CSc 522, Fall 2019, Program #3: Redundant MPI

Due date: Wednesday, December 11th, at 8am. No late assignments will be accepted.

In this assignment you will build a redundant MPI implementation. Specifically, you will implement, using PMPI, a scheme by which an MPI program runs redundantly such that if one MPI process fails, the program will continue with the replica process. In other words, you will be able to survive certain types of failures.

Assignment details are given below.

Requirements

The following are required.

- You are to implement the *parallel* protocol as described in the Ferreira SC11 paper.
- If the user wishes to execute an MPI program on n processes, we assume the user desires, additionally, an n process replica—so full replication of the user's n processes. To do this, the program will be executed with $2n$ processes, i.e., `mpirun -np 2n <rest of params>`. You will interpret this as follows: the first n processes in the `MPI_COMM_WORLD` communicator will be the primary, and the second n processes will be the replica.
- You will only support (and therefore intercept) the following communication routines: `MPI_Send`, `MPI_Recv`, and `MPI_Barrier`. You will intercept `MPI_Send` and `MPI_Recv` and implement the redundancy protocol. For `MPI_Barrier`, "dissolve" it into `MPI_Send` and `MPI_Recv` calls (you can do this by intercepting `MPI_Barrier` and making calls to `MPI_Send` and `MPI_Recv` instead of invoking `PMPI_Barrier`). Note that within `MPI_Barrier`, you must invoke `MPI_Send` and `MPI_Recv` (not `PMPI_Send` and `PMPI_Recv`), because you need redundancy there also.
- To support these communication functions, you will need to also need to intercept other MPI functions.
  - First, you must intercept `MPI_Comm_rank` and `MPI_Comm_size`. Specifically, you need to make sure that `MPI_Comm_rank` maps any processes in the range n to $2n - 1$ to 0 to $n - 1$. Also, `MPI_Comm_size` must return n, not $2n$.
  - Second, you will almost certatinly want to intercept `MPI_Init` (as you will be setting things up there).
  - Third, you will be intercepting `MPI_Pcontrol` (see below). This function provides a common interface for profiling—it does not actually do anything, but instead exists so that the programmer can inform the profiling layer of application-specific items. In our case, the programmer (i.e., me) is informing the profiling layer (i.e., you) of which process is killed.
- You must support `MPI_ANY_SOURCE` on an `MPI_Recv`.
- You must not send messages over `MPI_COMM_WORLD`. You need to create three communicators within your interception code: one for messages that are sent within the primary group, one for messages

that are sent within the replica group, and one for sending messages between the groups. A couple of helpful functions are `MPI_Comm_split` and `MPI_Comm_dup`.

- As mentioned above, a process will be "killed" (by my test programs) explicitly, via user program calls to `MPI_Pcontrol`. The single parameter to `MPI_Pcontrol` will indicate the process to be killed; note that if a replica is to be killed, the id will be in the range n to $2n - 1$. Note that the process will not actually be killed, but you are **not** to use it for the rest of the program. The "killing" of a process takes effect at the `MPI_Barrier` following the call to `MPI_Pcontrol` (which will be invoked on all processes); any process killed must call `MPI_Finalize` rather than returning from the barrier; this will ensure that no further MPI calls occur. User programs (i.e., my test cases) will not invoke any MPI calls (other than another `MPI_Pcontrol`) between an `MPI_Pcontrol` and the next barrier. You also need to have the now-dead process invoke `exit(0)` immediately after invoking `MPI_Finalize`.

- It will **never** be the case that, for a particular process, a primary and replica will both be killed. Therefore, you do not need to do any checkpointing. However, you need to be able to survive multiple failures (for example, primary process zero and replica process one can both fail). This means that when a process fails, the surviving partner process must take over the role of the failing partner for the rest of the program. For example, if A and B are primaries and A' and B' are replicas, then if A fails, A' now has two processes to which it must communicate: B and B'. Note that you have the option to simply suppress a send from B to A' as long as B' is still alive (because in such a case, B' will send to A', and A' doesn't actually need the second message).

- Note that it is possible for the entire primary set (or the entire replica set) to be killed.

- Moreover, in any program in which an `MPI_ANY_SOURCE` is used, any processes killed will *only* be ones that solely receive; and, only *one* process in the entire program will be killed.

- The total number of MPI processes in the primary (and therefore also in replica) will be at least two.

- You may not send messages within your implementation that are do not have a matching receive.

- My test programs will only ever use `MPI_COMM_WORLD`.

- My test programs will not send and receive across a barrier, i.e., there will not be a barrier between the invocation of the send and the receive.

Implementation Issues

Here are some suggestions.

- You will want to create the new communicators within `MPI_Init`.

- You must ensure that the primary group of processes and the replica group of processes does not get too far out of phase. In other words, if the primary has a process fail, the replica process that is used in its place needs to be "in sync" with the primary processes. If, for example, the replica process is two global synchronization points behind the primary process it is replacing, there is no way to bring it up to date. You may address this any way you like, but the easiest way I can see is to have the senders do a message exchange.

Note: ignoring the killing of a process will be considered academic dishonesty. If you do not implement something, you must disclose it to me.

Experiments

Use your redundant MPI system on 4 (total) processes to make sure it is functioning correctly (two primary, two replicas).

To turn the assignment, create a tar file named `prog3.tar` with your profiling code along with a Makefile that will build a target of `app` given `app.c`. **Note: if you do not name your file prog3.tar, you will lose points.** The turnin directory is `csc522-f19-prog3`. I will be executing your program on lectura.