COMP/CS 605: Introduction to Parallel Computing
Topic : MPI: Derived Datatypes

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 02/20/17
Updated: 02/21/17

Table of Contents

1 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 theyre 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:

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,
    const int array_of_blocklengths[],
    const MPI_Aint array_of_displacements[],
    const MPI_Datatype array_of_types[],
    MPI_Datatype *newtype    )
```

Input Parameters:

- count: number of blocks (integer); also number of entries in arrays *array_of_types*, *array_of_displacements* and *array_of_blocklengths*
- *array_of_blocklengths*: number of elements in each block (array of integer)
- *array_of_displacements*: byte displacement of each block (array of address integer)
- *array_of_types*: type of elements in each block (array of handles to datatype objects)

Output Parameters:

- newtype: new datatype (handle)

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(
    const void *location
    MPI_Aint *address
)
```

Input Parameters:

- location in caller memory (choice)

Output Parameters:

- address: address of location (address integer)

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

```
int MPI_Type_commit(
     MPI_Datatype *datatype    )
)
```

Input Parameters:

- datatype: datatype (handle)

MPI_Type_free: Fees any storage used for this datatype.

```
int MPI_Type_free(
     MPI_Datatype *datatype    )
)
```

Input Parameters:

- datatype: datatype that is freed (handle)

Datatype Example: Pacheco code: mpi-trap4.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_Aint array_of_displacements[3] = {0};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;

   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 */
```

Datatype Example: Pacheco code: mpi-trap4.c

```
/*---------------------------------------------------------------
 * Function:     Get_input
 * Purpose:      Get the user input:  the left and right endpoints
 *               and the number of trapezoids
 * Input args:   my_rank:  process rank in MPI_COMM_WORLD
 *               comm_sz:  number of processes in MPI_COMM_WORLD
 * Output args:  a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 */
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 */
```

MPI_Bcast: Broadcasts a message from the process with rank "root" to all other processes of the communicator

```
int MPI_Bcast( void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm )
```

Input/Output Parameters:

- buffer: starting address of buffer (choice)

Input Parameters:

- count: number of entries in buffer (integer)
- datatype: data type of buffer (handle)
- root: rank of broadcast root (integer)
- comm: communicator (handle)

Datatype Example: Pacheco code: mpi-trap4.c

```
/* File:     mpi_trap4.c
 * Purpose:  Use MPI to implement a parallel version of the trapezoidal rule.
 *        This version uses collective communications and    MPI derived datatypes
 *        to distribute the input data and  compute the global sum.
 *
 * Input:    The endpoints of the interval of integration and the number  of trapezoids
 * Output:   Estimate of the integral from a to b of f(x) using the trapezoidal rule and n trapezoids.
 *
 * Compile:  mpicc -g -Wall -o mpi_trap4 mpi_trap4.c
 * Run:      mpiexec -n <number of processes> ./mpi_trap4
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of integration.
 *    2.  Each process estimates the integral of f(x) over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from  the individual processes and prints the result.
 *
 * Note:  f(x) is all hardwired.
 * IPP:   Section 3.5 (pp. 117 and ff.)
  */
#include <stdio.h>
#include <mpi.h>     /* MPI routines, definitions, etc. */

/* Build a derived datatype for distributing the input data */
void Build_mpi_type(double* a_p, double* b_p, int* n_p, MPI_Datatype* input_mpi_t_p);

/* Get the input values */
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p);

/* Calculate local integral  */
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len);

/* Function we're integrating */
double f(double x);
```

Datatype Example: Pacheco code: mpi-trap4.c

```
int main(void) {
   int my_rank, comm_sz, n, local_n;
   double a, b, h, local_a, local_b;
   double local_int, total_int;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);

   /* Get my process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   Get_input(my_rank, comm_sz, &a, &b, &n);

   h = (b-a)/n;          /* h is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* Length of each process' interval of integration = local_n*h.  So my interval starts at: */
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   /* Add up the integrals calculated by each process */
   MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */
```

MPI_Reduce: Reduces values on all processes to a single value

```
int MPI_Reduce(
        const void *sendbuf,
        void *recvbuf,
        int count,
        MPI_Datatype datatype,
        MPI_Op op,
        int root,
        MPI_Comm comm )
```

Input/Output Parameters:

- buffer: starting address of buffer (choice)

Input Parameters:

- sendbuf: address of send buffer (choice)
- count: number of entries in buffer (integer)
- datatype: data type of buffer (handle)
- op: reduce operation (handle)
- root: rank of broadcast root (integer)
- comm: communicator (handle)