CSCI-UA.0480-003

Parallel Computing

Lecture 10: MPI - IV

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adopted and slightly modified from:

- Gerassimos Barlas
- Peter S. Pacheco

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 $0 \le q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

Serial bubble sort

```
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 */
```

$O(n^2)$

How can we parallelize this?

The problem with bubble-sort

- We cannot execute the compare-swap out-of-order!
- Can we decouple that?

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

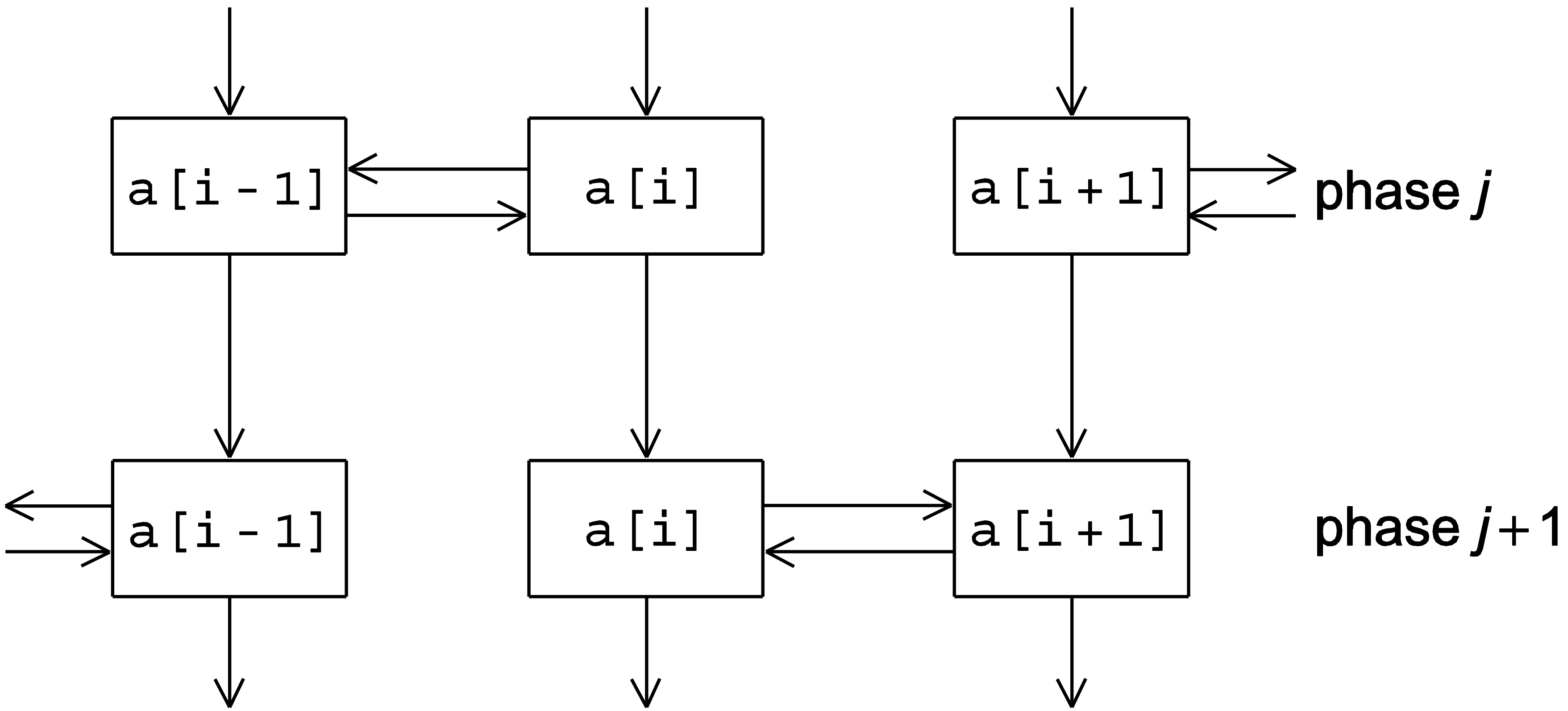
```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   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 */
```

Even phase

Odd phase

Communications among tasks in odd-even sort



Parallel odd-even transposition sort

Assume P processors (=4) and list n (=16) numbers

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

phase 0 and phase 2

- Processes (0,1) exchange their elements
- Processes (2, 3) exchange their elements
- Processes 0 and 2 keep the smallest 4
- Processes 1 and 3 keep the largest 4

phase 1 and phase 3

- Processes (1, 2) exchange their elements
- Process 1 keeps smallest 4 and process 2 keeps largest 4

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

```
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;
```

- Constant defined by MPI
- When used as source/destination in point-to-point comm, no comm will take place.

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.

So … What can we do?

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.

```
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 */);
```

How does MPI_Ssend help?

- Replace all MPI_Send calls in your code with MPI_Ssend
- If your program does not hang or crash → the original program is safe
- What do we do if we find out that our program is not safe?
- The main problem is due to the fact that all processes send then receive… Let's change that!

Restructuring communication

original

```
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.
```

Updated version

```
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);
}
```

Note: The above two versions show a ring communication
(i.e. processor comm_sz-1 sends to process 0.)

MPI_Sendrecv

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- Especially useful because MPI schedules the communications so that the program won't hang or crash.
- Replaces a pair of consecutive send and receive calls.

MPI_Sendrecv

```
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   */);
```

MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace{
        void *            buf_p,
        int               buf_size,
        MPI_Datatype      buf_type,
        int               dest,
        int               send_tag,
        int               recv_tag,
        MPI_Comm          communicator,
        MPI_Status *      status_p };
```

In this case, what is in buf_p will be sent and replaced by what is received.

Back to our 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;
   }
}
```

How will you implement this?

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 */
```

At the end, my_keys[] will have the smallest n/p keys of local and received keys

Run-times of parallel odd-even sort

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

(times are in milliseconds)

Run-times of parallel odd-even sort (Larger problem size)

comm_sz	Run-Times (in seconds)				
	Number of Elements				
	1M	2M	4M	8M	16M
1	4.10E-02	8.73E-02	2.22E+00	4.65E+00	9.69E+00
2	2.04E-02	4.32E-02	1.10E+00	2.31E+00	4.81E+00
4	1.10E-02	2.24E-02	5.65E-01	1.18E+00	2.46E+00
8	6.73E-03	1.25E-02	2.98E-01	6.22E-01	1.29E+00
16	5.19E-03	8.45E-03	1.70E-01	3.53E-01	7.31E-01

Run-times of parallel odd-even sort (Larger problem size)

comm_sz	Speedups				
	Number of Elements				
	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	2.01	2.02	2.02	2.02	2.02
4	3.73	3.90	3.93	3.94	3.93
8	6.09	6.98	7.46	7.48	7.50
16	7.89	10.34	13.11	13.19	13.26

comm_sz	Efficiencies				
	Number of Elements				
	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	1.00	1.01	1.01	1.01	1.01
4	0.93	0.97	0.98	0.98	0.98
8	0.76	0.87	0.93	0.93	0.94
16	0.49	0.65	0.82	0.82	0.83

PREFIX-SUM

What is that?

- Generalization of global sum
- Input: vector x[] of n elements
- Output: vector prefix_sum[] of n elements, such that:
  - prefix_sum[0] = x[0]
  - prefix_sum[1] = x[0] + x[1]
  - prefix_sum[2] = x[0] + x[1] + x[2]
  - prefix_sum[n-1] = x[0] + x[1] + … + x[n-1]

```
/* First compute prefix sums of my local vector */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
   loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

if (my_rank != 0) {
   /* If I'm not 0 receive sum of preceding components */
   MPI_Recv(&sum_of_preceding, 1, MPI_DOUBLE, my_rank-1, 0, comm,
         MPI_STATUS_IGNORE);

   /* Add in sum of preceding components to my prefix sums */
   for (loc_i = 0; loc_i < n; loc_i++)
      loc_prefix_sums[loc_i] += sum_of_preceding;
}

/* Now send my last element to the next process */
if (my_rank != comm_sz - 1)
   MPI_Send(&loc_prefix_sums[loc_n-1], 1, MPI_DOUBLE, my_rank+1, 0, comm);
```

MPI_Scan

```
int MPI_Scan(
        void *          sendbuf,
        void *          recvbuf,
        int             count,
        MPI_Datatype    datatype,
        MPI_Op          op,
        MPI_Comm        comm);
```

MPI_BAND
Bitwise AND
MPI_BOR
Bitwise OR
MPI_BXOR
Bitwise XOR
MPI_LAND
Logical AND
MPI_LOR
Logical OR
MPI_LXOR
Logical XOR
MPI_MAX
Maximum value
MPI_MAXLOC
Maximum value and location
MPI_MIN
Minimum value
MPI_MINLOC
Minimum value and location
MPI_PROD
Product
MPI_SUM
Sum

Returns for process of rank i,
the prefix reduction values for elements 0 … i

Be Careful

Assume 4 processes:

[1 2 3] [4 5 6] [7 8 9] [10 11 12]

The output of MPI_Scan, for MPI_SUM, is:

[1 2 3] [5 7 9] [12 15 18] [22 26 30]

NOT:

[1 3 6] [10 15 21] [28 36 45] [55 66 78]

Conclusions

- Choosing the correct algorithm for a problem to be solved by MPI depends on:
  - Opportunities of parallelization
  - The complexity