$, $j=2$:


Image: Jim Demmel
Canon’s algorithm


- 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 \( B[2,2] \).
Canon’s algorithm – 1 element of C

\[ C_{1,2} = A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \]

• 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

\[ C_{1,2} = A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \]

- 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
Cost of Cannon’s Algorithm

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

forall \( j = 0 \) to \( \sqrt{p} - 1 \)
    Cshift-up B[\( :, j \)] by \( j \) \hspace{1cm} // T = \alpha + \beta n^2/p

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] \hspace{1cm} // T = 2n^3/p^{3/2}
        CShift-left A[i; :] by 1 \hspace{1cm} // T = \alpha + \beta n^2/p
        Cshift-up B[\( :, j \)] by 1 \hspace{1cm} // T = \alpha + \beta n^2/p

end forall

end for

\[ T_P = 2n^3/p + 2(\alpha(1+\sqrt{p}) + \beta n^2/(1+\sqrt{p})/p) \]
\[ E_P = T_1 / (pT_P) = (1 + \alpha p^{3/2}/n^3 + \beta \sqrt{p}/n))^{-1} \approx (1 + O(\sqrt{p}/n))^{-1} \]
\[ E_P \rightarrow 1 \text{ as (}n/\sqrt{p}\text{) grows [sqrt of data / processor]} \]
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
Establishing row communicators

- Create a communicator for each row and column
- By Row

key = myRank div √P
Creating the communicators

```c
MPI_Comm rowComm;
MPI_Comm_split( MPI_COMM_WORLD,
myRank / √P, myRank, &rowComm);
MPI_Comm_rank(rowComm,&myRow);
```

- Each process obtains a new communicator
- Each 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
Circular shift

• Communication with columns (and rows)
  MPI_Comm_rank(rowComm,&myidRing);
  MPI_Comm_size(rowComm,&nodesRing);
  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);

• 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