HPC - OpenMPI

Callum Mann(cm13558)

December 18, 2015

1 Introduction

OpenMPI emphasises the distribution of computation across nodes and their processors and promotes extreme scalability of code. This report describes the efforts to optimise the Lattice Boltzmann code to run as fast as possible with OpenMPI, drawing upon previous work in OpenCL and OpenMP to provide a relative benchmark for performance.

2 Overview

Optimisations found during the OpenMP and OpenCL work provided a fast underlying codebase to begin building the OpenMPI design upon. Having these optimisations from the beginning allowed for more in depth analysis of the core principles involved with MPI. The optimisations are briefly summarised as:

- Ignoring all non-edge obstacles completely (no iteration)
- Change of memory structure to increase speed of reads and stores
- Scattered reads rather than scattered stores

The previous fastest times for MP and CL were 30s and 4.8s for the *large_pipe* test case. The final fastest time for MPI was ≈ **0.98**s (4 nodes, 16 ppn.).

3 OpenMPI Design

The MPI design is heavily influenced by the ability to quickly test ideas in the code by using other processes to reconstruct the MPI code.

3.1 Meta-programming with PyMPI

Rather than writing pure C code to run LBM, instead a Python precompiler (named *PyMPI*) was designed to effectively bootstrap and enhance the ease of design change when it was required. In addition to this, the precompiler allows all processes to be compiled and suited specifically to the memory that they are chosen to work with. In this way, when the program is run with `mpirun`, we are actually launching several distinct binaries that are still part of `MPI_COMM_WORLD`, but actually hold different code inside them, precompiled to reduce unnecessary computation in their allocated area. In MPI this is known as MPMD (Multiple Program Multiple Data).

Specific optimisations that can be applied include unrolling the acceleration loop entirely so that no branches are taken, and more importantly less memory access to obstacles. This effect is contrasted with vectorization of loops later in the report.

3.2 Region splits

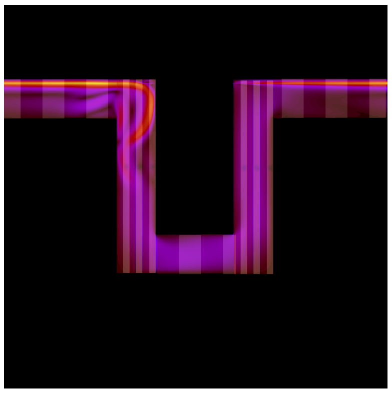

Figure 1: Rectangle slicing for rank regions

The memory layout is important specifically in MPI for the efficiency of the exchange of speeds inside the halo exchange, which will be discussed in the next section. Each rank in `MPI_COMM_WORLD` is given a region of the total simulation that averages total computation that must be done on each rank. One such solution is to distribute regions to ranks in a rectangular fashion, such that the region is easily iterable over by that rank. In addition, the region should only contain *active* cells and *edge cells*, so that there is no needless iteration over cells that need not be updated, as shown in Fig. 1, where the rectangles are grown by claiming adjacent columns. To maximize overall speed, the algorithm attempts to select rectangles that contain the same amount of cells, so that on average, all of the ranks will synchronize halos at the same time. However, it is also greedy, so each rank will take *above* the amount of cells it should take on average, to comply with the rectangle constraint. A side effect of this constraint is that there are a minimum number of processors required, otherwise parts of the space are not claimed. In this case the region is not split into rectangles (this was 8 processors for small/large pipe)

Memory layouts can be completely modified with a C macro that determines the positions of speeds in memory, in the case of the verticle layout (using SoA), the macro is stated as:

$$\texttt{spd}(j, x, y, w, h) = (j * w * h) + (h * x) + y \quad (1)$$

4 Communication

MPI offers a range of communication methods that can be adapted to suit the problem being solved. For LBM, a *halo exchange* design has been chosen. The halo exchange is explained simply as an extra border around the iterable region, acting as a buffer for the speeds from neighbouring regions. The speeds that are sent in by the neighbours are placed into these buffers where they will sit until the rank uses their values. When the memory is mapped vertically and halos are striped vertically, this allows for a very simple halo exchange, where the section of particular speeds are simply sent to the other rank, since they are contiguous. The halos themselves are essentially only used as a temporary storage for the speeds until the rank *pull propagates* them in.

4.1 Ease vs. Scalability

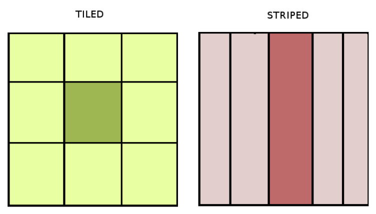

Figure 2: Tiled and Striped halo arrangements

Deciding upon the most efficient way to arrange the halos really depends on the properties of the case in hand. Fig. 2 shows two different halo arrangements. For very large cases that require huge amounts of memory, the striping approach could cause problems where the ranks do not have enough memory to contain the region that they have been allocated. The potential drawback here is that there is a communication overhead, since every tile must communicate with all adjacent tiles. It is immediately noticable that the tiled scheme may lose efficiency due to some sides of the tile being able to recieve contiguous speeds, while the other sides cannot. However, tests in the further sections would indicate that this overhead may be negligible.

4.2 Synchronous vs. Asynchronous

Syncronisation poses the problem that all ranks must send and recieve in a respectful manner so that no two ranks are trying to conduct the same operation (send or recieve) with each other at the same time. Even when the ranks respect each other, they are time inefficient because they must pause all computation to wait entirely for data to arrive.

4.2.1 MPI Window

An MPI window is another form of communication where a rank can open a "window" of memory from which other ranks can load and store data. The main difference with this method is that the memory can be accessed remotely, bypassing the need to call the kernel and reducing these overheads (given that there exists an appropriate inter-connect medium, e.g Infiniband).

When using this method, speed for large pipe fell from 1.75s to 1.6s.

4.2.2 Asynchronous

Asynchronous communications allow the ranks to continue computing while their data is being sent to their neighbours, making it an ideal choice to remove communication overheads.

When using asynchronous communications, the time for *large_pipe* dropped from 1.6s to $\approx$ 1.47s, a difference of 9%. While this is an increase in speed, it is not significant enough for us to conclude that communication is the main limiting factor. In fact, when removing all synchronization the resulting speed was only 1.3s, suggesting that the problem is memory or computationally bound. This again, is in favour of the tiling method for becoming more optimal as the problem space grows.

5 OpenMP within MPI

Since it is fairly trivial to combine OpenMP with MPI, it was worth testing the combination of the two. When using OpenMP across 4 nodes with varying number of processes per node, on average it was found that OpenMP was no faster, or worse. This can be explained because of a few reasons:

1. Every timestep there is a startup cost for all of the threads
2. Additional barriers occur at the end of the loop, causing synchronisation and slowdown between uneven areas in the iterable space

6 Compilation optimisations

6.1 Loop vectorization vs. Loop unrolling

The two main loops in the MPI code that can be effectively vectorized, one being the accelerate step, and the other being the combination of the other steps. There is a decision to be made in the precompilation process on whether to vectorize a particular loop, or to unroll it into several different portions of code in the effort to remove overheads such as unnecessary memory reads and branches. Branches in particular are worth considering for optimisations, as with powerful nodes with deep pipelines the cost of a mispredicted branch can be costly[1].

In particular, the unrolling of the accelerate flow across the processes yielded a small increase $\approx$ 0.05 in speed over the loop vectorised equivalent. It's probable that the unrolled code was vectorized as well, at a lesser cost than when in the loop.

When using the Intel compiler and applying a SIMD pragma to the second computation loop, the code increased by 0.2s from 1.45 to 1.25, which is a speedup of 14%. The vectorisation required the knowledge that it must respect the reduction of *tot_u*, so that it did not force the reduction in such a way that the real computation is lost.

6.1.1 Fast math

One final speedup was to simply compile the processes with `-Ofast`, which turns on all of the aggressive, unsafe compiler optimisations. This allows the compiler to shift the float values around between instructions, so that they can possibly be optimised into other expressions. This pushed the large pipe speed from 1.25 to 0.98, which is a 22% increase in speed.

7 Timings

Test case	OpenMP	OpenCL	OpenMPI
Small pipe	0.9494s	0.28s	0.086s
Box	5.436s	1.63s	0.72s
Rect	9.45s	2.49s	1.02s
Large pipe	30s	4.9s	0.98s

Figure 3: Display of the fastest run times for all test cases

Num Processors	1	2	4	8	16	32	64
Large Pipe	105	63	29	13	5	2.83	0.98

Figure 4: Display of MPI scaling from 1 to 64 cores

MPI is clearly the fastest time given the historical data available. Much of the speed difference between the test cases are really due to the optimisation of the problem given rather than an indication of which tool can be faster.

8 Final Remarks

The precompilation route provides access to optimisation not otherwise accessible when compiling normally. However, there are a few drawbacks of the implementation. Firstly, the parameter parsing mechanism is written in Python, which significantly reduces speed in finding edges, calculating halos, etc. On top of this, the code must be repeatedly compiled for the number of desired processes, and for very large test cases this may result in impractically large binaries, due to unrolling of certain loops. To combat this, a further implementation would have port the heavy Python sections to C, and formulate a reasonable cap on the amount of C code that can be precompiled into one file.
Because the precompilation time can be large for test cases like *large_pipe*, a status bar was implemented to see how far it is through the compilation:

```
(echo "" > status) && qsub mpi_submit -v LBM_PARAM=param && (tail -f status 2>/dev/null)
```

9 References

[1] https://software.intel.com/en-us/articles/branch-and-loop-reorganization-to-prevent-mispredicts