Practical session 5: Parallel Scaling

Overview

We are going to:

Create a simple "MPI" program to perform parallel computation.

Run the program, first with one core (a serial job)

Then run it with 2, 4, 8, … cores (parallel jobs)

Calculate the Speed-up based on the jobs' timings

Does the job speed-up with an increasing number of cores?

Notes

We must use the "scratch" storage for this job. The program will use some parallel file writing features that only work on the scratch filesystem. This is another good reason to use scratch instead of the "home" storage area when running jobs (earlier we said scratch is faster and larger, now we have a third reason – it can do parallel input/output from parallel apps.)

Most of you will not be writing any code in your projects so you won't normally need to do the compilation step that we do here. Don't worry about the details of this, just run the commands as shown. If you are interested in what our parallel application does, have a look at the `mandel-mpi-io.c` file for how we've written this app.

Exercise

1. Start with a clean environment. Either log out and back in again or run the following to remove all settings made by any previous module loads.

   ```
   module purge
   ```

2. You will use the "scratch" storage as this is the only place from which to run this job so that the parallel I/O works. We already have a "`mace-course`" directory (folder) in our scratch area, but we can create it if it doesn't exist:

   ```
   mkdir -p ~/scratch/mace-course
   cd ~/scratch/mace-course
   pwd
   ```

   (notice that `pwd` expands the ~ to show your full home directory path)

3. Copy the MPI example folder to your scratch area:

   ```
   cp -r ~/mace-course/mpi-example .
   ```

   (the . at the end is important! It means "current directory" - so we are copying the `mpi-example` folder *from* our home area *to* the current directory – which is in our scratch area. The `-r` flag copies an entire folder.)

4. Now go into the newly copied folder and see what's there:

```
cd mpi-example
pwd
ls
```

You should now be in the `mpi-example` folder in your scratch area.

5. Compile the app. This generates an executable program from the C source file. We need a "compiler" with access to "MPI" libraries. As always, there's a `modulefile` to give us a specific version. The "`make`" command will run the compiler with some settings we've specified in the `Makefile` (in the `mpi-example` folder.)

```
module purge
module load mpi/gcc/openmpi/5.0.7-gcc-14.2.0
make
```

You should now see a new `mandelbrot-mpi` file – this is an executable program we can run from a jobscript.

6. Now submit the job. We've already written the jobscript for you. It will run the program you've just compiled. The jobscript is a *serial* jobscript so the job will use just one core:

```
sbatch mandel-mpi.sh
```

If you are trying this outside of a course session, use

```
sbatch --reservation="" mandel-mpi.sh
```

7. Now let's submit the job again but with a different number of cores. We can specify the number of cores on the `sbatch` command-line instead of editing the jobscript:

```
# Add --reservation="" if outside of a course
sbatch -p multicore -n 2 mandel-mpi.sh
sbatch -p multicore -n 4 mandel-mpi.sh
sbatch -p multicore -n 8 mandel-mpi.sh
sbatch -p multicore -n 16 mandel-mpi.sh
sbatch -p multicore -n 32 mandel-mpi.sh
```

You can submit each job immediately (you don't need to wait for the previous one to finish before submitting the next.)

8. Check the runtime of each job. Our app will report how long it took to do its computation. Hence we can look in the usual job output files:

   ```
   cat slurm-*.out
   ```

   The `*` is a "wildcard" and will show you contents of all files whose names begin with "`mandel-`" and end with "`.out`".

   You can also view the generated images using

   ```
   eog mandel-1.ppm
   ```

   where the 1 is the number of cores used by the job. So you will eventually have files named `mandel-2.ppm`, `mandel-4.ppm` and so on. They should all be identical (same size, same contents) because we have not changed any input parameters, only the number of cores used by the app.

9. Calculate the speed-up for *each* job using the formula: $\text{Speedup}_N = T_1 / T_N$ (where T_1 is the time taken on 1 core and T_N is the time taken on N cores.)

Cores (N)	Runtime (T_N)	Speedup (T_1/T_N)
1		1
2		
4		
8		
16		
24		
48		

10. **Question**: Does the app "scale" as you increase the number of cores? For example, is the 2-core job twice as fast as the 1-core job? Is the 4-core job 4 times as fast? What about the 16 cores job and so on?