CSc 522, Fall 2014, Program #2: MPI critical-path profiler

Due date: Monday, November 10th, at 9am. No late assignments will be accepted.

In this assignment you will build an MPI profiler. For any MPI program that is run, the profiler should generate a critical-path profile. Broadly speaking, the user will execute a program, and you will produce a directed graph called a *MPI task graph* with the critical path annotated. You will also produce summary information about the program. Details are given below.

Creating a Profiler

The way you create the profiler is by using the PMPI layer. Essentially, MPI is designed in such a way that each MPI function is defined as a weak symbol. That means that all MPI functions have an implementation within the MPI library. However, if you, the programmer, override that implementation by creating your own version of the MPI function, your version will instead be invoked. The existing MPI library version of the calls simply invokes a function by the same name except there is a "P" before it. For example, if the user invokes `MPI_Barrier(MPI_COMM_WORLD)`, the MPI library implementation of this function looks as follows:

```
MPI_Barrier(MPI_COMM_WORLD) {
  PMPI_Barrier(MPI_COMM_WORLD);
}
```

All the "real" code that implements a barrier is within `PMPI_Barrier`. This allows you to override `MPI_Barrier` as follows:

```
MPI_Barrier(MPI_COMM_WORLD) {
  Pre_MPI()  // you probably want to pass in an MPI opcode here
  PMPI_Barrier(MPI_COMM_WORLD);
  Post_MPI()  // you probably want to pass in an MPI opcode here
}
```

For example, one thing one might do within your version of `MPI_Barrier` is add one to a counter, so you can keep the total number of barriers executed.

To provide a generic wrapper that will intercept all functions, please use Todd Gamblin's MPI wrapper generator (not required if you really feel like you need to implement this yourself, but recommended). See `https://github.com/tgamblin/wrap` for details.

MPI Task Graph

During execution of the MPI program, you need to generate an MPI task graph. Below, we define terms and explain how to do this. An MPI task graph is a directed, weighted graph with the following attributes:

- There is one vertex corresponding to each **invoked** MPI operation (**if an MPI operation is invoked n times, there will be n vertices for that MPI operation**).

- There is an edge between every pair of consecutive vertices v and w, and the edge weight is the wall-clock time that has elapsed from the end of the MPI operation corresponding to v and the beginning of the MPI operation corresponding to w.
- If the MPI operation is an `MPI_Send` or `MPI_Isend`, then there is also an edge from vertex v (where v corresponds to the `MPI_Send` or `MPI_Isend`) to a vertex z, which is the matching `MPI_Recv` or `MPI_Wait` call, with weight equal to the message latency. Assume all `MPI_Send` operations do *not* block, and assume that an MPI rank never sends to itself.
- To estimate the message latency, you are to run a series of benchmarks that will generate a function that takes as input a message size (when necessary) and returns the associated message latency. This experiment requires just two nodes for point-to-point messages. For the point-to-point experiments, set up a program that takes as input a message size and then has rank 0 send to rank 1 a message of that size and then receive a message (from rank 1) of that size. Rank 1 performs those actions in the opposite order (first receive, then send). This "ping-pong" should be done a large number of times. The message latency is then the total time divided by the number of repetitions, divided by 2 (since we are estimating one-way latency). Perform this experiment for many different message sizes, e.g., 4 bytes, 8 bytes, 16 bytes, . . ., 32K bytes. Take the series of (byte, time) pairs and perform a linear regression to get the function you need. Software packages such as Excel, Google Spreadsheet, and R can do the regression. Note that for the collectives, your regression will have two independent variables. (Also, you should execute all the collectives, as `MPI_Alltoall` has a different latency than a barrier.)
- There must be a single vertex representing each of `MPI_Init` and `MPI_Finalize`, with appropriate edges to the first operation on each rank (for `MPI_Init`) and edges from the last operation on each rank (for `MPI_Finalize`).
- For collectives (specifically, `MPI_Barrier`, `MPI_Alltoall`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Allreduce`), you should have one vertex with a fan-in and fan-out (see Figure 1). Note that this is not actually correct for most of the collectives, but we are making a simplifying assumption. For all collectives, the (vertex) weight should be the based on your regression function for that collective.
- Also, convert `MPI_Waitall` into multiple equivalent calls to `MPI_Wait`.

Critical Path

After program execution (at `MPI_Finalize`), the MPI critical path graph should then be displayed on the screen, and, you must save the critical path itself to a file in the following strict format: (1) each vertex must contain the MPI function name with the exact capitalization and punctuation conventions used in MPI itself, followed by one blank space and then the rank; (2) for each edge, (a) if it is a computation edge, the integer value (rounded), or (b) if it is a latency edge, the number of total bytes in the message. Separate all fields by one blank space. For `MPI_Init`, `MPI_Finalize`, and any collective, use -1 for the rank. Please output this into the file `critPath.out`. Figure 2 has an example task graph and corresponding output to be placed in `critPath.out`. Note that the send-receive edges have the number of bytes in parentheses.

For on-screen display, you should use the "DOT" tool (see: `http://www.graphviz.org/`, along with the "Cluster" example at `http://www.graphviz.org/Gallery/directed/cluster.html`.

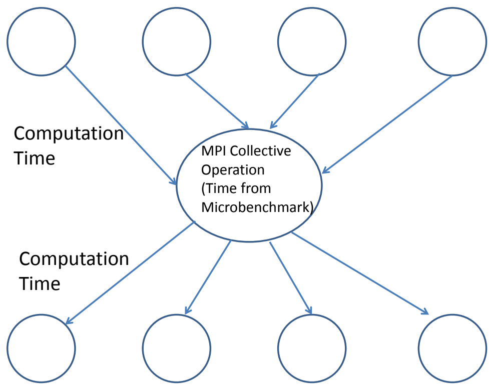

Figure 1: Pictorial representation of how all collectives should look.

The Cluster example is quite similar to what you want (though only for one process at this point). The edge weights are mandatory, and the vertices must be named with the MPI function they represent. If you use another tool, I cannot help you with it. Your output must be the entire MPI task graph with the critical path colored red, with all other edges colored black. Also, I am requiring that the vertices be labeled with the MPI operation names.

The MPI task graph graph is directed and acylic, so there is a relatively simple linear time dynamic programming algorithm to find the critical path. You can find the algorithm, among other places, at `http://en.wikipedia.org/wiki/Longest_path_problem`. (The critical path is the same thing as the longest path.) You will need to convert your DOT graph to an actual graph representation such as an adjacency list before running the critical path algorithm.

To find the critical path via the algorithm above, you must first have a consistent, merged task graph that represents the entire program over all nodes. The issue is that you need to collect the graph on each node locally, and then you need to merge them into one graph after `MPI_Finalize` is invoked.

During execution, we suggest the following approach:

- Upon each MPI operation on each node, write record to a local file. Add any metadata along with that record that you need.

Then, after execution, we suggest the following:

- Accumulate all per-rank MPI task graphs onto a single node.
- Use the per-node files to create one consistent MPI task graph.

Simplifying Assumptions

You should support only the following operations: `MPI_Barrier`, `MPI_Alltoall`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Irecv` `MPI_Wait` `MPI_Waitall`.

```
critPath.out contents

MPI_Init  -1
5
MPI_Send 0
100
MPI_Recv 1
10
MPI_Barrier -1
1
MPI_Finalize  -1
```

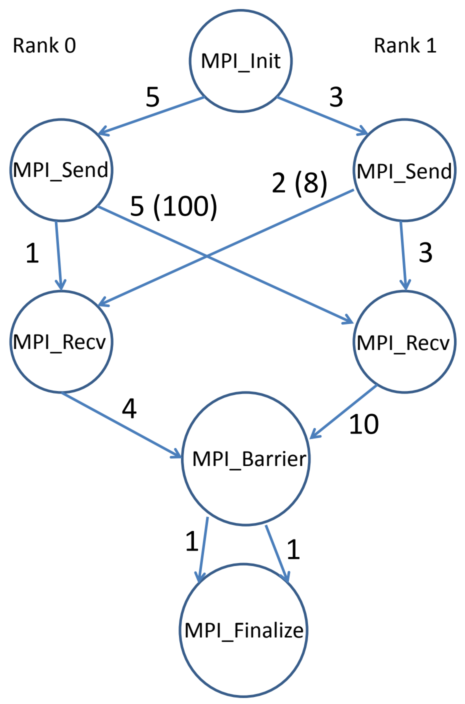

Figure 2: Critical path output (left) for an example MPI task graph (right). The notation 5 (100) on the send-receive edge means that the cost is 5 and the number of bytes is 100.

Other Profiling Information

You are also to produce the following:

1. Determine how many times each MPI operation is invoked by the client program.
2. Find the average, min, median and max execution time spent in the MPI library, per MPI function. Note this should be timing the PMPI functions, so that your code is not counted as part of the MPI operation time.

For items 1 and 2 above, please generate a table; the rows are MPI operations with the precise MPI operation name, and there are additional columns for number of invocations and average, minimum, median, and maximum execution times for that operation. **Important:** Exclude `MPI_Finalize` from this table. This must be done as follows: output this data to a file called `stats.dat`. You must have a header row that has fields exactly as *Function, Invocations, Mean, Min, Median, Max*. You also must have one data row for each unique supported MPI function. Separate each column with a tab and nothing else.

Experiments

Use your critical path system on 4 nodes to make sure it is functioning correctly.

To turn the assignment, create a tar file named `prog2.tar` with your profiling code along with a Makefile that will build a target of `app` given `app.c` (for an arbitrary program “app.c”). **Note: if you do not name your file prog2.tar, you will lose points.**