HPC Assignment 1:

Serial optimisations & OpenMP parallelism

In this assignment, apply serial optimisations and then use OpenMP to parallelise the lattice Boltzmann code

40%

Assignment description

- Start with the provided serial lattice Boltzmann code
  - https://github.com/UoB-HPC/advanced-hpc-lbm

- ***First***, apply serial optimisation to improve the time on 1 core
- ***Second,*** apply OpenMP to parallelise to run your code from 1 core, up to all cores of one node of BCp4
- A 4-page report is required

Ballpark times in seconds

Optimised serial	Optimised serial + vectorised	OpenMP (28 core)
128x128: 22s	128x128: 6.2s	128x128: 0.9s
256x256: 178s	256x256: 48s	256x256: 3.1s
1024x1024: 730s	1024x1024: 269s	1024x1024: 19.0s

Coursework submission

- Your submission will be made via Blackboard and should include:
  - A **4 page** report in PDF form, which must include:
    - Your name and user id
    - A description of your serial optimisations and OpenMP parallelism;
    - Comparisons of your OpenMP performance vs optimised serial;
    - An analysis of the scalability of your code from one core up to all cores in one node;
  - The working code you used to generate the results in your report.

- Your code must converge to the same solutions as the starting code, after the same number of iterations (within a reasonable tolerance).

Marking scheme – 50%+

- To achieve a mark of 50%+, you will need:
  - A reasonable 4 page report that demonstrates you have a some understanding of the main issues
  - Optimised serial code that works, and an OpenMP version that's faster in parallel than your optimised serial version
- Your time on all 28 cores of Blue Crystal phase 4 should be consistent with the table on page 2 of this assignment

- You should be able to get a serial optimised version of the code working in a day, and a simple OpenMP version working in one more day (~2 x 7-8 hours)

Marking scheme – 60%+

- To achieve a good mark of 60%+, you will need:
  - A well-written, 4 page report that clearly demonstrates you have a good understanding of the main issues
  - Include an analysis of parallel scaling from 1 to 28 OpenMP threads
  - Optimised serial code and parallel OpenMP code that use most of the techniques we've shown you, and that at least match the ballpark timings

- This should take about 1 day longer than the simple serial optimised and OpenMP version required for 50%+

Marking scheme – 70%+

- To aim for a first (70%+), you’ll need:
  - An excellent 4 page report
    - Includes more detailed analysis, such as correctly applied roofline or similar
  - Code that:
    - Applies further OpenMP techniques beyond those we’ve shown you, that improve performance further
    - Achieves performance on all the cores notably faster than the given ballpark times
- There are ~6 weeks allocated to the OpenMP assignment, and 20 hours allocated to the course each week for 10 weeks, with 4 hours per week spent in lectures and labs. You don’t need to spend more than 3-4 days on this assignment in total; use the rest of your time to explore the HPC space, play with new tools like profilers, read around etc.
  - It should only take a couple of days to do the simpler version which, along with some interesting experiments and a decent report, should be good enough to earn 60%+

Submission specification

- Your **report** which must be in a file called “**report.pdf**”,
  - Lower case r: “**report.pdf**” NOT “**Report.pdf**”
- Your **source code**, i.e. “**d2q9-bgk.c**”
- Your **makefile**, called “**Makefile**”
- **Don’t** modify the timing code in the starting code, as we’ll use this to automatically extract timing information from each submission
- We must be able to reproduce any runtimes you quote in your report by compiling and running the code that you submit
- Submit all these files as a single zip file (not RAR, not tar, …), i.e. do no submit them as separate files on BlackBoard
  - Do not include any directories for the base submission
  - For extensions, add a single directory for each one, in addition to your base submission. Include a README file for each directory explaining its purpose and how to compile and run it

Submission specification

- Include a Makefile where the default target, i.e. typing “make” in the command line, builds the code that you want marked
  - Make sure you don’t accidentally name your source code files differently than the Makefile expects
- Name your produced executable “d2q9-bgk”
- Include an “env.sh” file containing ***only*** the module commands needed for your compiler/libraries choice
  - Don’t add any other commands or text
  - If a module is not loaded here, it won’t be available when your program is compiled and run, regardless of the default environment settings on your own account (e.g. .bashrc)
  - Be careful with modules that interact with other modules, e.g. behaviour of icc depending on what version of gcc is available (this has caught us out before!)
- Don’t change the command line parameters of the application in any way, and
- Don’t change the output of the application in any way
  - This is because the auto marker relies on passing in the parameters, and comparing the output against what it’s expecting

Things to avoid!

- Invalid commands in env.sh: -1
- Missing env.sh: -3
- Invalid Makefile syntax (line endings or spaces): -1
- Missing build files or otherwise broken build: -3
- Minor validation issue (~1 failure): -1
- Major validation issue (3+ failures): -3
- Catastrophic (almost complete) validation failure: -5
- No name on report: -1
- Altered program output: -1

Hints and tips – serial optimisation

- We recommend you apply serial optimisations first, don’t jump straight to OpenMP
- You should be able to hit the performance goal by:
  - Choosing a good compiler and optimisation flags
  - Applying appropriate loop fusion (you should only need to iterate over the whole grid once per timestep)
  - Removing unnecessary copying between tmp_cells[] and cells[] ("pointer swap”)
  - Manually improving the arithmetic in the loops to help the compiler optimise it more easily
  - Use the profiler to see how you’re getting on
- With these steps, and leaving the data layout in the original "array of structures (AoS)" format, you should be able to match or exceed the ballpark times. This is good enough for a reasonable OpenMP implementation - with nice scaling and a well-written report, this can get you into the 60+% range.

Hints and tips – vectorization part 1

- Enabling efficient vectorisation of your combined inner loop will require you to help the compiler as there's a lot of code to vectorise here. However, if you can get it working, the benefits are clear from the speedups we're indicating
- Things to consider when vectorising include:
  - Setting the right compiler flags (e.g. -xAVX for the Intel compiler)
  - Making sure you've used "restrict" and "const" everywhere you can, so that the compiler knows that your pointers to cells[] and tmp_cells[] won't alias (overlap) in the inner loop
  - Changing the data layout to suit vectorising the inner loop. Right now the array of structures (AoS) suits non-vectorised scalar code. To vectorise across iterations of the inner loop, you really want a structure of arrays (SoA) layout, with all of the speeds[0] in a single array etc.
  - NOTE that when you change your scalar code from AoS to SoA we would expect it to get slightly slower, until you then also get vectorisation working. This is because the AoS layout has better data locality for the scalar, non-vectorised version of the code (once you access speeds[0] you've also brought in speeds[1]-speeds[15] to the cache due to the 64-byte cache lines). Once you've made the AoS->SoA transformation and got vectorisation working properly, the code should once again be faster than the scalar AoS version, by quite some margin.

Hints and tips – vectorization part 2

- If you want to get vectorisation working, then to help the compiler vectorise your code, you may need to consider techniques such as:
  - **_mm_malloc(num_bytes, alignment)** : a variant of malloc() which returns data aligned on the requested boundary (64 bytes is a good choice, to match the length of a cache line)
  - **__assume_aligned(pointer, alignment)** : tells the compiler a pointer is aligned to a certain boundary (good idea to match whatever you used with _mm_malloc). Useful to have these inside routines about to use certain pointers for array access inside important inner loops, otherwise the compiler won't know that our data is aligned
  - **__assume()** : a good way to tell the compiler something useful about loop counters, to aid it in producing more efficient loop unrolling for vectorisation. Repeatedly using __assume() to tell the compiler multiple things about the same counter may also help; for example, whether a loop counter is exactly divisible by several different powers of 2 (__assume((params.nx)%2==0) and so on.)
  - Using "**#pragma omp simd**" on the loop you really want vectorised may help the compiler further (this is highly dependent on whatever else you've done up to this point)
- A really good webpage to read for help with advanced vectorisation is on the Intel website.

How the automarker works

1. Starts with a clean environment
   - See: https://github.com/UoB-HPC/hpc-course-getting-started/blob/master/2_Modules.md#unloading-modules
2. Runs "**source env.sh**" to load your chosen modules
3. Runs "**make**" to build your application and check that d2q9-bgk has been produced
4. Runs the application: **"./d2q9-bgk input.params obstacles.dat**"
5. Runs the checking script

If any of the steps above results in an error, a failure is reported at that point. Otherwise, the time is recorded. We recommend you go through these steps manually with your submission, e.g. by redownloading your submitted zip file, to avoid any surprises with the automarker.

Plagiarism checking

- The HPC assignments are all for individuals, they are **not** group work
- We will check **all** submitted code for plagiarism using the MOSS online tool
  - MOSS ignores the example code we give you
  - MOSS will spot if any of you have worked together or shared code, so **please don't!**
- We'll also check all submitted reports using the TurnItIn tool, which will find any shared text
- So please don't copy code or text from each other! You **will** get caught, and then ***both*** the copier and original provider will get a **0** for the whole assignment.

Summary

Remember, you'll get marks for:

- A well written, comprehensive, report
- An OpenMP code that successfully explores most of the optimisations we suggest

Have fun exploring OpenMP parallelism and the LBM code!