CSc 522, Fall 2014, Program #3: Redundant MPI

Due date: Monday, December 3rd, at 9am. No late assignments will be accepted.

In this assignment you will build a redundant MPI implementation. You will implement, using PMPI, a scheme by which an MPI program runs redundantly such that if one MPI rank fails, the program will continue with the replica rank. In other words, you will be able to survive certain failures. Note that depending on how you implement your solution, you may need to understand MPI communicators; you can find this information on the web. Essentially, a communicator is a mechanism that alows you to create a self-contained (sub-) group of ranks.

Assignment details are given below.

Assumptions

The following are assumptions that you must make.

- You are to implement both the *mirrored* and *parallel* protocols as described in the Ferreira paper.
- The user will execute an MPI program on n ranks, desiring an n rank replica. The user will then specify $2n$ ranks, i.e., `mpirun -np 2n <rest of params>`. You will interpret this as follows: the first n ranks in the `MPI_COMM_WORLD` communicator will be the primary, and the second n ranks will be the replica.
- You will only support `MPI_Send`, `MPI_Recv`, and `MPI_Barrier`. For `MPI_Barrier`, you should "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`).
- You must support `ANY_SOURCE` on an `MPI_Recv`.
- A node will be "killed" (by my test programs) explicitly from user program calls to `MPI_Pcontrol`. The single parameter will indicate the node 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 we will not actually kill the process, but, if it's "killed", you are **not** to use it for the rest of the program. This takes effect at the next `MPI_Barrier` after the call to `MPI_Pcontrol` (which will be invoked on all ranks). User programs will not invoke any MPI calls after being killed (even before becoming effective at the next barrier). Any node that is killed must immediately call `MPI_Finalize`.
- It will **never** be the case that, for a particular rank, 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 rank zero and replica rank one can both fail). This means that when a rank fails, the surviving partner rank must take over the role of the failing partner for the rest of the application.
- You may handle barriers any (correct) way you see fit.
- The total number of MPI ranks in both the primary and the replica will be at least two.

Implementation Issues

Here are some suggestions.

- At `MPI_Init`, you may need to create multiple communicators (you can use `MPI_Comm_create`). If you use such an implementation, you will need one each for the primaries and replicas. You can use `MPI_COMM_WORLD` for communication between the primary and replica (which is used when a rank fails). Moreover, you would need to re-create the appropriate communicator upon failure of a rank.
- You will need to modify (via interception) `MPI_Comm_rank` and `MPI_Comm_size`. Specifically, you need to make sure that `MPI_Comm_rank` maps any ranks in the range n to $2n-1$ to 0 to $n-1$. Also, `MPI_Comm_size` must return n, not $2n$.
- You will need to intercept all calls to `MPI_Send` and `MPI_Recv` and implement the necessary items for the mirrored or parallel protocols. As mentioned above, you can use subcommunicators or do everything with `MPI_COMM_WORLD`.
- For the parallel protocol, you must ensure that the primary group of nodes and the replica group of nodes does not get too far out of phase. In other words, if the primary has a node fail, the replica node that is used in its place needs to be "in sync" with the primary nodes. If, for example, the replica node is two global synchronization points behind the primary node 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.
- The mirrored protocol posts multiple receives in response to a single receive, because it must receive from both the primary and the replica. In a real implementation (as discussed in the paper we read), this means that one of those receives may never actually happen (if one of the primary or replica dies). This is quite complicated and involved canceling one of the receives. You need not worry about this because of the simplifying assumption that failures are known to you and deferred to the next barrier.

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

Specifying the Parallel or Mirrored Protocol

Whether to use the parallel or mirrored protocol will be encoded in an environment variable `PROTOCOL_TYPE`. Its two possible values will be `PARALLEL` and `MIRRORED`. (Use `getenv` to retrieve the value.)

Experiments

Use your redundant MPI system on 4 (total) ranks to make sure it is functioning correctly (two primary, two replicas). Compare the performance of mirrored and parallel and discuss the results in a file `results.pdf`. Please be brief.

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 `cs522-f14-prog3`.