Scalability of our geometric decomposition

1 Introduction

We discussed in the lecture about how, when we are applying the geometric decomposition pattern, the granularity of our sub-domain is crucially important. Too large a granularity (a small number of large sub-domains) means that there isn't enough parallelism within the system, but too fine a granularity (a large number of small sub-domains) then each UE doesn't have enough computation in order to outweigh the cost of parallelism (namely that of communication.) I also mentioned that it can be very difficult (or impossible) to find the optimum granularity theoretically and doing some experimentation by actually running the code at different core counts is often the only way to identify the best configuration.

It was also discussed that, at the potential cost of code simplicity, additional techniques can be applied to improve the scalability and performance of the code.

2 Scalability testing

We are going to do some large runs on ARCHER2 and in order to make this a bit more interesting from a communications point of view, we have a few different versions of the code with different parallel communication patterns. This code is the same as what you were working on in practical 1, so whereas I am supplying it to you, you are also free to use your own code rather than mine if you wish.

IMPORTANT on ARCHER2 your home directory is not visible to the compute nodes, instead you need to place files in your work directory which can done via executing *cd /work/m24oc/m24oc/$USER* (assuming you are in the m24oc group. If you are not sure you can find this via executing the *groups* command).

The tasks to do in this practical are:

1. Log into ARCHER2 (login.archer2.ac.uk) and cd into your *work* directory (*/work/m24oc/m24oc/$USER*)
2. Make sure you have built the code, if you are using my code then just issue *make* in the C or Fortran directory.
3. Perform some strong scaling tests on the code, as-is. In the script we have a size of 16384 by 1024 (16.7 million global grid points) solving to a relative residual of 7e-4 and I suggest keeping this the same. Try running on 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 processes and record the runtime for each. For more information about how to change the number of cores in your submission script then view the next section, *Changing the number of cores on ARCHER2* of this worksheet.
4. The submission script will submit the blocking version of the code to the queue, repeat step two for the non-blocking version too.

You will need to provide your user's account ID to the submission command line, for instance ***sbatch --account=m24oc submit.srun*** (you will charge to the central MSc ARCHER2 budget rather than individual ones as done on Cirrus).

Taking a long time to get through the queue? Don't worry, once you have submitted the required jobs then you can move onto section 4 whilst these are waiting in the queue to run.

3 Changing the number of cores on ARCHER2

The submission script ***submit.srun*** is for ARCHER2 and set to schedule your job on 512 cores, across 4 nodes (there are 128 CPU cores per node). To change the number of cores you have to:

- Edit the line in the submission script *--nodes=4* , selecting the number of nodes you want
- Each node has 128 cores and by default we are scheduling 128 tasks (MPI processes per node). For smaller runs (over 16, 32, 64 cores) then select one node and change *--tasks-per-node* to equal the number of MPI processes you want

4 Let's look at the performance in more detail

Depending how busy the queue gets it might take some time to run through your scaling jobs, so once you have scheduled them you can start looking at this whilst you wait for them to run.

It is highly likely that the versions of the parallel code (blocking, non-blocking and overlapping communications) exhibit different runtime characteristics – and a key question is why? To explore this in more detail we are going to use some open source profiling tools developed by the Barcelona Supercomputing Centre (BSC) which allow us to explore parallel performance in a visual manner. For this exercise you will need an X server installed, if you are using Windows I would suggest Xming, on Linux this should already be installed and on MacOS you can also download one.

1. Rebuild your executables for profiling, first issue ***make clean*** at the command line and then ***make instrumented***
2. **Important**: There is a ***submit_profiling.srun*** job submission script, use this one rather than the scaling one as it will limit the number of iterations (which is important as the amount of profiling data collected is huge!) It will also run over four node but with only 16 MPI processes per node (64 processes in total), again this is to limit the overall profiling data generated.
3. In a separate window, ssh back into ARCHER2 but with X forwarding enabled (the argument ***-Y*** on Linux, which enables X forwarding and compresses the data to speed up transfer times.) Start the paraver tool by executing /work/z19/shared/paraver/bin/wxparaver
    a. This will start the paraver graphical tool on ARCHER2, which might be quite slow if you are not in Edinburgh. Alternatively you can download this to your local machine (from https://tools.bsc.es/downloads) and scp the .prv profiling file from ARCHER2 to your local machine for exploration.
4. In the Paraver tool click ***Load trace*** and via the menu navigate to the directory that your codes reside in. You should see a file with a .prv ending, load this one up.
5. There are plenty of things to explore in the Paraver tool, we are just going to look at a couple of them here, but feel free to have a play around with other aspects too. Once you have loaded the trace, under the Hints toolbar menu click MPI and then MPI Profile. Once this has generated if you click the magnifying glass (fourth from the left) at the top then it will show

the overall % time spent in computation (which is what we want to maximise) and communication calls (what we want to minimise.)

6. On this same window click the "D" icon (second from the left.) A new window will appear, and this illustrates a timeline of your application. Ignore all the black at the left/middle of the window as this is the application starting up, but towards the end of the trace you will see coloured lines, place the mouse pointer over these and drag a box which has the effect of zooming in. In this zoomed in picture black is computation (what we want to maximise) and the other colours are communication calls. Is there a pattern here at all and what proportion of computation vs communication is your code exhibiting?

5 Advanced - Comparing profiling between the code versions

So far we have looked at profiling of the blocking version of the code I provided to you, let's compare and contrast this against the non-blocking (and then the overlap) version.

1. Edit the ***submit_profiling.srun*** submission script to execute the non-blocking version by changing the name of the executable. Then re-submit the job.

2. Keeping the existing windows open, go back to the main window of Paraver and load in your non-blocking trace file which will have been newly created by this second execution. Again, click the Hints toolbar menu, MPI and MPI profile. Once again click the "D" button (second from the left.)

3. Zoom in as appropriate, based on this comparison, do the different applications (blocking, non-blocking and overlap) exhibit different communication vs computation patterns? Does one application do more iterations in this timespan than the other?

4. By right clicking on both views and clicking ***synchronize*** then it will synchronise the timelines so that they show the same period of time. Have a play with this, although you need to be a bit careful here as the codes are running at different speeds. In the new, timeline window that appears, right click and select ***synchronize*** – this will match the time span of the two timeline windows and allows us to directly compare what is going on.

5. Obviously you can make these timeline windows larger, which might help you see more clearly what is going on. Also you can further zoom into the time by dragging a box around a specific area you want to focus on.

You can follow the same approach for the *overlap* version and compare that one too.

6 Advanced – further strong and weak scaling exercises

In section 2 we compared scaling of the blocking and non-blocking version, there is also an overlap version – resubmit the strong scaling runs for this (just by changing the executable name), how do the runtimes differ (especially at large core counts?)

So far we have been looking at strong scaling, where the global problem size is fixed and local problem size varies with number of processes. Weak scaling, where the local problem size is fixed and global problem size varies with the number of processes can give us a better idea of the scalability of code as we are not hitting the granularity problem. Do weak scaling experiments with a local problem size of 32768 elements, for simplicity a table of the problem size in X and Y for different numbers of processes is included below. Work out the parallel efficiency for each run (more details about this at http://www.shodor.org/refdesk/Resources/Tutorials/Speedup/).

Number of cores	Number of nodes	Size in X	Size in Y
128	1	2048	2048
256	2	4096	2048
512	4	4096	4096
1024	8	8192	4096
2048	16	8192	8192
4096	32	16384	8192
8192	64	16384	16384