THE UNIVERSITY of EDINBURGH

MPI RMA Exercise

Advanced MPI course

epcc

1 Introduction and Aims

In this exercise you will explore using MPI RMA for halo swapping in a simple computational code. The code currently issues non-blocking P2P communications the halo swap and we want to replace this with RMA. It is a very simple example of what you might want to do on your own existing codes and replacing P2P calls (especially halo swapping) with RMA is popular for increasing performance and scalability.

2 The existing code

This is a simple Jacobi iteration in 2D, with decomposition in 1D. Decomposition is done over the X dimension and is allocated as evenly as possible amongst the processes. After computing the initial absolute residual, the code progresses in iterations. Firstly halo swapping is performed with the rank-1 and rank+1 neighbour, the current relative residual is then calculated from the ***u_k*** array and the calculation then performed, writing new values into the ***u_kp1*** array (which are values for the next iteration.) Lastly the ***u_k*** and ***u_kp1*** values are swapped around for the next iteration by using a ***temp*** array (in the C code lines 86-88, Fortran code lines 93-95.)

We provide both a C version and Fortran version, use which ever language is most familiar to you.

Important: For simplicity we are going to assume an even decomposition of data in the X dimension, this will make your RMA work simpler. The submission script is set so X=1024, Y=512 running over 128 processes, I suggest keeping these values the same until you get the exercises working.

3 Exercise

These exercises involve writing MPI code, you can refer to the API online, which will give you the syntax of all the required calls, at

http://www.mpich.org/static/docs/v3.2/www3/

3.1 Compilation

Make sure you are in your `/work` filesystem and download the ***`jacobi.zip`*** file onto ARCHER:

wget http://www.archer.ac.uk/training/course-material/2016/09/160929_AdvMPI_EPCC/jacobi.zip

Then unpack the archive using the ***`unzip`*** command, switch to the `jacobi` directory and if you prefer C go into the *c* directory, if you prefer Fortran then the *f* directory. Lastly issue the ***`make`*** command.

After compilation the ***jacobi*** executable will have been created.

3.2 Submit the existing code

Submit this existing code to ARCHER via ***qsub subjacobi.pbs*** you can track your job's progress via ***qstat –u $USER***

The output of the job will be written as a file in your current directory (called ***Jacobi.oXXXXX***) where the X's are the job ID number. This contains information about the global system size (size in X and Y), a summary of progress as the code ran, the number of iterations it took to converge and the total runtime in seconds.

3.3 Replace non-blocking P2P with RMA (using fences)

Refactor the code (specifically the halo swapping at lines 62-70 of the C code or lines 69-77 of the Fortran code) to use RMA rather than non-blocking P2P. Create a window on ***u_k*** at the start of the code and free it at the end. Use the fence synchronization (use no assertions, i.e. 0 for the assert argument) that we discussed in the first lecture to start and stop the epoch. You can use either the ***get*** or ***put*** communication calls and I suggest operating on the buffer ***u_k*** directly. Be a bit careful when thinking about which target displacements (and locations in your ***u_k*** buffer) should be used for each communication call.

The default submission script is splitting up X=1024 over 128 cores, this ensures an even decomposition of data which is easier to work with. I suggest assuming this even decomposition, certainly until you get a version working.

Submit the job to ARCHER and time it. How does it compare to the non-blocking P2P? Now consider which assertions are appropriate for which fences. Pop these in, recompile and resubmit – does the addition of these assertions decrease the runtime?

3.4 Modifying the code to use Post-start-complete-wait (PSCW)

Note: These concepts are discussed in the second RMA lecture

Instead of fence synchronization use PSCW, for every process you will need to think about the groups of ranks which need to be involved in the exposure epoch and which need to be involved in the access epoch. We are just changing the synchronisation here - your communication calls should remain unchanged from the fence code.
Once you have done this retime the code, does using PSCW make an impact on the runtime?

3.5 Modifying the code to use lock & unlock

Note: These concepts are discussed in the second RMA lecture

Use the lock/unlock (with a shared lock) synchronization calls and see how this impacts the runtime. Note that this code doesn't really suit the lock/unlock approach (as we need some synchronisation between iterations for data consistency.) Therefore I suggest you place a barrier after this halo swapping to ensure the target does not rush ahead with further iterations. The ***unlock*** guarantees that RMA operations have completed both at the origin and target – hence you can use either a ***get*** or a ***put***

How about using an exclusive lock, does this impact the overall runtime? On each iteration we are locking and unlocking – instead use a ***flush*** and move the lock (use a shared lock) before the loop and unlock after the loop. We are now only creating one access epoch per process for the entire code, how does this impact the runtime?

4 Summary

In this exercise we have looked at refactoring an existing computational code to replace some of the P2P calls with RMA. You can see that how many options there are available to you and this increases as we work with more complex programs. Over and above simply replacing P2P calls there are methods, such as double buffering, which work well with RMA and can provide significant performance benefits.