ECE 432/532
Programming for Parallel Processors

MPI Derived Datatypes

- In virtually all distributed-memory systems, communication can be *much* more expensive than local computation.
- For example, sending a **double** from one node to another will take far longer than adding two **double**s stored in the local memory of a node.
- The cost of sending a fixed amount of data in multiple messages is much greater than the cost of sending a single message with the same amount of data.

MPI Derived Datatypes

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

VS

```
if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

MPI Derived Datatypes

50 times longer !

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

VS

```
if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

MPI Derived Datatypes

- 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.

- The idea is that if a function that sends data knows the types and the relative locations in memory of 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.

MPI Derived Datatypes

- Example: in the trapezoidal rule program we needed to call MPI_Bcast three times:
  - once for the left endpoint *a*
  - once for the right endpoint *b*
  - and once for the number of trapezoids *n*.

- As an alternative, we could build a single derived datatype that consists of two **double**s and one **int**.
  - need one call to MPI_Bcast

- On process 0, *a*,*b*, and *n* will be sent with the one call, while on the other processes, the values will be received with the call.

MPI Derived Datatypes

- Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes.

- In the trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses

Variable	Address
a	24
b	40
n	48

$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$

MPI_Type_create_struct

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

```
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_Type_create_struct

```
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  */);
```

- The argument count is the number of elements in the datatype → three (for trapezoidal rule)
- The array_of_block_lengths, allows for the possibility that the individual data items might be arrays or subarrays
  - first element is an array containing five elements → `array_of_blocklengths[0] = 5;`
  - In our case → `int array_of_blocklengths[3] = {1, 1, 1};`

MPI_Type_create_struct

```
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  */);
```

- The third argument, array_of_displacements specifies the displacements, in bytes, from the start of the message:

```
array_of_displacements[] = {0, 16, 24};
```

- The array_of_datatypes stores the MPI datatypes of the elements

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

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_Datatype input_mpi_t;

int array_of_blocklengths[3] = {1, 1, 1};

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;

array_of_displacements[] = {0, 16, 24};

MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};


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

MPI_Type_commit

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

Performance evaluation

- We write parallel programs because we expect that they'll be faster than a serial program that solves the same problem.
  - How can we verify this?

- We're not interested in the time taken from the start of program execution to the end of program execution.
  - E.g. the time it takes to type in the matrix or print out the product

- We're only interested in the time it takes to do the actual operation

Performance evaluation

- MPI provides a function, MPI_Wtime, that returns the number of seconds that have elapsed since some time in the past:

```
double MPI_Wtime(void);
```

Elapsed parallel time

```
double MPI_Wtime(void);
```

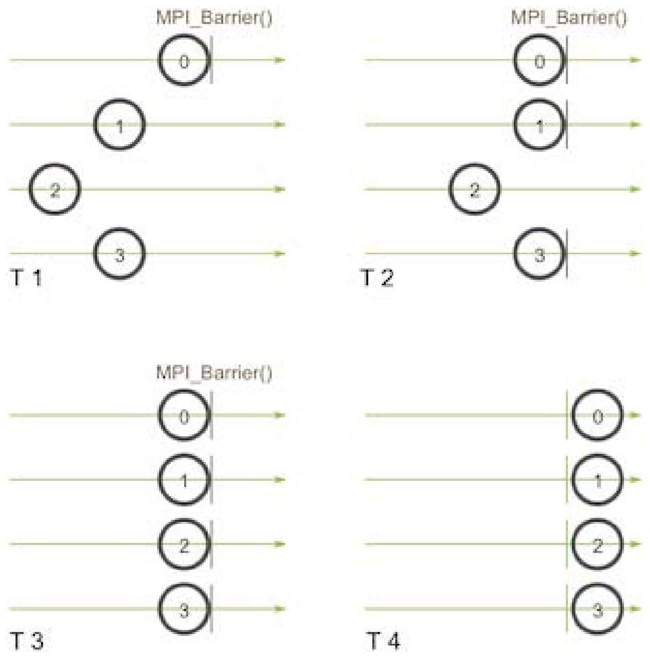
- Returns the number of seconds that have elapsed since some time in the past.

```
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);
```

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 */);
```



Elapsed parallel time

```
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);
```

```
int myrank, numprocs;
double mytime, maxtime, mintime, avgtime; /*variables used for gathering timing statistics*/

MPI_Comm_rank(MPI_COMM_WORLD, &myrank); MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Barrier(MPI_COMM_WORLD); /*synchronize all processes*/

mytime = MPI_Wtime(); /*get time just before work section */
work();
mytime = MPI_Wtime() - mytime; /*get time just after work section*/ /*compute max, min, and
average timing statistics*/

MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE,MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0,MPI_COMM_WORLD);
MPI_Reduce(&mytime, &avgtime, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

if (myrank == 0)  {
       avgtime /= numprocs;
       printf("Min: %lf Max: %lf Avg: %lf\n", mintime, maxtime,avgtime);
 }
```

Elapsed serial time

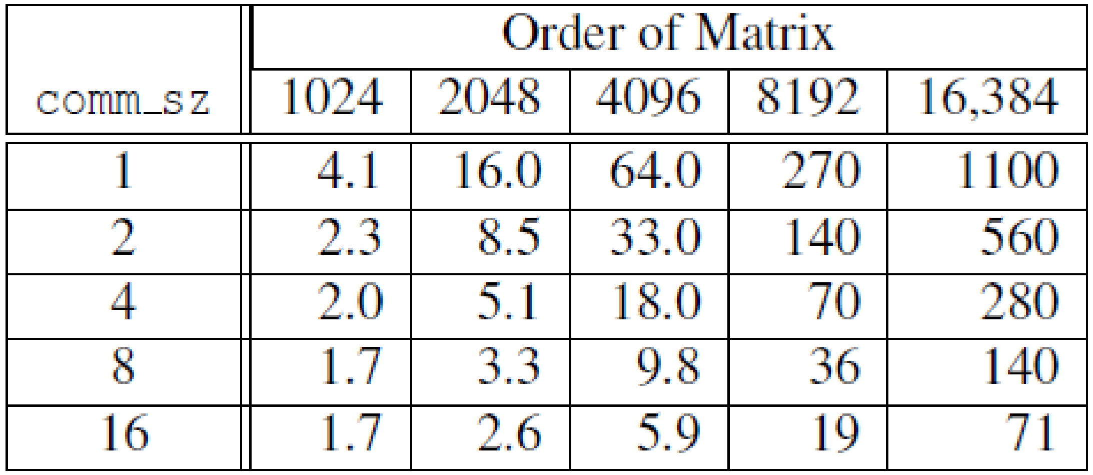
- In this case, you don’t need to link in the MPI libraries.
- Returns time in microseconds elapsed from some point in the past.

```
#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);
```

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

- If we fix *n* and increase comm_sz, the run-times usually decrease
- For small *n*, there is very little benefit in increasing comm_sz
- As we increase the problem size, the run-times increase
- As we increase the number of processes, the run-times typically decrease for a while. However, at some point, the run-times can actually start to get **worse** → Why?

Speedup and efficiency

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

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

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

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.