PX425 Assignment 5

Introduction

Your task in this assignment is to simulate a scientific problem, choosing an appropriate parallelisation strategy on up to 56 cores of `orac`. You will then measure some of the metrics discussed in lectures to analyse the performance of your program. A working code is provided, which has not been parallelised at all, and may also be significantly optimisable. I strongly suggest reading the entirety of this briefing *before* you start coding. The assessment in this exercise is split roughly threefold between code quality, performance (including improving serial performance), and writing a full report on the results. Please note that it is possible to write a good report even if the parallel performance of your code is not as good as you might have hoped, so do not skimp on that section or leave it to end, as much of the report-writing is easiest if done as you go along.

The scenario

The Empire are planning their next Superweapon. It will be yet another giant spherical space station with a massive laser, only this time considerably *larger* than previous efforts. It has not yet been decided exactly how large: the latest in a long line of black-helmetted villains will decide that based on available resources. The villainous boss demands good wifi reception in his throne room, to make sure he can signal the control room to fire the giant laser at any passing rebel ships.

Because of the complexity of designing a wired network for such a large space station (not a moon!), and then choosing the placement of wireless routers to maximise signal, it is proposed to instead allow for an “ad hoc” wireless network to form [1]. This involves configuring all the wifi-enabled devices to act as routers, and allowing the signal to propagate from device to device. Devices acting as routers can be assumed to be relatively short-ranged, only able to communicate with near-neighbours. They will also be assumed to be distributed throughout the space station with a certain density.

The architects are very keen to know what density of routers will be required before this design of network can be expected to be a success (you may recognise this scenario somewhat from Assignment 3!). There are many challenges associated with maintaining the information required for successful practical routing of data in such a network, but we will not concern ourselves with these in this assignment: instead, we will focus on whether such routing is even possible.

To count as successful, the wireless signal must “span” the space station: that is to say, there must be a continuous cluster of connected routers which reaches all 8 octants of the sphere’s surface. It does not matter if there are clusters of routers that are not connected to this “spanning” cluster: those unlucky users would simply be required to relocate to a different part of the station to acquire a signal.

You are tasked with simulating the behaviour of this network and determining what density can be expected to be sufficient for making a spanning cluster likely (i.e. probability greater than 0.5). This task is equivalent to discovering the ‘percolation threshold’

[1] `https://en.wikipedia.org/wiki/Wireless_ad_hoc_network`

for a continuum percolation model of randomly-distributed overlapping spheres in 3D. You may wish to read up on the meaning of the percolation threshold. Wikipedia may even hold a computed answer to this specific question if you look sufficiently carefully [2].

However, these calculations are computationally demanding, and the architects will need answers very quickly as soon as they make their mind up just how large the space station is going to be. You have been given a simple, unparallelised code which is able to perform the calculation by domain decomposition (though it is not currently set up to do so). You must therefore work out how to optimise and parallelise the code so as to maximise the speed of your calculation on the available resources, which are 2 nodes (56 cores) of `orac`.

Background and Model

We will model the routers by spheres of radius R, and defined them as connected if the distance between two routers is less than $2R$ (this is the range of a router). We first determine the number of routers which will be found in a sphere of radius S if it contains a volume fraction P of routers:

$$N_{\rm rtr} = P\frac{\frac{4}{3}\pi S^3}{\frac{4}{3}\pi R^3}\,. \tag{1}$$

The spheres associated with many of these routers will overlap: the actual volume fraction occupied by *any* router is $\phi = 1 - {\rm e}^{-P}$ (can you think why?). We choose random positions $\mathbf{r}_i = (x_i, y_i, z_i)$ within the range $\sqrt{x_i^2 + y_i^2 + z_i^2} \le S$ for each router i. Two routers i, j, are defined as "connected" if

$$|\mathbf{r}_i - \mathbf{r_j}| < 2R \tag{2}$$

The program uses a recursive algorithm to find the clusters of connected routers in a given list of routers. However, naively implemented, without any sort of domain decomposition, this algorithm would scale poorly with the number of routers, as $\mathcal{O}(N_{\rm rtr}^2)$ because of all the pair-wise checks between all the routers in the system. Because $N_{\rm rtr} \propto S^3$, this is effectively a $\mathcal{O}(S^6)$ algorithm.

To mitigate this poor scaling, the code is set up to use a domain decomposition to divide the space up into a cartesian grid of $n_x \times n_y \times n_z$ "cells", each containing a subset of the total list of routers. These cells must be larger than $2R$ to ensure that routers can only be connected to neighbouring cells in each direction. Then, we can find connected clusters by first searching for clusters within a cell, then by iteratively merging clusters with those of neighbouring cells until there are no more changes to be made, thus allowing the identification of clusters which span the whole system. This spanning cluster identification requires us to specify a somewhat arbitrary criterion for what counts as 'spanning". In the case of a sphere geometry we choose to require that the cluster contains at least one router that overlaps with the sphere edge in each of the 8 octants of the spherical shell.

[2] `https://en.wikipedia.org/wiki/Percolation_threshold`

Program Structure

Download `wifi.zip` from the course web pages and uncompress into your space on the SCRTP system in the usual way. The code as provided will compile and run with some default parameters that implement a rather slow calculation at several different system sizes and volume fractions. It will report back the parameters it has used, which can be overridden with various command line flags as detailed in the table in the Appendix.

The main routines are in wifi.c. Examine the code and make sure you understand the layout. Three sections of the code are currently timed with `clock()`: the recursive cluster search over the routers of each cell; the iterative merging of clusters in adjacent cells; and the final analysis of the resulting clusters to determine if any of them span the system. Each of these may take a significant or insignificant amount of time depending on the settings used.

The code is currently set up with default values to produce runs that are quick enough to be run interactively a few times (eg on `stan1` or your own machine) to familiarise yourself with how it works. As soon as you modify the code or settings in ways that might involve longer runtimes, you should work on `orac` by submitting the code with the supplied job script, which sends jobs to the `PX425` reservation. This queue reserves for 3 nodes until January. You should not have to wait long for jobs to run in this reservation: please get in touch with the module leader if the queues become problematic.

First Steps

Here are some investigations you can perform to ensure you understand the code and its results before you try to parallelise it. Consult the Appendix for full details of input flags, and make notes on the results of all these tests in your report.

1) Run the code with its default parameters: this will trigger 5 runs of the code, for spherical space stations each with an increasing value of S from 20 to 40 in steps of 5, at a volume fraction $P = 0.34$. These runs should produce no spanning clusters, and will report 3 timings as discussed above for each run. The number of cells in the domain decomposition has been set to 1 in each direction, so there will be $\mathcal{O}(S^6)$ scaling of the time to find clusters recursively, and no time required to merge them. Searching for spanning clusters should be very quick at these small system sizes. Plot the time required for the recursive cluster search against S and verify it scales as S^6 with an appropriate fit.

2) Modify the parameters `cellmin` and `cellmax` defined near the start of the `main()` routine so that the code is tested again for domain decompositions ranging from 5 to 20 cells. Plot timings for an appropriate selection of domain decompositions, and discuss briefly in your report how the different timings vary for different sizes and domain decompositions. What are the fastest domain decompositions at $S = 20$ and $S = 40$ respectively? Can you explain any variation?

3) Now fix `cellmin` and `cellmax` at 20. You now need to set values of `P`, `Q`, `S`, `D`, `N` and `M` such that the code runs five times at each of 8 values of P, ranging from 0.300 to 0.405 in steps of 0.015, all for *fixed* `S`$= 40$. Note in your report what happens to the average number of clusters and the probability of a spanning cluster, as P rises. You

can repeat this with different random seeds by using the `-t` flag to initialise the random number generator with a time-based seed. Where does it start to become likely that the cluster spans the system? Compare this to the literature value if you have found one, and discuss under what limits the value of volume fraction P would agree with the literature result.

Once you are happy you have understood the main factors determining the behaviour of the algorithm and the input variables, proceed to the next section.

Parallel implementation

You may use OpenMP, MPI, or a combination of the two to parallelise your code. You are welcome to try CUDA acceleration (in which case use the `PX425gpu` reservation on `orac`) if you can think of a way to GPU-accelerate this code efficiently, but this is likely to be rather harder than using standard parallelism and may not work well. Ideally your code should be capable of running on any number of processors, although a pure OpenMP code will be restricted to a maximum of 28 on `orac` - it may be quite difficult to get good scaling to this level anyway, so do not try to run before you can walk. You should try to parallelise as much of the code as produces tangible speedups. I would prefer you to avoid unnecessary changes to the code that generates the routers, so that it produces reproducible random numbers for the purposes of testing your code.

Some requirements and suggestions follow:

- Subject to being run with a user-chosen number or threads or processes (or both), the parallel code must be capable of determining its own appropriate "parallel strategy", and must not require significant user-intervention on a size-dependent basis to achieve this.
- You may wish to modify the domain decomposition strategy, based on the system size or parallelisation strategy or both. Ensure it does not choose cells smaller than $2R$.
- The formatting of output to the screen should match the serial code. This will likely mean ensuring that only one thread/task calls `printf`.
- The parallel code should give the *same result for whether there is a spanning cluster* on any number of processors/threads, including a single processor/thread. The number of this spanning cluster is arbitrary and may well vary based on parallelisation and domain decomposition strategies.
- You will be penalised for writing code that could result in a *race condition* or a *deadlock* situation, even if in practice they do not seem to occur very often.
- You may change any code you wish within `wifi.c` to achieve this, subject to the above conditions. You do not need to touch the random number generator, but you must ensure that different processors are using statistically-independent random numbers and that the RNG is called in a thread-safe way.

- You *must* appropriately adjust the statements that time your code to use the appropriate parallel routine, `omp_get_wtime()` or `MPI_Wtime()` to ensure appropriate timing in parallel (`clock()` will measure OpenMP codes completely wrongly!). Consider carefully the effect of synchronisation and load-balance on your timings, and discuss in your report what measures you have taken to ensure accurate timing.

Do not run on `godzilla` *with anything other than the initial version of the code for small systems, and never run your code directly on the* `orac` *login node. This could result in suspension of your SCRTP accounts.* You must submit your parallel jobs via the `PX425` reservation on `orac`, though if you have access to other suitable multi-core machines you are welcome to test and develop code there. MPI and OpenMP example scripts for use with the `PX425` reservation is provided. They will need modifying for your choice of parallel strategy. Note that if you are passing command line arguments to an mpi executable via `srun`, they will need to come after the executable name otherwise they will be interpreted as arguments to `srun`.

Your report should detail the parallelism strategy you've chosen, giving reasons why you think this is appropriate/efficient. Describe any problems you've had implementing the simulation in parallel. If you try strategies that do not work, feel free to mention them in the report as you may get some credit for the ideas.

Performance metrics and Report

Using timings gathered from your final program running on `orac`, measure both the speedup ψ and the Karp-Flatt metric F (experimentally determined serial fraction) as a function of the number of OpenMP/CUDA threads, MPI tasks or products thereof if you have written a hybrid code. Plot graphs of ψ and F, as a function of total number of processes, with separate lines for different OpenMP/MPI balances if you choose to use hybrid parallelism. Make sure all graphs are clearly labelled with the system sizes and volume fractions used, and test at several different widely spaced system sizes. Run at a P value near where you believe the percolation threshold to be: I do not expect a precise value as that requires significant data collection — 3 s.f.'s is plenty. What system sizes are appropriate will depend on how efficiently you implement the parallelism: I suggest at least $S = 100$ as a starting point, and up to $S = 300$ if your code can go that large within the maximum walltime of 1 minute of the `PX425` reservation.

Your report should *detail the parallelism strategy you've chosen*, giving reasons why you think this is appropriate. If it has not turned out to be efficient under all circumstances, that is fine, but you should understand and discuss the reasons why not in your report, with reference to the performance metrics requested below.

Discuss these plots in your report. Specifically, determine from your Karp-Flatt data if the strong scaling of your code is likely to be limited by sections which have not been parallelised, or by parallel overheads such as inter-processor communication and thread management. Your conclusions may depend on the system size.

As part of the assessment of your code, it will be run with a selection of different combinations of the flags described in the Appendix, on varying numbers of nodes: you should ensure that you get good performance from as many combinations of these as you can. Describe carefully in your report what steps you have taken to ensure good performance across a range of system sizes.

Deadline and Submission

You have until mid day on Friday 7th January to complete the assignment. Compress all files needed to compile and run the program (including a script), and your report, into a single zip file with the same name as your ITS user code, e.g.

```
zip phuwxyz.zip assignment5/*
```

where you use your user code instead of `phuwxyz` and I have assumed all your files are in the directory `assignment5`. Please *test* that your final zip file contains everything it requires and can be successfully unzipped and then compiles using just `make`.

Then upload the zip file to the Moodle page for Assignment 5:

- `https://moodle.warwick.ac.uk/mod/assign/view.php?id=1419808!`

.

Appendix: Command Line Arguments

These are the command line arguments accepted by the program:

Argument	Purpose	Syntax & Example

`-t`	If present, uses current time as random seed	`-t`
		`./wifi -t`

`-P`	Sets volume fraction starting value	`-P <val>`
		`./wifi -P 0.50`

`-S`	Sets space station size starting value	`-S <val>`
		`./wifi -S 120`

`-R`	Sets radius of each router (default 1)	`-R <val>`
		`./wifi -R 1.5`

`-D`	Sets increment of space station size (default 10)	`-D <val>`
		`./wifi -D 5`

`-Q`	Sets increment of volume fraction	`-Q <val>`
		`./wifi -Q 0.01`

`-N`	Sets number of increments of space station size	`-N <val>`
		`./wifi -N 5`

`-M`	Sets number of increments of volume fraction	`-M <val>`
		`./wifi -M 8`

Note, by setting `Q` to zero and using nonzero `M`, each value of `S` is re-run `M` times with the same `P`. Likewise, by setting `D` to zero and using nonzero `N`, each value of `P` is re-run `N` times with the same `S`.

Examples:

- To generate 11 values of length between $S = 50$ and $S = 150$ at $P = 0.343$:
  ```
  ./wifi -S 50 -D 10 -N 11 -P 0.343
  ```
- To generate 9 values of P between $P = 0.330$ and $P = 0.350$ at $S = 60$:
  ```
  ./wifi -S 60 -P 0.330 -Q 0.0125 -M 9
  ```
- To repeat the above sweep of P values, but with 6 repeat trials each time (fixed S) with time-dependent seed:
  ```
  ./wifi -S 60 -P 0.340 -Q 0.0025 -M 9 -D 0 -N 6 -t
  ```