Distributed Memory Programming With MPI (4)

2014 Spring

Jinkyu Jeong
(jinkyu@skku.edu)
Roadmap

- Hello World in MPI program
- Basic APIs of MPI
- Example program
  - The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.
A PARALLEL SORTING ALGORITHM
### Parallelizing sorting

- **n** keys and **p** = comm sz processes.
- **n/p** keys assigned to each process.
- No restrictions on which keys are assigned to which processes.
- When the algorithm terminates:
  - The keys assigned to each process should be sorted in (say) increasing order.
  - If \( 0 \leq q < r < p \), then each key assigned to process **q** should be less than or equal to every key assigned to process **r**.
Serial bubble sort

- Bubble sort cannot be efficiently parallelized
  - Inner-loop parallelization
    - P0 compares and swaps $a[0]$ and $a[1]$
    - P1 compares and swaps $a[1]$ and $a[2]$
  - Outer-loop parallelization
    - After each loop, data stored in the array are changed

```c
void Bubble_sort(
    int a[] /* in/out */,
    int n /* in */
) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length - 1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
} /* Bubble_sort */
```
Odd-even transposition sort

- Even phases, compare swaps:
  \((a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots\)

- Odd phases, compare swaps:
  \((a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots\)

- This odd-even transposition sort can be parallelized
  - P0 compares and swaps \(a[0]\) and \(a[1]\)
  - P1 compares and swaps \(a[2]\) and \(a[3]\)
Example: odd-even transposition sort

Start:  5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)  
getting the list  5, 9, 3, 4

Odd phase: compare-swap (9,3)  
getting the list  5, 3, 9, 4

Even phase: compare-swap (5,3) and (9,4)  
getting the list  3, 5, 4, 9

Odd phase: compare-swap (5,4)  
getting the list  3, 4, 5, 9
Serial odd-even transposition sort

```c
void Odd_even_sort(
    int a[] /* in/out */,
    int n /* in */)
{
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        else /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
} /* Odd_even_sort */
```
Parallelizing Odd-even transposition sort

- For each phase, processes perform compare and swap of two numbers in parallel

- Even phases, compare swaps:

  \[
  \begin{align*}
  & (a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots \\
  & P0 \quad P1 \quad P1 \quad \ldots
  \end{align*}
  \]

- Odd phases, compare swaps:

  \[
  \begin{align*}
  & (a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots \\
  & P0 \quad P1 \quad P1 \quad \ldots
  \end{align*}
  \]
Parallelizing odd-even transposition sort - con’t

Start: 5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)
getting the list 5, 9, 3, 4

Odd phase: compare-swap (9,3)
getting the list 5, 3, 9, 4

Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

Odd phase: compare-swap (5,4)
getting the list 3, 4, 5, 9

P0 P1
Communications among tasks in odd-even sort

- For each phase, a process have to know whether two values to be compared have changed or not
- So, a process have to consult its sibling process before comparing two values
Parallel odd-even transposition sort

**Assumption**

- Each process performs local sorting for the given numbers
- Then, during the compare-swap phases, numbers in two processes are re-distributed based on the sorting order

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>15, 11, 9, 16</td>
<td>3, 14, 8, 7</td>
<td>4, 6, 12, 10</td>
<td>5, 2, 13, 1</td>
</tr>
<tr>
<td>After Local Sort</td>
<td>9, 11, 15, 16</td>
<td>3, 7, 8, 14</td>
<td>4, 6, 10, 12</td>
<td>1, 2, 5, 13</td>
</tr>
<tr>
<td>After Phase 0</td>
<td>3, 7, 8, 9</td>
<td>11, 14, 15, 16</td>
<td>1, 2, 4, 5</td>
<td>6, 10, 12, 13</td>
</tr>
<tr>
<td>After Phase 1</td>
<td>3, 7, 8, 9</td>
<td>1, 2, 4, 5</td>
<td>11, 14, 15, 16</td>
<td>6, 10, 12, 13</td>
</tr>
<tr>
<td>After Phase 2</td>
<td>1, 2, 3, 4</td>
<td>5, 7, 8, 9</td>
<td>6, 10, 11, 12</td>
<td>13, 14, 15, 16</td>
</tr>
<tr>
<td>After Phase 3</td>
<td>1, 2, 3, 4</td>
<td>5, 6, 7, 8</td>
<td>9, 10, 11, 12</td>
<td>13, 14, 15, 16</td>
</tr>
</tbody>
</table>
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I’m not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```c
if (phase % 2 == 0) /* Even phase */
    if (my_rank % 2 != 0) /* Odd rank */
        partner = my_rank - 1;
    else /* Even rank */
        partner = my_rank + 1;
else /* Odd phase */
    if (my_rank % 2 != 0) /* Odd rank */
        partner = my_rank + 1;
    else /* Even rank */
        partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL;
```
Safety in MPI programs

- The MPI standard allows MPI_Send to behave in two different ways:
  - it can simply copy the message into an MPI managed buffer and return,
  - or it can block until the matching call to MPI_Recv starts.

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.

- Relatively small messages will be buffered by MPI_Send.

- Larger messages, will cause it to block.
Safety in MPI programs

- If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.

- Each process is blocked waiting for an event that will never happen.

(see pseudo-code)
Safety in MPI programs

- A program that relies on MPI provided buffering is said to be *unsafe*.

- Such a program may run without problems for various sets of input, but it may hang or crash with other sets.
MPI_Ssend

- An alternative to MPI_Send defined by the MPI standard.
- The extra “s” stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts.

```c
int MPI_Ssend(
    void* msg_buf_p, /* in */,
    int msg_size, /* in */,
    MPI_Datatype msg_type, /* in */,
    int dest, /* in */,
    int tag, /* in */,
    MPI_Comm communicator /* in */);
```
Restructuring communication

```c
if (my_rank % 2 == 0) {
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
} else {  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```
MPI_Sendrecv

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- The dest and the source can be the same or different.
- Especially useful because MPI schedules the communications so that the program won’t hang or crash.

![Diagram showing the use of MPI_Sendrecv](image-url)
```c
int MPI_Sendrecv(
    void*     send_buf_p   /* in */ ,
    int       send_buf_size /* in */ ,
    MPI_Datatype send_buf_type /* in */ ,
    int       dest       /* in */ ,
    int       send_tag   /* in */ ,
    void*     recv_buf_p  /* out */ ,
    int       recv_buf_size /* in */ ,
    MPI_Datatype recv_buf_type /* in */ ,
    int       source     /* in */ ,
    int       recv_tag   /* in */ ,
    MPI_Comm  communicator /* in */ ,
    MPI_Status* status_p  /* in */);
```
Restructuring communication using MPI_Sendrecv

if (my_rank % 2 == 0) {
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
} else {
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}

MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0, recv_keys, n/comm_sz, MPI_INT, partner, 0, comm, MPI_Status_ignore);
Merging Numbers during Compare & Swap in a Process

```c
void Merge_low(
    int my_keys[],  /* in/out */
    int recv_keys[], /* in */
    int temp_keys[], /* scratch */
    int local_n  /* = n/p, in */)
{
    int m_i, r_i, t_i;

    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if (my_keys[m_i] <= recv_keys[r_i]) {
            temp_keys[t_i] = my_keys[m_i];
            t_i++; m_i++;
        } else {
            temp_keys[t_i] = recv_keys[r_i];
            t_i++; r_i++;
        }
    }

    for (m_i = 0; m_i < local_n; m_i++)
        my_keys[m_i] = temp_keys[m_i];
} /* Merge_low */
```
## Run-times of parallel odd-even sort

<table>
<thead>
<tr>
<th>Processes</th>
<th>Number of Keys (in thousands)</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
<th>3200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88</td>
<td>190</td>
<td>390</td>
<td>830</td>
<td>1800</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td>91</td>
<td>190</td>
<td>410</td>
<td>860</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>46</td>
<td>96</td>
<td>200</td>
<td>430</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>24</td>
<td>51</td>
<td>110</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>7.5</td>
<td>14</td>
<td>29</td>
<td>60</td>
<td>130</td>
<td></td>
</tr>
</tbody>
</table>

*(times are in milliseconds)*
Concluding Remarks

- **MPI or the Message-Passing Interface**
  - An interface of parallel programming in distributed memory system
  - Supports C, C++, and Fortran
  - Many MPI implementations
    - Ex. MPICH2

- **SPMD program**

- **Message passing**
  - Communicator
  - Point-to-point communication
  - Collective communication
  - Safe use of communication is important
    - Ex. MPI_Sendrecv()