BITS Pilani
Pilani Campus

Message Passing Interface (MPI)

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Programming with MPI: Derived Data Types

Sending Data in a Single Message

- Sending a, b and n in one message will take lesser time compared to sending a, b and n individually
- MPI provides three basic approaches to consolidate data
  - Send in an array using count argument if data is of same type
  - Derived data types
  - MPI_Pack/MPI_Unpack are used to pack data into a contiguous buffer space
    - The pack/unpack routines are provided for compatibility with previous libraries. Derived data types is offered in later versions of MPI

```
if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a,b,n);
for (dest = 1; dest < comm_sz; dest++) {
        MPI_Send(a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        MPI_Send(b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        MPI_Send(n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
}
} else { /* my rank != 0 */
        MPI_Recv(a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS IGNORE);
        MPI_Recv(n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS IGNORE);
}
```

Create a New Data Type

BITS Pilani

- In MPI, a derived datatype can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory
- In our trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses
  - Then the following derived datatype could represent these data items
  - {(MPI_DOUBLE,0),(MPI_DOUBLE,16),(MPI_INT,24)}
    - The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the beginning of the type

Variable	Address
a	24
b	40
n	48

Create a New Data Type


```
int MPI_Type_create_struct(
  int count, //no of elements in the struct
  int array_of_blocklengths[],
  MPI_Aint array_of_displacements[],
  MPI_Datatype array_of_types[],
  MPI_Datatype *newtype
);
```

- We can use MPI_Type_create_struct to build a derived datatype that consists of individual elements that have different basic types
- The argument count is the number of elements in the datatype
  - for our example, it should be three
- Each of the next 3 array arguments should have count elements
- The first array, array of block lengths, allows for the possibility that the individual data items might be arrays or subarrays
  - for example, the first element were an array containing five elements, we would have array_of_blocklengths[0] = 5;
  - in our case array_of_blocklengths[3] = {1, 1, 1};

Create a New Data Type

```
int MPI_Type_create_struct(
  int count, //no of elements in the struct
  int array_of_blocklengths[],
  MPI_Aint array_of_displacements[],
  MPI_Datatype array_of_types[],
  MPI_Datatype *newtype
);
```

array_of_blocklengths allows for the possibility that the individual data items might be arrays or subarrays.

- array_of_displacements, specifies the displacements, in bytes, from the start of the message
  - array_of_displacements[] = {0, 16, 24};
- We can use MPI_Get_address to calculate displacements

```
int MPI_Get_address(
void* location_p /* in */,
MPI_Aint* address_p /* out */);
/*returns address of the memory
 location referenced by location_p*/
```

```
MPI_Aint a_addr, b_addr, n_addr;
MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr − a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr − a_addr;
```

Create a New Data Type

BITS Pilani

```
int MPI_Type_create_struct(
  int count, //no of elements in the struct
  int array_of_blocklengths[],
  MPI_Aint array_of_displacements[],
  MPI_Datatype array_of_types[],
  MPI_Datatype *newtype
);
```

- The array_of_types should store the MPI datatypes of the elements
  - array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT}
- With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
MPI_Type_create_struct(3, array_of_blocklengths,
array_of_displacements, array_of_types,
&input_mpi_t);
```

- Before we can use input_mpi_t in a communication function, we must first commit it with a call to

```
int MPI_Type_commit(MPI_Datatype* new_mpi_t p /*in/out*/);
```

Using Derived Data Type

BITS Pilani

- Now, in order to use new_mpi_t, we make the following call to MPI_Bcast on each process:
- MPI_Bcast(&a, 1, input_mpi_t, 0, comm);

Example

```c
/*-------------------------------------------------------------------
 * Function:     Build_mpi_type
 * Purpose:      Build a derived datatype so that the three
 *               input values can be sent in a single message.
 * Input args:   a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 * Output args:  input_mpi_t_p:  the new MPI datatype
 */
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};

   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 */
```

Example

BITS Pilani

```c
void Get_input(
        int      my_rank  /* in  */,
        int      comm_sz  /* in  */,
        double*  a_p      /* out */,
        double*  b_p      /* out */,
        int*     n_p      /* out */) {
    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 */
```

Taking timings

- We’re usually not interested in the time taken from the start of program execution to the end of program execution

```c
    MPI_Barrier(comm);
    start = MPI_Wtime();
    Mat_vect_mult(local_A, local_x, local_y, local_m, n, local_n, comm);
    finish = MPI_Wtime();
    loc_elapsed = finish-start;
    MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);

#   ifdef DEBUG
    Print_vector("y", local_y, m, local_m, my_rank, comm);
#   endif

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

- Time taken by parallel algorithm is the time of the slowest process. We can’t get exactly this time because we can’t insure that all the processes start at the same instant. However, we can come reasonably close. The MPI collective communication function MPI Barrier insures that no process will return from calling it until every process in the communicator has started calling it.

Odd-even Transposition Sort

- Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but that has more opportunities for parallelism

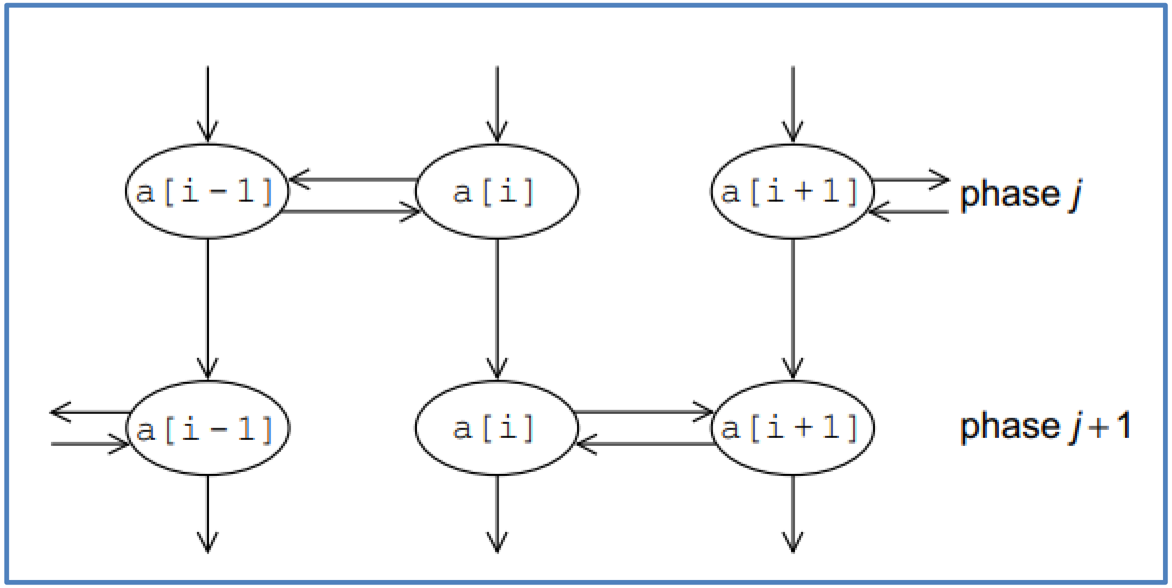
```c
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   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 */
```

Suppose a = {9, 7, 8, 6}.

	Subscript in Array						
Phase	**0**		**1**		**2**		**3**
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Parallel odd-even transposition sort



- When n = p: depending on the phase, process i can send its current value, a[i], either to process i − 1 or process i + 1. At the same time, it should receive the value stored on process i − 1 or process i + 1, respectively, and then decide which of the two values it should store as a[i] for the next phase.

Parallel odd-even transposition sort

- It's unlikely that n = p
- How should this be modified when each process is storing n/p > 1 elements?

At any time, each process should have exactly n/p elements.

```
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;
    }
}
```

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

Parallel odd-even transposition sort

- how do we compute the partner rank?

```
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;
```

MPI_PROC_NULL is a constant defined by MPI. When it's used as the source or destination rank in a point-to-point communication, no communication will take place and the call to the communication will simply return.

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

Parallel odd-even transposition sort

- how do we exchange elements?
  - If a process is not idle, we might try to implement the communication with a call to MPI_Send and a call to MPI_Recv

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);
```

  - This might result in deadlock.
  - 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
    - Furthermore, many implementations of MPI set a threshold at which the system switches from buffering to blocking. That is, messages that are relatively small will be buffered by MPI Send, but for larger messages, it will block.
  - If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv leading to deadlock

Parallel odd-even transposition sort

- 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. If we use MPI_Send and MPI_Recv in this way, our program will be unsafe, and it's likely that for small values of n the program will run without problems, while for larger values of n, it's likely that it will hang or crash
  - we can use an alternative to MPI_Send i.e. MPI_Ssend
    - MPI_Ssend is guaranteed to block until the matching receive starts

Parallel odd-even transposition sort

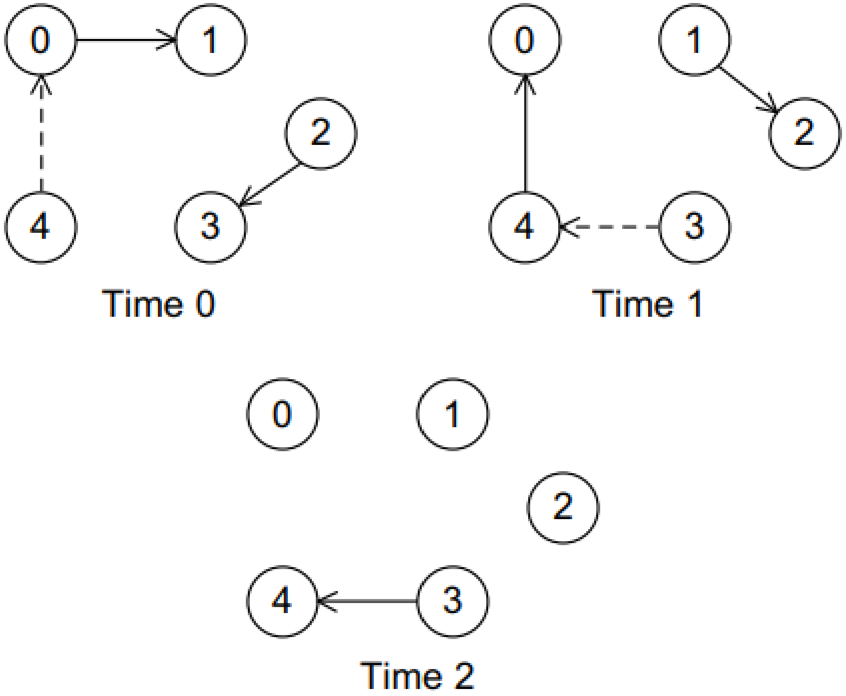
- How can we modify the communication in the parallel odd-even sort program so that it is safe?
  - Communication must be restructured. The most common cause of an unsafe program is multiple processes simultaneously first sending to each other and then receiving
  - need to restructure the communications so that some of the processes receive before sending

```
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 MPI_STATUS_IGNORE 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);
}
```

  - this will work if comm_sz is even. But when it is odd?

Parallel odd-even transposition sort

- For 5 processes, dashed arrows shows waiting communications though expected to be complete in the same phase.



Parallel odd-even transposition sort

- MPI provides an alternative to scheduling the communications ourselves— 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 */
);
```

- This function carries out a blocking send and a receive in a single call. The dest and the source can be the same or different.
- MPI implementation schedules the communications so that the program won't hang or crash.

Parallel odd-even transposition sort

```c
 * Purpose:      One iteration of Odd-even transposition sort
 * In args:      local_n, phase, my_rank, p, comm
 * In/out args: local_A
 * Scratch:      temp_B, temp_C
 */
void Odd_even_iter(int local_A[], int temp_B[], int temp_C[],
        int local_n, int phase, int even_partner, int odd_partner,
        int my_rank, int p, MPI_Comm comm) {
   MPI_Status status;

   if (phase % 2 == 0) {
      if (even_partner >= 0) {
         MPI_Sendrecv(local_A, local_n, MPI_INT, even_partner, 0,
            temp_B, local_n, MPI_INT, even_partner, 0, comm,
            &status);
         if (my_rank % 2 != 0)
            Merge_high(local_A, temp_B, temp_C, local_n);
         else
            Merge_low(local_A, temp_B, temp_C, local_n);
      }
   } else { /* odd phase */
      if (odd_partner >= 0) {
         MPI_Sendrecv(local_A, local_n, MPI_INT, odd_partner, 0,
            temp_B, local_n, MPI_INT, odd_partner, 0, comm,
            &status);
         if (my_rank % 2 != 0)
            Merge_low(local_A, temp_B, temp_C, local_n);
         else
            Merge_high(local_A, temp_B, temp_C, local_n);
      }
   }
```

References

BITS Pilani

- https://computing.llnl.gov/tutorials/mpi/
- "An introduction to Parallel Programming", Peter S. Pacheco (Chapter 3)

Thank You