A PARALLEL SORTING
ALGORITHM
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 */
```

Serial Odd-Even Transposition Sort

---

In the C code, phase 1 is phase 0.
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 */
```

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;
    }
}
Sort 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;
```

**Sorting**

- $n$ keys and $p = \text{comm sz}$ 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 $r$. 
Parallel odd-even transposition sort

<table>
<thead>
<tr>
<th>Time</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Start</td>
<td>15, 11, 9, 16</td>
</tr>
<tr>
<td>After Local Sort</td>
<td>9, 11, 15, 16</td>
</tr>
<tr>
<td>After Phase 0</td>
<td>3, 7, 8, 9</td>
</tr>
<tr>
<td>After Phase 1</td>
<td>3, 7, 8, 9</td>
</tr>
<tr>
<td>After Phase 2</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>After Phase 3</td>
<td>1, 2, 3, 4</td>
</tr>
</tbody>
</table>

\[ n = 16, \ p = 4 \]

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.
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);
}

"Ring pass"
It is safe if comm_sz is even

Safe communication with five processes

Time 0

Time 1

Time 2
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.

```
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 */);
```
Parallel odd-even transposition sort

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)
Concluding Remarks (1)

• MPI or the Message-Passing Interface is a library of functions that can be called from C, C++, or Fortran programs.
• A communicator is a collection of processes that can send messages to each other.
• Many parallel programs use the single-program multiple data or SPMD approach.

Concluding Remarks (2)

• Most serial programs are deterministic: if we run the same program with the same input we’ll get the same output.
• Parallel programs often don’t possess this property.
• Collective communications involve all the processes in a communicator.
Concluding Remarks (3)

- When we time parallel programs, we’re usually interested in elapsed time or “wall clock time”.
- Speedup is the ratio of the serial run-time to the parallel run-time.
- Efficiency is the speedup divided by the number of parallel processes.

Concluding Remarks (4)

- If it’s possible to increase the problem size (n) so that the efficiency doesn’t decrease as p is increased, a parallel program is said to be scalable.
- An MPI program is unsafe if its correct behavior depends on the fact that MPI_Send is buffering its input.