Table of Contents

1 MPI: Misc. Topics
   • MPI: Derived Data Types
   • MPI: Parallel Sorting
   • MPI: Safety in Programs

2 Next Time
MPI DERIVED DATATYPES
Derived datatypes

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they’re received.
Derived datatypes

- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}
MPI_Type create struct

- Builds a derived datatype that consists of individual elements that have different basic types.

```c
int MPI_Type_create_struct(
    int count,
    int array_of_blocklengths[],
    int array_of_displacements[],
    MPI_Aint array_of_types[],
    MPI_Datatype* new_type_p
)
```
MPI_Get_address

- Returns the address of the memory location referenced by location_p.
- The special type MPI_Aint is an integer type that is big enough to store an address on the system.

```c
int MPI_Get_address(
    void* location_p  /* in */,
    MPI_Aint* address_p /* out */);
```
MPI_Type_commit

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```
MPI_Type_free

- When we’re finished with our new type, this frees any additional storage used.

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
Example from mpi-trap4.c

Get input function with a derived datatype (1)

```c
void Build_mpi_type(
    double* a_p, /* in */,
    double* b_p, /* in */,
    int* n_p, /* in */,
    MPI_Datatype* input_mpi_t_p /* out */) {

    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
```
Example from mpi-trap4.c

Get input function with a derived datatype (2)

```c
MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Type_create_struct(3, array_of_blocklengths,
                     array_of_displacements, array_of_types,
                     input mpi_t_p);
MPI_Type_commit(input mpi_t_p);
} /* Build_mpi_type */
```
Example from mpi-trap4.c

Get input function with a derived datatype (3)

```c
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {
    MPI_Datatype input_mpi_t;

    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```
A PARALLEL SORTING ALGORITHM
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 ≤ 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

```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

- A sequence of phases.
- Even phases, compare swaps:
  \[ (a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots \]
- Odd phases, compare swaps:
  \[ (a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots \]
Example

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 */
```
Communications among tasks in odd-even sort

Tasks determining $a[i]$ are labeled with $a[i]$. 
Parallel odd-even transposition sort

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>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>
Pseudo-code

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;
    }
}
Compute_partner

```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.
Safety in MPI programs

- 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

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);

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.
MPI_Sendrecv

```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 */);
```
Safe communication with five processes
Parallel odd-even transposition sort

```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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>7.5</td>
</tr>
</tbody>
</table>

(times are in milliseconds)
Next Time

- Next class: 10/16/14
- HW #3 Due: 10/23/14
- Quiz 2 (MPI) and Quiz 3 (Pthreads) will be held on 11/04/14.
- Quiz 2: two parts:
  - Part 1: take home assignment on 10/30/14, due 11/04/14
    [was 10/23/14, due 10/28] (75% of the grade)
  - Part 2: in class quiz on 11/04/14 [was 10/28] (25% of the grade)
- Start Pthreads on 10/15/14