Lab #9 – Derived data types in MPI

So far, we have been communicating data between processes in MPI. But it has usually been just integers and real numbers. Sometimes it's a single value and sometimes an array of such values. The reason why this has worked out so conveniently for us is that MPI is built to handle this information. For example, the MPI_Reduce() function has a parameter where we indicate the type of values to send, and how many.

But what if you wanted to send a value that was not a simple (primitive) predefined type in MPI? In section 3.5, we learn that there is a way to define our own data type in MPI. It would be a good idea to look over section 3.5 for today's activity.

Usually, when we want to define our own data type, it's because we want something that can contain several different attributes at once. In Java, we create a class. In C, the way to do this is to define a struct. For parallel programming, we will continue this idea by telling MPI what our struct is made from. Here are the steps:

1. Create your struct. Create arrays of these objects as you would normally. Typically, process 0 will need to keep around the total array, and each process has a local array which might have part of this data.

2. Determine the byte displacements of each attribute. Then, you are ready to define three special arrays for MPI. (See page 118)
    a. block_lengths – how many of each base type? These numbers are just 1 unless an attribute itself is an array.
    b. Displacements – the number of bytes from the beginning of the struct.
    c. types – an array of MPI_Datatype.

    For example, let's say your struct consists of a double followed by 2 int values. The block lengths would be 1, 1, 1. The displacements would be 8, 12, 16. And the types would be MPI_DOUBLE, MPI_INT, MPI_INT.

3. Declare a variable called new_type of type MPI_Datatype. We call 2 special MPI functions to prepare our new type. See page 119 for precise specifications.
    a. MPI_Type_create_struct
    b. MPI_Type_commit

4. When you use an MPI communication function such as MPI_Gather() or MPI_Reduce(), make use of your new_type in place of where you would have previously used MPI_INT or some other built-in type.

5. The book goes on to say that when you reach a point in your program where you are sure you won't need to use this struct type for communication anymore, you can deallocate it by calling the MPI_Type_free() function.

Lab activity:

Before you begin with today's new work, please finish your previous lab on Yahtzee.

Let's write a program that computes the centroid of a 2-dimensional region. In other words, we are computing a weighted average. On the class Web site there is a folder containing input files for today's lab. The first file is very short, and is intended to be a simple check on your program's calculations before you get too far along. The other files are larger representing real-world input.

Each line of input contains a location represented by x and y coordinates, followed by a weight. The centroid that your program needs to compute is the weighted average of all of the x values, along with the weighted average of all of the y values. For example, let's say we have just two points:

Point #1: x = 4, y = 5, weight = 10

Point #2: x = 10, y = -1, weight = 20

We would compute the centroid as follows:

Weighted sum of x = 4*10 + 10*20 = 240

Weighted sum of y = 5*10 + (-1)*20 = 30

Sum of all the weights = 30 (and we would divide the sums by this value…)

Centroid x = 240 / 30 = 8

Centroid y = 30 / 30 = 1

And we would conclude the centroid is the point (8, 1).

1. You need to create a struct for your program to represent a point (i.e. one line of input). What attributes do you need? Please note that the values of x and y are real numbers. (See the input files)

2. The name of the input file should be specified at the command line.

3. Create a separate function to read in the input file. This function should be called only from process 0.

4. Inside main(), create an MPI-style program as usual. Remember to have variables for the current process rank, the total number of processes, and the processor name. You will also need local and total arrays of points.

5. Each process needs to be given part of the array of points to work on. It will thus be computing what we may call a local_centroid.

6. The local_centroids calculated by each process need to be sent back to process 0. Process 0 will compute the overall centroid and print out the answer. Please note that a local centroid value is not meaningful unless we also have the weight associated with it. Therefore, each process actually should be storing its local centroid information inside a point struct.

   Process 0 needs to compute a weighted average on these points. The MPI_Reduce() function does not do weighted averages, just simple aggregate functions like add and max. So, you should use the MPI_Gather function instead. Finally, process 0 can perform a weighted average calculation on this small array of points that came from all the processes.

7. Run your program with input1.txt. You should be able to tell immediately if your answer is correct. If so, run it on the other input files. For example, input2.txt shows the general population distribution of the United States and Canada. Use Google Earth to find the centroid in each case. Are your answers reasonably accurate?

Please remember that your first programming assignment is due at 9:00am tomorrow.