CSC510
Parallel Programming

Distributed-Memory Programming
(MPI - Part 3)

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

Derived datatypes

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(MPI\_DOUBLE,0), (MPI\_DOUBLE,16), (MPI\_INT,24)\}
MPI_Type create_struct

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

```c
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.

```c
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.

```c
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.

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
Get input function with a derived datatype (1)

```c
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)

```c
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;
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)

```c
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 */
```
**Elapsed parallel time**

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

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

```c
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.
- Returns time in microseconds elapsed from some point in the past.

```c
#include "timer.h"
...
double now;
...
GET_TIME(now);
```
Elapsed serial time

```c
#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.

```c
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

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>Order of Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1024 2048 4096 8192 16384</td>
</tr>
<tr>
<td>1</td>
<td>4.1   16.0 64.0 270 1100</td>
</tr>
<tr>
<td>2</td>
<td>2.3   8.5 33.0 140  560</td>
</tr>
<tr>
<td>4</td>
<td>2.0   5.1 18.0  70  280</td>
</tr>
<tr>
<td>8</td>
<td>1.7   3.3  9.8  36  140</td>
</tr>
<tr>
<td>16</td>
<td>1.7   2.6  5.9  19   71</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.1</td>
<td>3.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>4.8</td>
<td>6.5</td>
<td>7.5</td>
<td>7.9</td>
</tr>
<tr>
<td>16</td>
<td>2.4</td>
<td>6.2</td>
<td>10.8</td>
<td>14.2</td>
<td>15.5</td>
</tr>
</tbody>
</table>

### Efficiencies of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.94</td>
<td>0.97</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
<td>0.78</td>
<td>0.89</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
<td>0.61</td>
<td>0.82</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>16</td>
<td>0.15</td>
<td>0.39</td>
<td>0.68</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
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.

• 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**.