

Parallel Programming

Distributed Memory Programming with MPI (3)

Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.

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 efficiently.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

Derived datatypes

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

Variable	Address
a	24
b	40
n	48

$\{(\text{MPI_DOUBLE}, 0), (\text{MPI_DOUBLE}, 16), (\text{MPI_INT}, 24)\}$

MPI_Type create_struct

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

```
int MPI_Type_create_struct(  
    int          count          /* in */,  
    int          array_of_blocklengths[] /* in */,  
    MPI_Aint     array_of_displacements[] /* in */,  
    MPI_Datatype array_of_types[] /* in */,  
    MPI_Datatype* new_type_p     /* out */);
```

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.

```
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.

```
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.

```
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```

Get input function with a derived datatype (1)

```
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};
```

Get input function with a derived datatype (2)

```
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;
array_of_displacements[2] = n_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 */
```

Get input function with a derived datatype (3)

```
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);

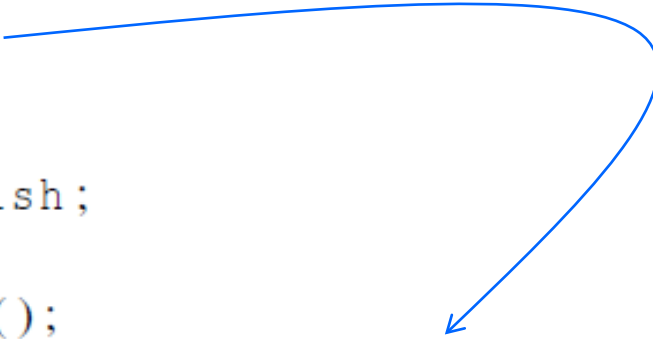
    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```

PERFORMANCE EVALUATION

Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);  
  
double start, finish;  
...  
start = MPI_Wtime();  
/* Code to be timed */  
...  
finish = MPI_Wtime();  
printf("Proc %d > Elapsed time = %e seconds\n"  
       my_rank, finish-start);
```



Elapsed serial time

- In this case, you don't need to link in the MPI libraries. The POSIX library function *gettimeofday* returns time in microseconds elapsed from some point in the past.
- Pacheco book example code provides a GET_TIME macro, which records the number of seconds since some time in the past.

```
#include "timer.h"  
.  
.  
.  
double now;  
.  
.  
.  
GET_TIME(now);
```



Elapsed serial time

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```


MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```
int MPI_Barrier(MPI_Comm comm /* in */);
```



MPI_Barrier

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
          MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
```

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Speedup

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Efficiency

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}$$

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Scalability

- A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

Scalability

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**.
- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.

A PARALLEL SORTING ALGORITHM

Sorting

- n keys and $p = \# \text{ processes}$.
- 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 order (we will use increasing order in the example).
 - 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

```
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:

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$$

- Odd phases, compare swaps:

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$$

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

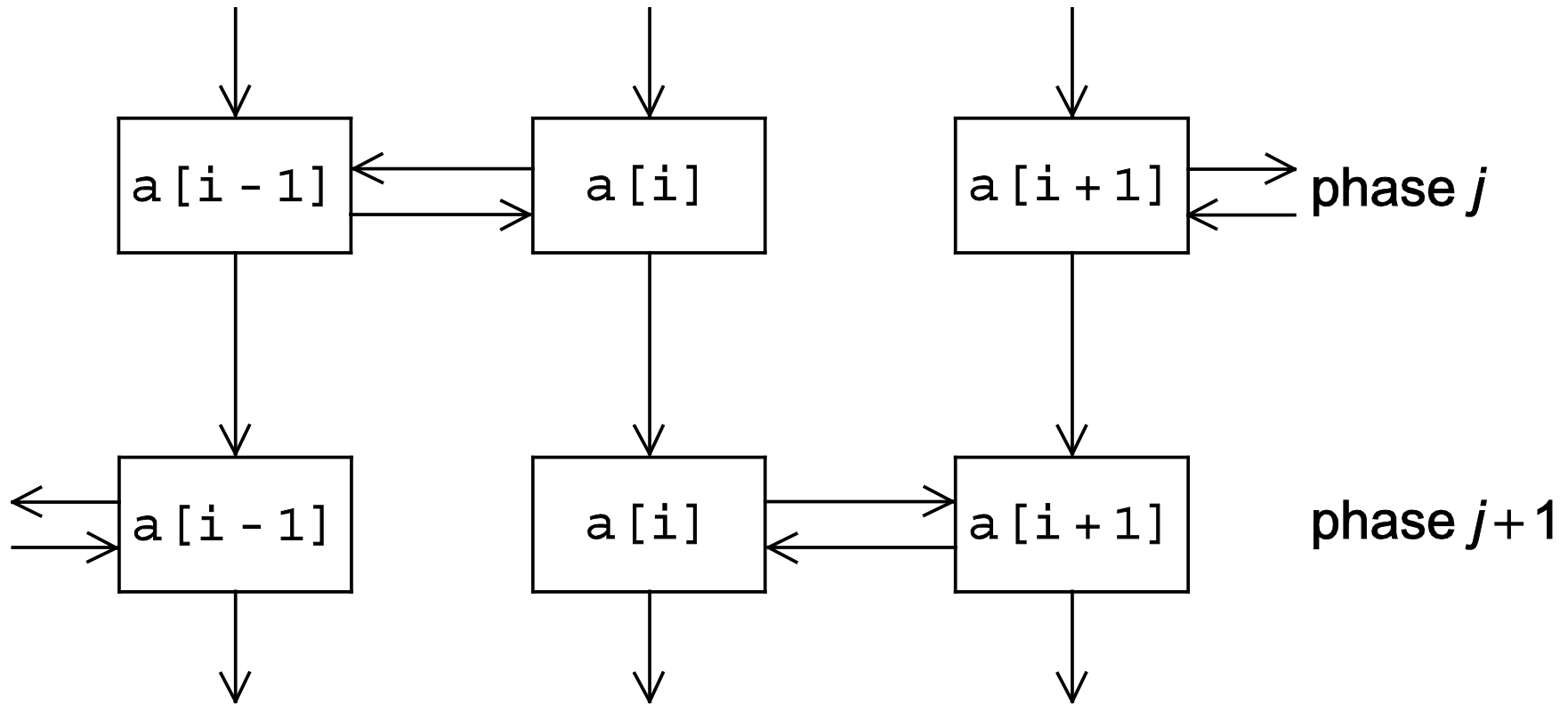
Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

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 */
```

Communications among tasks in odd-even sort



Tasks determining $a[i]$ are labeled with $a[i]$.

Parallel odd-even transposition sort

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

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;
```

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.

```
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);  
}
```

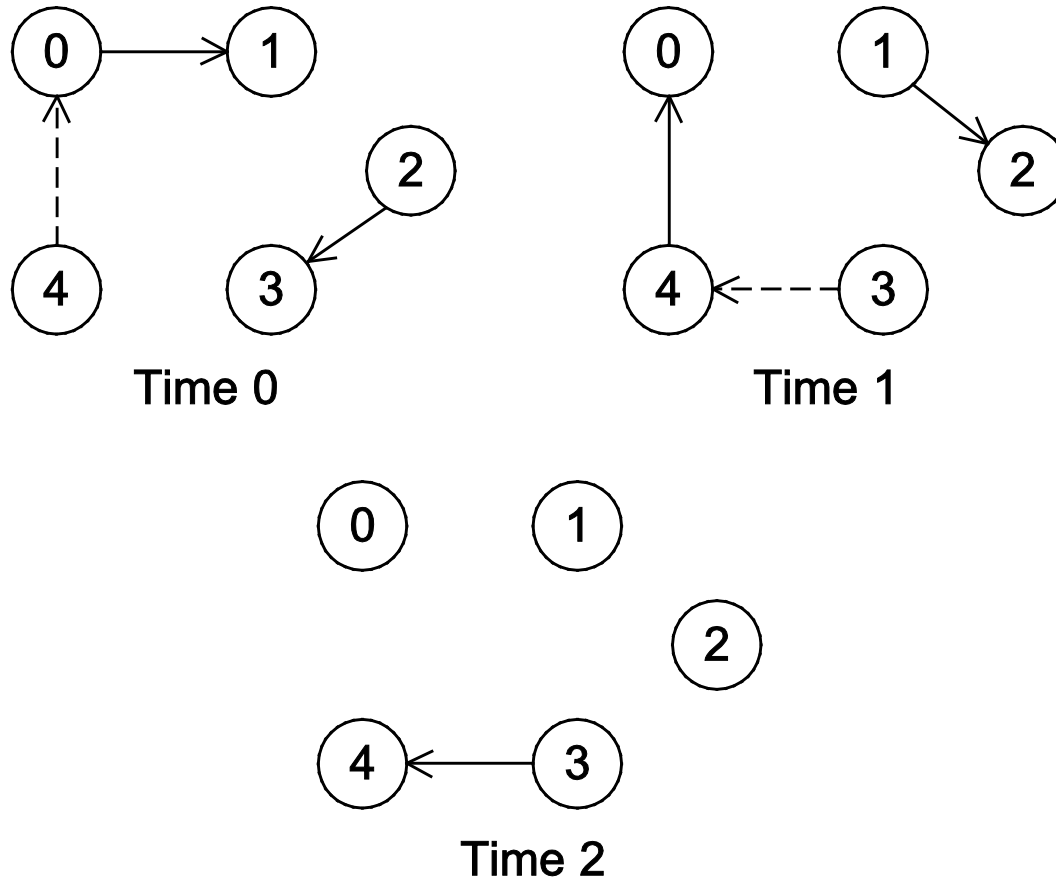
MPI_Sendrecv

- An alternative to scheduling the communications by ourselves.
- Carries out a blocking send and a receive in a single call.
- The destination 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

```
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

```
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-time 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

(time is 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.