Sample codes are available on Ada at

/general/public/training/mpi/Fall2017/part1
/general/public/training/mpi/Fall2017/part2
Introduction to Code Parallelization Using MPI (Part II)

Ping Luo
TAMU HPRC

October 26, 2017

HPRC Short Course – Fall 2017
Outline

- Exploring parallelism
  - Task-parallelism
  - Data-parallelism
- Data distribution
- Self-scheduling
- Matrix-vector multiplication
- Solving the 2D Poisson equation
- Domain decomposition
- MPI/OMP hybrid programming
- MPI/OMP: A Case Study
Important Note On Using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
Exploring Parallelism

- Task-parallelism: The programmer identifies different tasks of a program and distribute the tasks among different processors.

- Data-parallelism: The programmer partitions the data used in a program and distribute them among different processors, each performing similar operations on the subset of data assigned.
3 chefs need to prepare a three-course menu for 12 guests

- Preparing 12 salads
- Preparing 12 steaks
- Preparing 12 deserts

<table>
<thead>
<tr>
<th>Task</th>
<th>4 meals</th>
<th>4 meals</th>
<th>4 meals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>salad</td>
<td>steak</td>
<td>desert</td>
</tr>
<tr>
<td>Task 2</td>
<td>salad</td>
<td>steak</td>
<td>desert</td>
</tr>
<tr>
<td>Task 3</td>
<td>salad</td>
<td>steak</td>
<td>desert</td>
</tr>
</tbody>
</table>

Task parallelism

Data parallelism
Example 6: Data Distribution

Data is distributed in a round robin manner among the processes.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=myid+1; i&lt;=N; i+=nprocs){</td>
<td>do i=myid+1, N, nprocs</td>
</tr>
<tr>
<td>x = h*(i-0.5);</td>
<td>x = h*(i-0.5d0)</td>
</tr>
<tr>
<td>sum += 4.0/(1.0+x*x);</td>
<td>sum = sum+4.0d0/(1.0d0+x*x)</td>
</tr>
<tr>
<td>}</td>
<td>enddo</td>
</tr>
<tr>
<td>sum = sum*h;</td>
<td>sum = sum*h;</td>
</tr>
</tbody>
</table>

calc_PI_cyclic.c
Example 6: Data Distribution

Data is partitioned into $n$ contiguous parts, where $n$ is equal to the number of processes. Each process will take one part of the data.

<table>
<thead>
<tr>
<th>Block Distribution</th>
<th>data id</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process id</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Data is partitioned into $n$ contiguous parts, where $n$ is equal to the number of processes. Each process will take one part of the data.

C

```c
block_map(1,N,nprocs,myid,&l1,&l2);
for (i=l1; i<=l2; i++){
    x = h*(i-0.5);
    sum += 4.0/(1.0+x*x);
}
sum = sum*h;
```

Fortran

```fortran
call block_map(1,N,nprocs,myid,l1,l2)
do i=l1, l2
   x = h*(i-0.5d0)
   sum = sum + 4.0d0/(1.0d0+x*x)
endo
d0
sum = sum*h
```

`calc_PI_block.c`
Example 6: Data Distribution

<table>
<thead>
<tr>
<th>data id</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Distribution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process id</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

C

void block_map(int n1, int n2, int nprocs, int myid, int *l1, int *l2)
{
    int block, rem;
    block = (n2-n1+1)/nprocs;
    rem = (n2-n1+1)%nprocs;
    if (myid < rem){
        block++;
        *l1 = n1+myid*block;
    }else
        *l1 = n1+rem+block*myid;
    *l2 = *l1+block-1;
}

Fortran

subroutine block_map(n1,n2, nprocs, myid, l1, l2)
Integer n1, n2, nprocs, myid, l1,l2

integer block, rem
block = (n2-n1+1)/nprocs
rem = mod(n2-n1+1, nprocs)
if (myid < rem) then
    block = block+1
    l1 = n1+myid*block
else
    l1 = n1+rem+block*myid
end if
l2 = l1+block-1
end subroutine block_map
Example 6: Data Distribution

Data is divided into chunks of contiguous blocks and the chunks distributed in a round-robin manner.

```
<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| for (i=myid*BLK+1; i<=N; i+=nprocs*BLK){
  for (j=i; j<=MIN(N,i+BLK-1); j++){
    x = h*(j-0.5);
    sum += 4.0/(1.0+x*x);
  }
} sum = sum*h;             | do i=myid*BLK+1, N, nprocs*BLK |
|                            |    do j=i, MIN(N,i+BLK-1)    |
|                            |       x = h*(j-0.5d0)        |
|                            |       sum = sum+4.0d0/(1.0d0+x*x) |
|                            |     enddo                    |
|                            |   enddo                      |
|                            |  sum = sum*h                 |
```

```
calc_PI_bc.c
```
Example 7: Self-Scheduling

\[ A\hat{b} = \begin{pmatrix} \hat{a}_1 \\ \vdots \\ \hat{a}_n \end{pmatrix} \hat{b} = \begin{pmatrix} \hat{a}_1 \cdot \hat{b} \\ \vdots \\ \hat{a}_n \cdot \hat{b} \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} = \hat{c} \]

The algorithm distributes the computation of the dot product to any process. Each dot product is independent of the other.

**Master process**

“owns” A and b, and collects \( c = A \times b \)

1. Broadcast b each worker
2. Use tag to signify row number; send a row to each worker.
3. Loop:
   - receive a dot_product entry from worker_i
   - send a new row to worker_i from whom we just received the new dot_product;
   Until all dot_products are received.

**Worker process**

1. Receive b
2. Loop:
   - receive a new row from master
   - compute dot product
   - send dot product to master
Until master sends termination notice.

**Self scheduling**: also called work load distribution scheme whereby a “master” process hands work to an idle worker process.
Example 7: Self-Scheduling

MPI_Bcast(b, n, MPI_DOUBLE, master, 0, MPI_COMM_WORLD)
Example 7: Self-Scheduling

MPI_Send(a(i-1), n, MPI_DOUBLE, i, i-1, MPI_COMM_WORLD)

I = 1, 3

MPI_Recv(a_row, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)

row_num = status.MPI_TAG

The tag holds the row number
Example 7: Self-Scheduling

For i = 0 to n-1

\[ \text{MPI_Recv(ans, 1, MPI\_DOUBLE, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, status)} \]

source = status.MPI\_SOURCE
row_num = status.MPI\_TAG
c(row_num) = ans

\[ \text{MPI_Recv(a_row, n, MPI\_DOUBLE, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD)} \]
row_num = status.MPI\_TAG; do dot product;

\[ \text{MPI\_Send(ans, 1, MPI\_DOUBLE, 0, row\_num, MPI\_COMM\_WORLD)} \]
Example 7: Self-Scheduling

For $i = 0$ to $n-1$

```c
MPI_Recv(ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)
```

source = status.MPI_SOURCE
row_num = status.MPI_TAG
c(row_num) = ans

Worker 2 just finished and need more work

```c
MPI_Recv(a_row, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)
```

row_num = status.MPI_TAG; do dot product;

```c
MPI_Send(ans, 1, MPI_DOUBLE, 0, row_num, MPI_COMM_WORLD)
```
Example 7: Self-Scheduling

For $i = 0$ to $n-1$

```
MPI_Recv(ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)
```

source = status.MPI_SOURCE
row_num = status.MPI_TAG
c(row_num) = ans

```
MPI_Send(a(index), n, MPI_DOUBLE, source, index, MPI_COMM_WORLD)
```

```
MPI_Send(ans, 1, MPI_DOUBLE, 0, row_num, MPI_COMM_WORLD)
```

$\text{Master 0}$

$\text{Worker 1}$

$\text{Worker 2}$

$\text{Worker 3}$
Example 7: Self-Scheduling

For i = 0 to n-1

MPI_Recv(ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)

source = status.MPI_SOURCE
row_num = status.MPI_TAG
ans = c(row_num)

MPI_Send(a(index), n, MPI_DOUBLE, source, index, MPI_COMM_WORLD)

MPI_Recv(a_row, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)
row_num = status.MPI_TAG; do dot product;
MPI_Send(ans, 1, MPI_DOUBLE, 0, row_num, MPI_COMM_WORLD)
Example 7: Self-Scheduling

For $i = 0$ to $n-1$

- MPI_Recv($\text{ans}$, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, $\text{status}$)

- $\text{source} = \text{status.MPI_SOURCE}$
- $\text{row_num} = \text{status.MPI_TAG}$
- $\text{c(row_num)} = \text{ans}$

- MPI_Send($\text{a(index)}$, $n$, MPI_DOUBLE, $\text{source}$, $\text{index}$, MPI_COMM_WORLD)

- MPI_Recv($\text{a_row}$, $n$, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)

- $\text{row_num} = \text{status.MPI_TAG}$; do dot product;
- MPI_Send($\text{ans}$, 1, MPI_DOUBLE, 0, $\text{row_num}$, MPI_COMM_WORLD)
Example 7: Self-Scheduling

For $i = 0$ to $n-1$

1. **MPI_Recv**($\text{ans}$, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)

2. \text{source} = status.MPI_SOURCE

3. \text{row_num} = status.MPI_TAG

4. \text{c(row_num)} = \text{ans}

5. **MPI_Send**($\text{a(index)}$, n, MPI_DOUBLE, \text{source}, index, MPI_COMM_WORLD)

6. **MPI_Recv**($\text{a_row}$, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)

7. \text{row_num} = status.MPI_TAG; do dot product;

8. **MPI_Send**($\text{ans}$, 1, MPI_DOUBLE, 0, \text{row_num}, MPI_COMM_WORLD)
Example 7: Self-Scheduling

For $i = 0$ to $n-1$

```python
MPI_Recv(ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)

source = status.MPI_SOURCE
row_num = status.MPI_TAG
ans = status.MPI_TAG
MPI_Send(a(index), n, MPI_DOUBLE, source, index, MPI_COMM_WORLD)
```

```python
MPI_Recv(a_row, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)
row_num = status.MPI_TAG; do dot product;
MPI_Send(ans, 1, MPI_DOUBLE, 0, row_num, MPI_COMM_WORLD)
```
For $i = 0$ to $n-1$

```c
MPI_Recv(ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status)
```

source = status.MPI_SOURCE
row_num = status.MPI_TAG
$\mathbf{c}(row_num) = \mathbf{ans}$
index = index+1
if (index == n)
  MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, source, index, MPI_COMM_WORLD)
else
  MPI_Send($\mathbf{a}(index)$, n, MPI_DOUBLE, source, index, MPI_COMM_WORLD)

MPI_Recv($\mathbf{a}_row$, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD)
row_num = status.MPI_TAG
If (row_num < n)
do dot product
  MPI_Send($\mathbf{ans}$, 1, MPI_DOUBLE, 0, row_num, MPI_COMM_WORLD)
Example 8: matvec-scatterv

\[ A\vec{b} = (\vec{a}_1 \ldots \vec{a}_n)\vec{b} = b_1\vec{a}_1 + b_2\vec{a}_2 + \ldots + b_n\vec{a}_n = \vec{c} \]

\( \vec{a}_i \) is a column vector.

In this program, strips of consecutive columns of \( A \) are distributed to all processes. Each process carries out a part of the linear vector sum

\[ b_i\vec{a}_i + \ldots + b_j\vec{a}_j \]
Example 9: Solving the x-y Poisson Equation

Solve the partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

where $x, y \in [0,1]$ and $u = g(x, y)$ on boundary

using the finite difference method.

Discretize the domain along $x$ and $y$ using $n$ internal points in each direction.

The increment is $h = 1/(n + 1)$

$$x_i = ih, y_j = jh$$

($0 \leq i, j \leq n + 1$)

$$u_{ij} = u(x_i, y_j) = u(ih, jh)$$

($0 < i, j < n + 1$)
Example 9: the x-y Poisson Equation

5-point finite difference stencil approximation:
\[ u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2 f_{i,j} \quad (0 < i, j < n + 1) \]
\[ u_{i,j} = 0.25 \times (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}) \]

k+1 Jacobi iteration step at \( x_i = ih; \ y_j = jh; i, j = 1: n \)
\[ u^{k+1}_{i,j} = 1/4(u^k_{i-1,j} + u^k_{i+1,j} + u^k_{i,j-1} + u^k_{i,j+1}) - h^2 f_{i,j} \]

Jacobi iteration across all points:
\[ \text{do } j=1, n \]
\[ \quad \text{do } i = 1, n \]
\[ \quad \text{unew}(i, j) = 0.25 \times (u_{i-1,j} + u_{i,j+1} + u_{i+1,j} + u_{i,j-1}) - f_{i,j} \times h^2 \]
\[ \text{end do} \]
\[ \text{end do} \]
Example 9: Solving the x-y Poisson Equation

Boundary conditions (stay fixed):

\[
\begin{aligned}
&u(x_i, 0) = \frac{\cos(\pi x_i) - \pi^2}{\pi^2} & (0 \leq x \leq 1) \\
&u(x_i, 1) = \frac{\cos(\pi x_i) - \pi^2 \cos(x_i)}{\pi^2} & (0 \leq x \leq 1) \\
&u(0, y_j) = \frac{1}{\pi^2} - 1 & (0 \leq y \leq 1) \\
&u(1, y_j) = -\left(\frac{1}{\pi^2} + \cos(y_j)\right) & (0 \leq y \leq 1)
\end{aligned}
\]

RHS: \[ f(x_i, y_j) = (x_i^2 + y_j^2) \cos(x_i y_j) - \cos(\pi x_i) \]
1d-Domain Decomposition

C: decompose along y axis

Fortran: decompose along x axis
1d-Domain Decomposition

Border columns need to be copied into the memory space of the neighboring processes for the five-point stencil calculation. Same for row-wise decomposition.
1d-Domain Decomposition

Exchanged border columns are stored in 'ghost' columns in each process to be used in five-point stencil calculation.
Example 10: Hybrid Programming

Int MPI_Init_thread(int *argc, char **argv, int required, int *provided)

MPI_INIT_THREAD(required, provided, ierr)

- Four possible values for the parameter required:
  - MPI_THREAD_SINGLE
  - MPI_THREAD_FUNNELED
  - MPI_THREAD_SERIALIZED
  - MPI_THREAD_MULTIPLE

- To compile
  mpiicc -qopenmp [options] prog.c –o prog.exe
  mpiifort –qopenmp [options] prog.f90 –o prog.exe
## MPI+OMP Hybrid Programming

- Not all programs will benefit from hybrid programming

### poisson_1d performance comparison: pure MPI vs hybrid

<table>
<thead>
<tr>
<th>np</th>
<th>Pure MPI Single-node</th>
<th>Pure MPI 4-node</th>
<th>Hybrid OMP_NUM_THREADS=2</th>
<th>Hybrid OMP_NUM_THREADS=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22.8697 seconds</td>
<td>13.5297 seconds</td>
<td>25.0954 seconds</td>
<td>49.7846 seconds</td>
</tr>
<tr>
<td>4</td>
<td>11.9441 seconds</td>
<td>7.7067 seconds</td>
<td>13.3360 seconds</td>
<td>25.0660 seconds</td>
</tr>
<tr>
<td>8</td>
<td>6.5824 seconds</td>
<td>4.6635 seconds</td>
<td>7.5272 seconds</td>
<td>13.3054 seconds</td>
</tr>
<tr>
<td>16</td>
<td>3.6141 seconds</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

- example9/poisson_1d.c (.f90)
- example10-hybrid/poisson_1d_hybrid.c

**Hybrid is worse**
MPI+OMP: A Case Study

- The nlace code developed by Prof. Sevan Goenezen’s group from the TAMU ME department is used to estimate the non-homogeneous elastic material properties using force and surface displacement data from multiple measurements.
- Was partially parallelized with OpenMP
  - The inverse solver has a good speedup with up to 3 CPU cores. Increasing number of cores won’t help to speedup the code further
- Cannot process 3D cases due to extremely slow running time.
Profiling

- gradfun (99.7%)
- stiff_residl (70.59%)
- residl_dual (%12.35)
- calcgrad (%12.32)

Paralleled with OpenMP

Called once only:
- Forward solver
- Initialization
Amdahl’s Law

\[ S_n = \frac{1}{p \frac{n}{n} + (1 - p)} \]

- \( S_n \) – speedup of the parallel code on \( n \) core vs the serial code
- \( p \) – percentage of the code that can be parallelized
- \( n \) – number of CPU cores used to run the code

According to Amdahl’s law, \( S_n \) is bounded by the serial part of the code that cannot benefit from increasing the number of CPU cores.

The speedup of the original code cannot exceed 4 due to the 25% of serial code.
Results

- Parallelized the serial portion of the code with OpenMP
- Parallelized the entire code with MPI which has successfully explored the data parallelism among different measurements.

<table>
<thead>
<tr>
<th></th>
<th>Num of Cores</th>
<th>Wall Clock Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original OpenMP</td>
<td>3</td>
<td>108m</td>
<td>1</td>
</tr>
<tr>
<td>Improved OpenMP</td>
<td>10</td>
<td>38m47s</td>
<td>2.8</td>
</tr>
<tr>
<td>OpenMP+MPI</td>
<td>100</td>
<td>3m20s</td>
<td>32.7</td>
</tr>
</tbody>
</table>
References

- A significant amount of content is adapted from former Supercomputing Facility staff Mr. Spiros Vellas’ introductory MPI short course
- Two books: <<Using MPI>> and <<Using MPI II>>