CSc 522, Fall 2019, Program #2: MPI critical-path profiler

Due date: Tuesday, November 12th, at 8am. No late assignments will be accepted.

In this assignment you will build an MPI critical-path profiler . Broadly speaking, the user will execute a program, and you will produce a directed graph called an *MPI task graph* with the critical path annotated. Because this assignment uses MPI, you must use C or C++ to intercept MPI calls (which is a very small part of the assignment). You may use any language you wish to complete the rest of the assignment.

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()  // do something before calling the "real" barrier
  PMPI_Barrier(MPI_COMM_WORLD);
  Post_MPI()  // do something after calling the "real" barrier
}
```

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. Another might be to time the barrier.

To provide a generic wrapper that will intercept all functions, please use the MPI wrapper generator, developed by Todd Gamblin of Lawrence Livermore National Laboratory. While I am not requiring this if you really feel like you need to implement this yourself, I suggest spending the (short) time to learn it. See `https://github.com/tgamblin/wrap` for details. The distribution comes with an example wrap file, and you can use one of the examples in there to generate all your wrapper functions.

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, acyclic, weighted graph with the following attributes:

- Your task graph will support only the following operations: `MPI_Init`, `MPI_Finalize`, `MPI_Barrier`, `MPI_Alltoall`, `MPI_Allreduce`, `MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Irecv`, and `MPI_Wait`. **Note: do not put `MPI_Comm_rank` or `MPI_Comm_size` in your task graph.**

- 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 on the same rank, and the edge weight is the wallclock 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 (where z corresponds to the matching [remote] `MPI_Recv` or `MPI_Wait` call, with weight equal to the message latency. (The matching call is `MPI_Wait` when it is associated with an `MPI_Irecv` call.) Assume all `MPI_Send` operations do *not* block, and assume that an MPI rank never sends to itself.

- You must support message tags. This means that if rank i sends two messages with tags T_1 and T_2 to rank j, but rank j receives the message with tag T_2 first, your MPI task graph should correctly reflect this.

- 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. To do this, 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. The regression will produce a slope and a y-intercept. Write these two values to a file called `communicationCoefficients` (in the current directory) and then read these two values in when you intercept `MPI_Init`. They should be double precision, and slope should be first, and y-intercept second. The reason for this is that when I test your code I will possibly overwrite `communicationCoefficients` with my own values.

- 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`, and `MPI_Allreduce`), you should have one vertex with a fan-in and fan-out (see Figure 2). You will assume that all collectives take zero time. Also, assume that all messages, including collectives, use `MPI_COMM_WORLD` as the communicator.

- **Important:** your algorithm to create the MPI task graph must run with complexity $\mathcal{O}(r \times m)$, where r is the number of ranks, and m is the number of messages. Note that this is a lower bound, as simply processing all the messages takes time $r \times m$. Notably, you may *not* use an $\mathcal{O}(r \times m^2)$ algorithm.

Critical Path

After program execution (at MPI_Finalize), you must run an algorithm to find the critical path. 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 online (or perhaps in a library for the language you are using[1]). (The critical path is the same thing as the longest path.)

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, I suggest that upon each MPI operation on each node, write a record to a local file. Add any metadata along with that record that you need. Then, after execution, you can simply use the per-node files (the filesystem is shared) to create one consistent MPI task graph.

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, **only** the integer value **in seconds (rounded to the nearest integer)**, or (b) if it is a latency edge, **only** the number of total bytes in the message. You **must** use separate lines for vertices and edges; there should never be two vertices, two edges, or a vertex and an edge on the same line. To compute the number of bytes, use MPI_Type_size. **You may assume that user programs only send messages of either one int or one double.** Separate all fields by one blank space. For MPI_Init, MPI_Finalize, and any collective, use -1 for the rank. You must 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. Please understand that I will be using automated grading for critPath.out files, so if you deviate from the rules above, my grading program will deem it incorrect. An example of the critical path file, for the program in Figure 1 is shown in the left-hand side of Figure 2. (See the end of the document for all figures.)

An important assumption for this assignment is that the critical path will be unique. Without this assumption, I could not automate grading of the critical path output, as two correct programs could produce two different critical paths. This means that any user program I will test against your profiler will have a unique critical path.

In addition, we will generate a graphical version of the MPI task graph and its critical path on the screen. For this, you must use the "DOT" tool (see: http://www.graphviz.org/. The graph should similar to the right-hand side of Figure 2; the entire MPI task graph should be displayed. The edge weights are mandatory (with number of bytes in parentheses for any message between different ranks), and the vertices must be named with the rank and the MPI function they represent. If you use another tool, I cannot help you with it. Your dot output file must be a representation the entire MPI task graph with the critical path colored red, with all other edges colored black. Also, you must label the vertices with the MPI operation names. Please use only three digits to the right of the decimal point.

The DOT graph must be output into the file dotOutput.gv. Do **not** display the graph on the screen. I will not be using automated grading for your on-screen displays; that will be manual.

[1] If you are using the networkx library in python, be warned that the longest path algorithm does not work for the MultiDiGraph class. You can see me for an explanation.

Suggestions

- You have several weeks to do this assignment, but there are a lot of parts that you need to get working. As always, I suggest that you start early on the assignment.
- Your code to create the graph and find the critical path should be executed right after your `MPI_Finalize` wrapper calls `PMPI_Finalize`. Note that `PMPI_Finalize` implies a barrier synchronization, so all MPI processes are done at this point. Make sure to call your code with only one MPI process.
- You will need a way to match up `MPI_Send`/`MPI_Isend` with `MPI_Recv`/`MPI_Wait`. For example, consider the case where rank i invokes two identical `MPI_Send` operations, and the destination is rank j for both, and two `MPI_Recv` operations are invoked there. You need to make sure that you match the first send with the first receive and the second send with the second receive. There are a couple of ways this can be done: (1) piggyback a unique identifier along with the message, or (2) send an extra message with the identifier, right after a send is invoked. I suggest the second solution[2].
- Keep in mind that an MPI receive (or wait) operation *matches* an MPI send *only* when all of the following match: sender/receiver, tag, and type. Denoting the tag and type as the *signature*, the easiest way I know of to meet the time complexity constraints given above is to use a hash table indexed by signature. The sending of a message would cause an entry to be added, and receiving would then match in the hash table using the signature.
- While all ranks must participate in all collective operations (because of the assumption that the communicator always is `MPI_COMM_WORLD`), you cannot assume that all ranks do identical operations. For example, all ranks could invoke a barrier, followed by some ranks invoking another barrier immediately, but other ranks do point-to-point communication before invoking the second barrier.

Experiments

Use your critical path system on at least 4 nodes to make sure it is functioning correctly. However, there is nothing to turn in here.

To turn in 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`. Note that if `app.c` changes, you must re-build `app`. Upon untarring `prog2.tar`, the files should all be in the directory `prog2`. **Note: if you do not name your file prog2.tar, you will lose points. In addition, if your makefile does not work, you will lose points.** The turnin directory is `csc522-f19-prog2`.

[2] The second solution has a subtle race condition when handling MPI calls we will not be supporting in this assignment, so it is safe.

```
MPI_Init(&argc, &argc)

if (me == 0) {
        usleep(600000);
        MPI_Send(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
else if (me != 3) {
        MPI_Recv (&y, 1, MPI_INT, me-1, tag, MPI_COMM_WORLD, &status);
        usleep(600000);
        MPI_Send(&x, 1, MPI_INT, me+1, tag, MPI_COMM_WORLD);
}
else {
        MPI_Recv (&y, 1, MPI_INT, 2, tag, MPI_COMM_WORLD, &status);
        usleep(600000);
}

MPI_Barrier(MPI_COMM_WORLD);

if (me == 0) {
        usleep(1600000);
        MPI_Send(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
}
else if (me == 1) {
        MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        usleep(400000);
}

MPI_Finalize();
```

Figure 1: Example MPI program. Unimportant parts of the program are deleted for simplicity.

```
MPI_Init -1
1
MPI_Send 0
4
MPI_Recv 1
1
MPI_Send 1
4
MPI_Recv 2
1
MPI_Send 2
4
MPI_Recv 3
1
MPI_Barrier -1
2
MPI_Send 0
4
MPI_Recv 1
0
MPI_Finalize -1
```

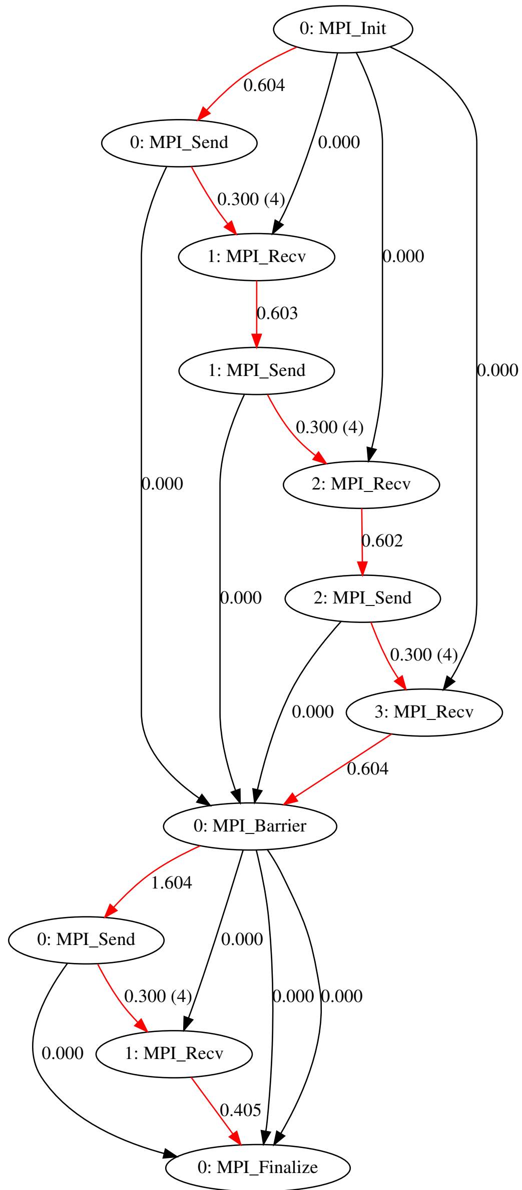

Figure 2: Critical path output (left) for an example MPI task graph (right); both are for the program shown in Figure 1 . The notation X (Y) on the send-receive edge means that the cost is X and the number of bytes is Y.