Understanding MPI Applications: A Perspective of Parallel Algorithms

Xiaoxu Guan

High Performance Computing, LSU

May 31, 2016
Overview

• Requirements for Parallel Computing
Overview

- Requirements for Parallel Computing
- Fundamental Steps of Designing Parallel Algorithms
Overview

• Requirements for Parallel Computing
• Fundamental Steps of Designing Parallel Algorithms
• Foster’s Methodology
  ◦ Partitioning;
  ◦ Data Communication;
  ◦ Agglomeration;
  ◦ Mapping;
Overview

- Requirements for Parallel Computing
- Fundamental Steps of Designing Parallel Algorithms
- Foster’s Methodology
  - Partitioning;
  - Data Communication;
  - Agglomeration;
  - Mapping;
- Potential Pitfalls and Maintaining Good Performance
Overview

- Requirements for Parallel Computing
- Fundamental Steps of Designing Parallel Algorithms
- Foster’s Methodology
  - Partitioning;
  - Data Communication;
  - Agglomeration;
  - Mapping;
- Potential Pitfalls and Maintaining Good Performance
- Three MPI Examples
  - Find Prime Numbers
  - MPI Input/Output
  - Matrix-Vector Products
  - Benchmark an MPI Application
Overview

• Requirements for Parallel Computing
• Fundamental Steps of Designing Parallel Algorithms
• Foster’s Methodology
  ◦ Partitioning;
  ◦ Data Communication;
  ◦ Agglomeration;
  ◦ Mapping;
• Potential Pitfalls and Maintaining Good Performance
• Three MPI Examples
  ◦ Find Prime Numbers
  ◦ MPI Input/Output
  ◦ Matrix-Vector Products
  ◦ Benchmark an MPI Application
• Further Reading
Parallel computing

- Requirements for Parallel Computing
- How does MPI meet these requirements?
  - Specify parallel execution – single program on multiple data (SPMD) and tasks;
  - Data communication – two- and one- side communication (explicit or implicit);
  - Synchronization – synchronization functions;
- Data parallelism;

```
1 for i from imin to imax, do
2   c(i) = a(i) + b(i)
3 end do
```

- Task (functional) parallelism;

```
1 { for c(i) = a(i) + b(i) }
2 { for d(j) = sin(a(j)) }
```
Parallel programming

- Fundamental steps of designing parallel algorithms;
- Shall we design parallel algorithms based on the existing serial algorithms? **Think in parallel!**
- **Foster** model:
  - (1) **Partitioning**
    Divide a large problem into many small ones (tasks);
    Domain decomposition;

  \[ \psi(x, y) \]

- **Load balance**: be sure that each task has the same or similar amount of data to process;
Parallel programming

- **(2) Data communication**
- Unless your application doesn’t need any exchange of data (trivial parallelism), we have to deal with data communication between different tasks;
  - **Local communication**: for a given task it only needs to talk to a very limited number of other tasks;
  - **Global communication**: a relatively large number of tasks are involved;
- Data communication is not free!
- **Reduce** the number of data communication calls and reduce the amount of data that needs to be transferred;
- Be sure that each MPI task has the **same** or **nearly the same** number of communication calls and amount of data;
Parallel programming

- **(3) Agglomeration**
  - This is related with the overhead of data communications;
  - Trade-off between the number of MPI tasks and the overhead of data communication;
  - Combines several small tasks into a larger task;
  - Sometimes, reducing the number of MPI tasks might improve the data locality;
  - Generally, a rule of thumb is that sending/receiving fewer but longer messages is better than sending/receiving more, but shorter messages;
- **More** computation and **less** communication;
Parallel programming

- **(4) Mapping**
- How were multiple tasks assigned to multiple cores?
  - Generally, this is probably the most difficult step;
  - Maximize CPU utilization and minimize data communication;
  - Something beyond load balance: internode and intranode communication;
- For a given size of the problem and fixed number of cores, how shall we assign tasks to cores: static and dynamic?
  - Static: (1) load balance; (2) regular communication pattern; (3) one task/core; (4) each core plays almost the same role;
  - Dynamic: master-worker model and dispatches tasks to available cores;
- Maintain load balance (computation and communication) and make the code scalable;
Potential Pitfalls

• Some common reasons for MPI code hanging or deadlock;
• Message passing should not be overtaking;

(1) MPI_Recv does not match MPI_Send (rank or tag).
  • There is a MPI_Send, but no matching MPI_Recv;
  • There is a MPI_Recv, but no matching MPI_Send;

(2) Collective MPI calls are not called so by all MPI ranks in the communicator (say, the issue with only one rank calling MPI_Bcast);
The Sieve of Eratosthenes for Prime Numbers
Find Prime Numbers

- **MPI** programming for prime number searching below $N$;
- **The serial Sieve of Eratosthenes** algorithm;
- One of the ancient but effective iterative methods;

**Step 1.** Generate a list for $2, 3, 4, \ldots, \text{and } N$;

**Step 2.** Let $k = 2$, the first prime in the list;

**Step 3.** Repeat the following procedure:

- Delete all multiples of $k$ in the region $[k^2, N]$.
- Locate the smallest number $> k$. Set the new $k$ to it.
- Until $k^2 > N$.

**Step 4.** All remaining numbers are primes.

- Let’s consider $N = 55$;
The Sieve of Eratosthenes

<table>
<thead>
<tr>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

2n, 3n, 5n,
## The Sieve of Eratosthenes

|  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |

$n$, $2n$:

|  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |

$n$, $2n, 3n$:

|  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |

$n$, $2n, 3n, 5n$, $7n$:

|  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
The Sieve of Eratosthenes

- How can we parallelize it using MPI?
- Domain (or data) decomposition;

  1. Break up the entire list into many smaller consecutive blocks (Partitioning);
  2. Shall data communication occur locally or globally (data communication)?
  3. We combine the searching multiple of $k$ and marking them out as a larger task (agglomeration);
  4. We can assign one block to one MPI task (mapping);

- In this case, we have global data communication, because each MPI task needs to know the value of $k$;
- How often do we need to make data communication?
The Sieve of Eratosthenes

- **Load balance**: Let's consider $N = 2016$ on 10 cores; block size is 201 for all cores, except the last task has 206;

- **Can we do better?**
  1. $r = \text{mod}(N - 1, p)$;
  2. block size of $\left\lfloor \frac{(N - 1)}{p} \right\rfloor$ for the first of the $r$ MPI tasks,
  3. block size of $\left\lceil \frac{(N - 1)}{p} \right\rceil$ for the rest of the $p - r$ MPI tasks;

- 5 MPI tasks have the block size of 201, and the rest of the 5 tasks have the block size of 202;

- A much **better** data distribution and it's quite general!

- **Version 0**: (1) istart, iend; (2) primes(:), marked(:); (3) search all integers in $[\text{istart}, \text{iend}]$ for multiple of $k$; (4) call MPI_Allreduce to determine the next global $k$;
The Sieve of Eratosthenes

1  ALLOCATE(idata(istart:iend),marked(istart:iend))
2  marked = .true. ;  k = 2
3  do while( k*k <= pmax )
4      istart_min = max(istart+2,k*k) - 2
5      iend_max = min(iend+2,pmax) - 2
6        do i = istart_min, iend_max
7            itemp = mod(idata(i),k)
8            if(itemp == 0) marked(i) = .false.
9        end do ;  kmin = pmax
10       do i = istart, iend
11          if( marked(i).and.idata(i) > k ) then
12            kmin = idata(i)
13          EXIT ; end if ; end do
14       call MPI_ALLREDUCE(kmin,k,1,MPI_INTEGER, &
15                        MPI_MIN,MPI_COMM_WORLD,ierr)
16     end do
The Sieve of Eratosthenes

1. ALLOCATE(idata(istart:iend),marked(istart:iend))
2. marked = .true. ; k = 2
3. do while( k*k <= pmax )
4.   istart_min = max(istart+2,k*k) - 2
5.   iend_max = min(iend+2,pmax) - 2
6.   do i = istart_min, iend_max
7.     itemp = mod(idata(i) i+2,k)
8.     if(itemp == 0) marked(i) = .false.
9.   end do ; kmin = pmax
10.  do i = istart, iend
11.    if( marked(i).and.idata(i) i+2 > k ) then
12.      kmin = idata(i) i+2
13.    EXIT ; end if ; end do
14.  call MPI_ALLREDUCE(kmin,k,1,MPI_INTEGER, &
15.                      MPI_MIN,MPI_COMM_WORLD,ierr)
16. end do
The Sieve of Eratosthenes

• Can we do better? Take a look at the do loop (lines 6-9):

```c
marked = (bool *) malloc (chunk*sizeof(bool));
for(i=0; i<chunk; i++) marked[i] = true;  // C
iend_max = MIN(iend+2,pmax); k = 2;  // version 2
do {
  istart_min = MAX(istart+2,k*k);
  rmn = istart_min % k;
  if(rmn != 0) istart_min = istart_min - rmn + k;
  for(i = istart_min; i <= iend_max; i+=k) {
    marked[i-istart-2] = false;
  }
  kmin = pmax;
  for( i = istart; i <= iend; i++) {
    if( marked[i-istart-2] && i > k ) {
      kmin = i; break;
    }
  }
  MPI_Allreduce(&kmin,&k,1,MPI_INT, MPI_MIN,MPI_COMM_WORLD);
} while ( k*k <= pmax );
```
The Sieve of Eratosthenes

- Can we do **even** better? Delete all **even** integers!

```c
1  k = 3;
2  do {
3  istart_min = MAX(istart,k*k);
4  rmn = istart_min % k;
5  if(rmn != 0) istart_min = istart_min - rmn + k;
6  for(i = istart_min; i <= iend_max; i+=k) {
7      if( i%2 != 0 ) { lk = (i-istart)/2;
8          marked[lk] = false; }
9  }
10  kmin = pmax; for( i=istart; i<=iend; i+=2) {
11      llk=(i-istart)/2;
12      if( marked[llk] && i>k ) { kmin = i; break; }
13  }
14  MPI_Allreduce(&kmin,&k,1,MPI_INT, MPI_MIN,MPI_COMM_WORLD);
15  } while ( k*k <= pmax );
```
Exercises 1 & 2

- **Exercise 1**: Based on the `mpi_primes_v3`, replace the collective `MPI_Allreduce` by other MPI data communications.
  
  **[Hint]** Break up `MPI_Allreduce` into two MPI commands.

- **Exercise 2**: We have found out the number of primes below $N$. Starting from `mpi_primes_v3`, add the necessary code segment to print out all the primes from small to large below $N$.
  
  **[Hint]** Let all other MPI tasks send the data to the master, and let the master print the primes out.
MPI Input/Output
MPI Input/Output

- The next problem we face is the parallel **MPI I/O**;
- *(1)* Assign **one** MPI task to take care of all the I/O, and send (receive) the necessary data to (from) other MPI tasks;
- *(2)* Each MPI task handles the **same** input or output file, but works on a **different** part of the file (the **best** solution);
 MPI Input/Output

- The next problem we face is the parallel MPI I/O;
- **(1)** Assign one MPI task to take care of all the I/O, and send (receive) the necessary data to (from) other MPI tasks;
- **(2)** Each MPI task handles the same input or output file, but works on a different part of the file (the best solution);

```
core 0 RAM  disk file 0
core 1 RAM  disk file 1
core 2 RAM  disk file 2
core 3 RAM  disk file 3
```

serial output

parallel output (trivial)
MPI Input/Output

- The next problem we face is the parallel MPI I/O;
- (1) Assign one MPI task to take care of all the I/O, and send (receive) the necessary data to (from) other MPI tasks;
- (2) Each MPI task handles the same input or output file, but works on a different part of the file (the best solution);

serial output

parallel output (better)
MPI Input/Output

- The next problem we face is the parallel MPI I/O;
- (1) Assign one MPI task to take care of all the I/O, and send (receive) the necessary data to (from) other MPI tasks;
- (2) Each MPI task handles the same input or output file, but works on a different part of the file (the best solution);

![Diagram of MPI I/O](image)
MPI Input/Output

• (1) One MPI collects all info, and makes the I/O; the amount of data gathered from all MPI tasks may not be the same;
• MPI_Gatherv(∗sbuf, int scount, MPI_Datatype stype, ∗rbuf, int ∗rcounts, int ∗displs, MPI_Datatype rtype, int root, MPI_Comm comm);

For root, define

displs=(int *) malloc (numprocs*sizeof(int));
displs[i] = \sum_{k=0}^{i-1} s[k];
displs[0]=0;
displs[1]=s[0];
displs[2]=s[0]+s[1];
...

Not really MPI I/O!
**MPI Input/Output**

- (2) Each MPI task handles the **same** input or output file;

```c
int MPI_File_open(MPI_Comm comm, char *dfilename, 
                   int amode, MPI_Info info, MPI_File *fh);

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, 
                       MPI_Datatype etype, MPI_Datatype filetype, 
                       char *datarep, MPI_Info info);

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, 
                      void *buf, int count, MPI_Datatype datatype, 
                      MPI_Status *status);
```

For all MPI tasks, set `disp=0;` (global offset)
local offset;
For instance, for core 2:
MPI Input/Output

- MPI also supports **nonblocking** I/O;

```c
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, \
           *buf, int count, MPI_Datatype datatype, \n           MPI_Request *request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

- **Overlap** computation or communication with the I/O;
- Note that files are in **binary** or **unformatted**;
- How can we assure that the output file makes sense?
- One way to check is to measure the **file size** and compare with what it should be;

```c
int MPI_File_get_size(MPI_File fh, MPI_Offset *size);
```
- size is in bytes;
MPI Matrix-Vector Multiplications
MPI matrix-vector products

- Matrix-vector multiplications are very common in physics, applied math, and engineering;
- Many practical problems can be represented in the matrix form, and it is likely matrix-vector products and systems of linear equations need to be handled. **Iterative** methods in linear algebra depend on matrix-vector products;
- Matrix-matrix products can be reduced to multiple matrix-vector products;
- Let’s say we need to compute a **power** of matrices operating on a vector: \( c = A^k \cdot b = AAA \cdots A \cdot b; \)
- Remember FLOPS for square matrix-matrix product \( \sim O(n^3), \) while matrix-vector \( \sim O(n^2); \)
- \( c_i = \sum_j A_{ij} b_j \)
MPI matrix-vector products

How can we parallelize it using MPI?

At least three options $A$:
1. Column-wise block decomposition;
2. Row-wise block decomposition;
3. 2D domain decomposition;

In all three cases, how should we distribute vectors $b$ and $c$?
MPI matrix-vector products

• (1) **Column-wise** block decomposition;

- Maintain **load balance**; each MPI task takes *(almost) same* number of columns;
- The same strategy as those of primes;
- Vectors are block striped;
- Vectors $b$ and $c$ are handled in the **same** way;
MPI matrix-vector products

- **(1) Column-wise** block decomposition;
  - Maintain **load balance**;
  - each MPI task takes **(almost) same** number of columns;
  - The same strategy as those of primes;
  - Vectors are block striped;
  - Vectors $b$ and $c$ are handled in the **same** way;
  - What about data **communication**?
MPI matrix-vector products

- **Data communication** in column-wise decomposition;

1. A MPI task computes its own contributions to a vector element;
2. A task needs to gather the contributions from all other tasks;
3. It sums up all contributions;
4. Different tasks may have different numbers of vector elements;
5. Use `MPI_Alltoallv`;

![Diagram showing MPI matrix-vector products](image)
MPI matrix-vector products

- **Data communication** in column-wise decomposition;
- We use **MPI I/O** to read in all matrix and vector elements;

```fortran
1  do i = 1, nsize
2       c_local_temp(i) = 0.0_idp
3  do j = istart, iend
4       c_local_temp(i) = c_local_temp(i) &
5               + matrix(i,j) * vector_input(j)
6  end do
7  end do

subroutine matvec()
```

- call **MPI_Alltoallv**;
- The difference between **MPI_Gatherv** and **MPI_Alltoallv**;
- After gathering all pieces of data from **other tasks** and **itself**, each MPI task needs to **reorganize** the data to obtain the final output vector;
MPI matrix-vector products

- The difference between `MPI_Gatherv` and `MPI_Alltoallv`;

```
<table>
<thead>
<tr>
<th>Communicator</th>
<th>core 0</th>
<th>sbuf[ ]</th>
<th>scount</th>
</tr>
</thead>
<tbody>
<tr>
<td>core 1</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 2</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 3</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 4</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 5</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Communicator</th>
<th>core 0</th>
<th>sbuf[ ]</th>
<th>scount</th>
</tr>
</thead>
<tbody>
<tr>
<td>core 1</td>
<td>rbuf[ ]</td>
<td>rcounds[ ]</td>
<td></td>
</tr>
<tr>
<td>core 2</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 3</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 4</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
<tr>
<td>core 5</td>
<td>sbuf[ ]</td>
<td>scount</td>
<td></td>
</tr>
</tbody>
</table>
```

`MPI_Gatherv` vs `MPI_Alltoallv`
MPI matrix-vector products

- The difference between MPI_Gatherv and MPI_Alltoallv;

```
core 0
12
0
1
2
2

core 1
9
0
1
2
2

core 2
9
0
1
2
2

Communicator

core 0
sbuf[]
scount

core 1
sbuf[]
scount

core 2
sbuf[]
scount

core 3
sbuf[]
scount

core 4
sbuf[]
scount

core 5
sbuf[]
scount

rcounts[]
```
MPI matrix-vector products

- Again, can we do better?
- Remember Fortran stores 2D arrays in **column-wise**, while C stores 2D arrays in **row-wise**;
- **Fortran version 0** is not optimized in terms of the way the matrix elements are addressed;

```fortran
1    c_local_temp = 0.0_idp
2     do j = istart, iend
3       do i = 1, nsize
4          c_local_temp(i) = c_local_temp(i) &
5                  + matrix(i,j) * vector_inp(j)
6        end do
7     end do
```

- Exchange the loops;
- The compilers wouldn’t do this for you;
### MPI matrix-vector products

- **(2) Row-wise** block decomposition;

- Maintain **load balance**; each MPI task takes **(almost) same** number of rows;

- The same strategy as those of column-wise;

- What about vectors?

- Vectors $b$ and $c$ are handled in the **same** way;

---

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>sum 0 ($c_1$)</th>
<th>sum 1 ($c_1$)</th>
<th>sum 2 ($c_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_8$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_9$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_{10}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\begin{align*}
\text{core 0} & \quad \text{core 1} \\
\text{core 2} & \quad \text{core 0}
\end{align*}$

$\begin{align*}
\sum 0 (c_1) & \quad \sum 1 (c_1) & \quad \sum 2 (c_1) \\
\sum 0 (c_8) & \quad \sum 1 (c_8) & \quad \sum 2 (c_8)
\end{align*}$

$\begin{align*}
\text{core 0} : & \quad b_1 \quad b_2 \quad b_3 \quad b_4 \\
\text{core 1} : & \quad b_5 \quad b_6 \quad b_7 \\
\text{core 2} : & \quad b_8 \quad b_9 \quad b_{10}
\end{align*}$

$\begin{align*}
b_1 & \quad b_2 \quad b_3 \quad b_4 \\
b_5 & \quad b_6 \quad b_7 \\
b_8 & \quad b_9 \quad b_{10}
\end{align*}$
MPI matrix-vector products

- (2) **Row-wise** block decomposition;

- Maintain **load balance**; each MPI task takes (almost) **same** number of rows;
- The same strategy as those of column-wise;
- How about vectors?
- Each MPI task has **entire** vectors $b$ and $c$;
- **Data communications** for vector $c$;

\[
\begin{align*}
\begin{bmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
c_6 \\
c_7 \\
c_8 \\
c_9 \\
c_{10}
\end{bmatrix}
& = \\
\begin{bmatrix}
\text{core 0} \\
\text{core 1} \\
\text{core 2}
\end{bmatrix}
\times
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
b_8 \\
b_9 \\
b_{10}
\end{bmatrix}
\end{align*}
\]
Benchmark an MPI Application

- Increasing **FLOPS per unit time** is one of our endless goals in the HPC community;
- Maintaining parallel **scalability**: how an MPI code behave with increasing numbers of cores or threads;
- Before we are able to benchmark an MPI application, be sure that the results are correct!
- **Strong scaling** and **weak scaling** from different perspectives of measurements;
- We are interested to spot this information in your allocation proposals!

- Rank 0 measures
  
  ```
  time_s = MPI_Wtime();
  time_e = MPI_Wtime();
  elapsed_time = time_e - time_s
  ```
  in second;
- **Average** wall-clock time or a **shortest** wall-clock time?
Benchmark an MPI Application

- Examples: run `mpi_matve_v1` and `v2` ($10^4$ times of $c = A \cdot b$);

![Graph showing speedup vs. number of cores for different values of n and version 1 of the application.]
Benchmark an MPI Application

- Examples: run `mpi_matve_v1` and `_v2` ($10^4$ times of $c = A \cdot b$);
Benchmark an MPI Application

- Examples: run `mpi_matve_v1` and `_v2` ($10^4$ times of $c = A \cdot b$);

- Performance really depends on the algorithms, problem sizes, etc;
Exercises 3

**Exercise 3**: Run `mpi_matvec_v1` for matrix sizes of 144, 576, 1200, and 1440, respectively. The number of MPI tasks is from 1 to 16. Benchmark the wall-clock time.

1. What is the max speedup you could get?
2. How would you explain the performance difference for small and large matrices?
**MPI matrix-vector products**

- **(3) 2D** domain decomposition (DD);
- 1D column- and row-wise decomposition are particular cases of 2D domain decomposition;

<table>
<thead>
<tr>
<th></th>
<th>core 0</th>
<th></th>
<th>core 1</th>
<th></th>
<th>core 2</th>
<th></th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>10</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>11</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>12</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>13</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>14</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td>15</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td>16</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td>17</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>18</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td>19</td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

- More MPI tasks in **2D** cases;
- Vectors are block striped (same as the column-wise case);
MPI matrix-vector products

- Generally, **2D** DD is much more complicated than 1D cases;
- (1) We only consider a **square** matrix times a vector;
- (2) Assume the number of MPI tasks is a **square** number;
- (3) Matrix size should be **dividable** by the number of MPI tasks along row (or column) dimension;
- Create a 2D **Cartesian** \((x, y)\) DD;

```c
1  MPI_Cart_create(MPI_COMM_WORLD, 2, dimes, bdperiodic, \   \   topology, &COMM2D);
2  MPI_Comm_rank(COMM2D, &my_id);
3  MPI_Cart_coords(COMM2D, my_id, 2, coords_2d);
4  mycoods_x = coords_2d[0];
5  mycoods_y = coords_2d[1];

C version 2
```

- A new communicator **COMM2D**;
- **bdperiodic** and **topology**;
### MPI matrix-vector products

- **2D Cartesian coordinates;**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
<td>(0,0)</td>
<td>(1,0)</td>
<td>(2,0)</td>
<td>(3,0)</td>
<td>(4,0)</td>
</tr>
</tbody>
</table>

- **Need to manipulate matrix elements in row or column patterns;**

- **Use** $x$ **or** $y$ **coordinates to map operation on MPI tasks;**

- **Analyze data communication for vectors** $b$ **and** $c$ **;**
MPI matrix-vector products

• 2D Cartesian coordinates;

- From the **left-most** cores to the **top-most** cores; then roll down for all **rows**;
MPI matrix-vector products

- 2D Cartesian coordinates;

- From the **left-most** cores to the **top-most** cores; then roll down for all rows;
MPI matrix-vector products

1. MPI_Comm_split(MPI_COMM_WORLD, mycoods_y, mycoods_x, 
   &COMM_ROW);
2. MPI_Comm_rank(COMM_ROW, &my_id_row);
3. MPI_Comm_split(MPI_COMM_WORLD, mycoods_x, mycoods_y, 
   &COMM_COL);
4. MPI_Comm_rank(COMM_COL, &my_id_col);

- How many COMM_ROW or COMM_COL do we have?
MPI matrix-vector products

1 MPI_Comm_split(MPI_COMM_WORLD, mycoods_y, mycoods_x, 
   &COMM_ROW);
2 MPI_Comm_rank(COMM_ROW, &my_id_row);
3 MPI_Comm_split(MPI_COMM_WORLD, mycoods_x, mycoods_y, 
   &COMM_COL);
4 MPI_Comm_rank(COMM_COL, &my_id_col);

• How many COMM_ROW or COMM_COL do we have?
MPI matrix-vector products

1. `MPI_Bcast(vector, chunk, MPI_DOUBLE, 0, COMM_COL);`
**MPI matrix-vector products**

```c
1  MPI_Bcast(vector,chunk,MPI_DOUBLE,0,COMM_COL);
```

- Should we use `COMM_ROW` or `COMM_COL`?
- Several small subsets of `MPI_WORLD_WORLD`;
- In this case, all subsets are named in the **same** way;
- All cores in the **same** column have the same chunk of the vector;
- Data communication within the **same subset**;
MPI matrix-vector products

- After local matrix-vector products, each MPI task has its own contribution to the final vector $c$;

```c
MPI_Reduce(c_local_temp,vector_out,chunk,
MPI_DOUBLE,MPI_SUM,0,COMM_ROW);
```

```c
if(mycoods_x == 0 && my_id != master_id) {
    my_id_trans = my_id * noblock_1d;
    MPI_Send(vector_inp,chunk,MPI_DOUBLE,
    my_id_trans,0,MPI_COMM_WORLD); }
else if( mycoods_y == 0 && my_id != master_id) {
    my_id_trans = my_id / noblock_1d;
    MPI_Recv(vector_inp,chunk,MPI_DOUBLE,
    my_id_trans,0,MPI_COMM_WORLD,&istatus); }

MPI_Bcast(vector_inp,chunk,MPI_DOUBLE,
0,COMM_COL);
```
Exercises 4 & 5

- **Exercise 4**: In the code `mpi_matvec_v2.c (.f90)`, `MPI_Reduce` was used, so MPI task with rank 0 gathered the final answer. Can we replace `MPI_Reduce` with `MPI_Allreduce`?

- **Exercise 5**: In the same code, the post data communication was done in the main program and was separated from the function (routine) of `matvec`. Make the other version (say, `mpi_matvec_v3`) in such a way that the post data communication is carried in the function of `matvec`;
Further Reading


Parallel Programming in C with MPI and OpenMP, M. J. Quinn (McGraw Hill, 2004).

Questions?

sys-help@loni.org