ARCHER2-eCSE02-11 eCSE Technical Report

Optimising BOUT++ MPI+OpenMP hybrid performance by refactoring compute kernels

Weronika Filinger (EPCC, The University of Edinburgh), Joseph Parker (United Kingdom Atomic Energy Authority), David Dickinson (University of York), Ben Dudson (Lawrence Livermore National Laboratory), and Peter A Hill (University of York).

Keywords: MPI + OpenMP, hybrid parallelisation, BOUT++ performance, ARCHER2, eCSE

Abstract

The goal of this eCSE project was to investigate the performance of the hybrid MPI+OpenMP implementation of BOUT++ on ARCHER2, improve its scaling to higher processor counts, and start refactoring the code so that there is only one large loop kernel per timestep. Although the code is parallelised with both MPI and OpenMP, the addition of OpenMP initially didn't improve the scalability of the code significantly. The first stage of the project was to investigate this behaviour and find the configuration of MPI processes and OpenMP threads that produces the best performance. The second part of the project was focused on investigating the performance improvements due to the refactoring of the code, which used RAJA to both move the main parallel loop to a higher level in the code structure and improve the portability of the code. RAJA is a parallelization abstraction layer that allows the same source code to be executed on GPUs or with OpenMP on CPUs. This investigation studied the CPU version, which essentially meant applying OpenMP to the main loop kernel. The actual code refactoring wasn't part of this project; however, the work carried out in this project evaluated the performance gains resulting from it, including improved scaling of the hybrid parallelisation strategy.

BOUT++ code

BOUT++[2] is a C++ framework for writing fluid and plasma simulations in curvilinear geometry. It is intended to be quite modular, with a variety of numerical methods and time-integration solvers available. BOUT++ is primarily designed and tested with reduced plasma fluid models in mind, but it can evolve any number of equations, with equations appearing in a readable form. The user-specified equations are separated from the implementations of differential geometry, parallel communications or I/O operations. BOUT++ is organised into classes and groups of functions which operate on them. It is not purely object-oriented, but takes advantage of many of C++'s object-oriented features. Macros are used extensively in the code, including the OpenMP regions.

Data Structures and Parallelisation Strategy

For execution on CPU systems, BOUT++ uses MPI to implement two-dimensional domain decomposition in the X and Y directions. The grid is divided into domains closest to square while ensuring the domains on every processor have the same size and shape

[2] BOUT++ documentation - https://boutproject.github.io/

and the branch cuts (mostly at X-points) are on processor boundaries. The communication between different processors is done primarily through the exchange of guard cells using the mesh communication functions i.e. the MPI functions are embedded into the mesh functions (e.g. mesh::communicate(), mesh::send(), mesh::wait()). The restrictions on the simulation domain require the equilibrium to be axisymmetric (in the Z coordinate), making it impossible to use MPI parallelisation in the Z direction. Instead, OpenMP is used in a large number of places to parallelise the z direction. The OpenMP pragmas are embedded inside the BOUT_FOR macros, which replace the for-loops.

These macros all have the same basic form: an outer loop over blocks of contiguous indices, and an inner loop over the indices themselves e.g. BOUT_FOR(index, region). This allows the outer loop to be parallelised with OpenMP while the inner loop can be vectorised with the CPU's native SIMD instructions. Alternative forms are also provided for loops that must be done serially, or require more control over the OpenMP directives used. The macros are: BOUT_FOR (OpenMP-aware), BOUT_FOR_SERIAL, BOUT_FOR_INNER (for use on loops inside OpenMP parallel regions) and BOUT_FOR_OMP (takes arbitrary OpenMP directives as an extra argument).

The Region class is defined to specify a set of indices which can be iterated over, as well as the begin and end methods for range-based loops. These indices can either be used directly, or blocks of contiguous indices may be used instead (used in the OpenMP regions). There are two main data types used in BOUT++: Field2D and Field3D. They provide a straightforward way to manipulate data by taking care of all memory management and looping over grid-points in algebraic expressions. Two optimisations used in the data objects to speed up code execution are memory recycling, which eliminates allocation and freeing of memory; and copy-on-change, which minimises unnecessary copying of data.

This code structure adds an additional level of abstraction, making it easier for the user to implement their model without worrying too much about the underlying details. At the same time, this makes the code a lot harder to profile and to optimise for parallel performance. The hybrid MPI+OpenMP parallelisation strategy, although providing high parallel coverage, doesn't perform very well compared to MPI only parallelization. One of the goals of this project was to investigate the reason behind that behaviour and suggest possible improvements.

Case studies

Two simple models have been used as case studies in this project. The first one, elm-pb, models edge localised modes (ELMs), which are instabilities driven by pressure gradients and currently at the edge of high performance tokamak plasmas. They result in repetitive bursts which could result in large heat loads on the walls of large tokamaks, and so are a concern for ITER. The elm-pb model included in the BOUT++ repository solves a 4-field model for pressure, vorticity, the parallel component of the plasma velocity and electromagnetic potential. It has been used for studies of ELM eruptions, mainly in circular geometry plasmas.

For most runs the default input settings have been used, including the provided grid files - cbm18_dens8.grid_nx68ny64.nc and cbm18_8_y064_x516_090309.nc. The default grid size in the first file is 68 x 64 x 16 cells. The number of points in the x direction also includes the boundary points, so the internal points for this grid size are 64 x 64 x 16. To observe how the code scales, the following two grid sizes have been used as well - 64 x 64 x 64 (first input file) and 512 x 64 x 64 (second input file).

The second case study is called blob2d. The 'blob' refers to isolated blobs of plasma often observed at the edge of tokamak (and other) plasmas. These blobs are elongated along the magnetic field, forming long filaments that have a surprisingly complicated range of nonlinear behaviour. The blob2D model simulates isothermal blobs with a sheath closure and is one of the basic examples included in the BOUT++ distribution. The following executable arguments have been used to collect the performance data: delta_1, mesh:nx=1028 mesh:nz=1024. The input file delta_1 includes the simulation parameters and is also available in the BOUT++ distribution. The number of time-steps was set to 200 and the grid size was overwritten from 260 x 256 cells to 1028 x 1024 cells (the x direction includes 4 guard cells).

The blob2d-outerloop example solves the same set of equations, with the same inputs, as blob2d. In this version of the code, the loop over the simulated region is placed as high as possible in the code call tree, whereas in the blob2d version the loops over the grid are hidden under the density and vorticity equations - executed in separate sections of the code. This refactoring utilised the RAJA[3] library, which is a software library of C++ abstractions that enables portability across different HPC architectures and programming models. It targets portable parallel loop execution by providing building blocks that extend the parallel for idiom to insulate application loop kernels from underlying architecture and programming model-specific implementation details. When RAJA is not enabled the loop constructs revert to the BOUT loops, which are OpenMP loops when OpenMP is enabled.

Results

Elm-pb - scaling results

Figure 1 shows the scaling of the MPI-only execution of the elm-pb model on 3 different grid sizes. The scaling clearly improves with the increasing grid sizes. The parallel efficiency drops below 50% when the grid size per PE is smaller than 16 x 8 x z, which for the two smaller grid sizes occurs on 32 MPI processes, and on 256 MPI processes for the bigger one. Using a smaller domain per PE incurs more significant communication overheads, making the ratio of computation to communication less favourable. The grid sizes are rather small but typical for these kinds of simulations. Using a larger number of smaller sub-domains will not improve the performance significantly, therefore there is no point in using a larger number of MPI processes. Instead OpenMP has been used to parallelise the sub-grids in the z direction.

To obtain a good performance using this hybrid approach, i.e. using MPI to partition the grid in the x and y direction and OpenMP to parallelise the z direction, it is necessary to pick a good ratio of MPI processes to OpenMP threads. For example, using a small number of MPI processes and a large number of OpenMP threads will not yield a good performance due to the bulk of computation being located in x-y space. There is also no need to go beyond 1 node for the smaller problem sizes.

The ARCHER2 documentation suggests using the --distribution=block:block option to ensure the processes are distributed across nodes and NUMA regions in a block

[3] https://raja.readthedocs.io/en/develop/

fashion. On a single node, the block distribution just ensures more local pinning of the processes to NUMA nodes - i.e. each NUMA region is filled before moving to the next one. For the MPI only execution there was no significant difference between the runs with and without the block distribution specified; however, the execution time for the runs with the block distribution is significantly faster when OpenMP threads are used (Table 1). Placing MPI processes and their OpenMP threads contiguously within one NUMA region before moving on to the next one ensures better data locality and cache utilisation.

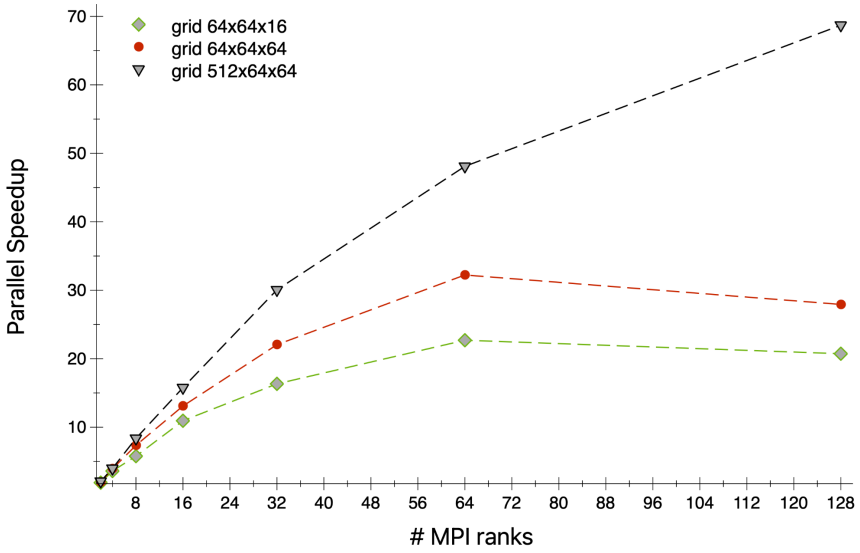

Figure 1 - Parallel Speedup of MPI only execution of the elm-pb model for 3 different grid sizes.

# MPI x OMP	Default distribution	Block distribution
128	29.61	29.54
128 x 1	32.43	31.89
64 x 2	35.37	29.39
32 x 4	32.15	25.35
16 x8	35.64	36.75
8 x 16	71.09	56.89

Table 1 - Run times for different numbers of MPI processes and OpenMP threads executed on a single node with and without the block distribution for the Elm-Pb model with the grid size of 64x64x16 run for 150 steps.

The fastest execution time is observed for the configuration using 32 MPI processes and 4 OpenMP threads. This setting corresponds to the optimal size of the sub-domains and also maps well onto the underlying hardware - using one MPI process per Core CompleX (CCX) and one OpenMP thread per core. Level 1 and level 2 caches are

private to each core, however, level 3 cache is shared between the 4 cores on the CCX. Moreover, L3 is a victim cache. Using 4 threads per MPI process, ensures better cache reuse i.e. one L3 cache per MPI process.

32 MPI x 4 OMP	thread time (s)	L2 cache misses	L2 PREFETCH HIT	L3 PREFETCH HIT	Rq from memory (lines)
Default distribution	32.15	586,158,508	282,819,673	303,162,367	79,853,288
Block distribution	25.35	588,747,878	424,460,596	509,701,382	63,862,692

Table 2 - Cache utilisation for the configurations of 32 MPI processes and 4 OpenMP threads for the elm-pb model - grid size 64x64x16 and run for 150 steps. Executed with and without the block:block distribution.

The combination of 32 MPI processes and 4 OpenMP threads gives one of the highest values of L2 prefetch accuracy (percentage of prefetch hits to all prefetches) and prefetch coverage (percentage of misses avoided due to prefetching i.e. prefetch hits/ (prefetch hits + cache misses)). Other combinations yield one of the values significantly higher than the other. Using the block distribution improved the prefetching coverage by over 10%. Both values are around 42%, signalling that the memory access patterns are perhaps not as optimal as hoped for. Data structures adopted in BOUT++ can be quite complicated. The main objects such as fields were tracked through the source code to ensure they are initialised correctly when OpenMP is used. Overall, the fields and other multidimensional arrays seem to be initialised inside OpenMP regions ensuring the correct affinity between the right segments of data for each thread - assuming the static schedule is used.

```
   type     max_buf[B]         visits time[s] time[%] time/visit[us]  region
    ALL 4,529,605,237 3,095,069,982 6960.09   100.0           2.25  ALL
    OMP 4,312,121,836 2,981,856,942 5305.19    76.2           1.78  OMP
    MPI   215,770,532   111,104,912  420.87     6.0           3.79  MPI
    USR       856,414     1,054,048    0.23     0.0           0.22  USR
    COM       856,414     1,054,048   14.69     0.2          13.93  COM
SCOREP            41            32 1219.11    17.5    38097124.33  SCOREP
```

Figure 2 - Overview of the ScoreP profile for the elm-pb mode executed on 32 MPI processes, using 4 OpenMP threads per processes - grid size 64x64x16 and 300 steps.

One of the reasons for the poor performance of the hybrid MPI-OpenMP implementation, besides the relatively small problem size, is the fragmented parallel coverage of the OpenMP regions. Many of the OpenMP regions are small and executed millions of times, even during short runs, significantly contributing to the parallel

overheads. Figure 2 shows an overview of the ScoreP profile for the best performing combination of 32 MPI processes and 4 OpenMP threads. It is clear that most of the time is spent in small OpenMP regions executed almost 3 billion times. To collate those small OpenMP regions and to prepare the code for future and more complex architectures RAJA has been used to refactor the code.

Blob2d - Refactored Code

As described in the earlier section, the blob2d-outerloop model should solve the same set of equations as blob2d when the same input parameters are used. Table 3 shows the runtimes for both code versions run for 200 steps using different numbers of MPI processes and OpenMP threads to fill a single node. The performance gain increases with the increasing number of OpenMP threads, with the fastest run still observed for the combination of 32 MPI processes and 4 OpenMP threads. The refactoring of the code had a significant impact on the memory access patterns and the overall number of operations. To better understand how the two codes differ, the PAPI hardware performance counters have been collected using CrayPAT via the PAT_RT_PERFCRT environmental variable, some of which are presented in Table 4.

MPI x OMP	blob2d	blob2d-outerloop	Relative Speedup
128 x 1	229s	83s	~2.8
32 x 4	200s	67s	~3
16 x 8	330 s	92s	~3.6

Table 3 - The runtimes for the blob2d and blob2d-outerloop examples run for 200 steps on a different number combination of MPI processes and OpenMP threads.

Event	Blob2d	Blob2d-outerloop
PAPI_TLB_DM (TLB data misses)	0.016G/sec 2,902,983,742.500 misses	0.008G/sec 566,396,378.719 misses
PAPI_TLB_IM (TLB instruction misses)	0.686M/sec 121,183,749.656 misses	0.002G/sec 112,647,635.125 misses
Write Memory Traffic GBytes	0.234G/sec 45.63 GB	0.111G/sec 8.13 GB
Read Memory Traffic GBytes	0.979G/sec 190.65 GB	0.733G/sec 53.75 GB
PAPI_L2_ICH (L2 instruction cache hits)	0.011G/sec 1,970,136,143.969 hits	0.024G/sec 1,577,067,719.000 hits

Event	Blob2d	Blob2d-outerloop
PAPI_FP_OPS MFLOPS (aggregate)	0.019G/sec 3,885,146,242 ops 592.69M/sec	0.006G/sec 461,588,495 ops 206.29M/sec
PAPI_FP_INS Floating-point instructions	1.961G/sec 407,467,258,215.062 ops	1.421G/sec 109,058,915,578.156 ops

Table 4 - A selection of PAPI events collected for both the blob2d and blob2d-outerloop examples, run for 200 steps using 32 MPI processes x 4 OpenMP threads.

The most significant difference between the two code versions can be observed for the Translation Lookaside Buffer (TLB) data misses. TLB assists the load and store address translations, therefore, a TLB miss is associated with the increased number of read and write operations to the main memory, thus having a negative impact on the performance. The outerloop version has over 5 times fewer TLB data misses overall and 2 times fewer per second. This difference is also reflected in the increases in the read and write memory traffic seen. The significant reduction in the TLB data misses in the outerloop is due to the rearrangement of the data access patterns. Looping over the grid once at a higher level ensures better reuse of the entries stored in the TLB, and thus reducing the misses. The total number of instruction TLB misses are comparable to those observed for the original version of the code, but there are more than 2 times fewer of them per second.

Although the number of L1 cache accesses is lower in the outerloop version, this is because there are fewer instructions performed overall, and the rate per second is actually higher compared to that observed in the original version of the code. The significant difference in the rates per second can be observed for the L2 instruction cache hits and misses. In the outerloop version the number of L2 instruction cache hits per second is over 2 times higher, and the number of the L2 instruction cache misses is much lower. The rates of the L2 data reads are very similar for both versions of the code. This confirms that the code refactoring improved the data locality. All calculations for each grid cell are performed at once, rather than spread throughout the code e.g. each component of the equation calculated separately.

The difference between the total number of branch instructions in two versions of the code comes mostly from moving the loop over the simulated region higher up in the code hierarchy, and thus reducing the number of loops over the grid. Replacing two lower level *for* loops with a single *for* loop, cuts the number of branch instructions by half. However, those are the easy to predict branches. In most cases the branch at the end of a loop will be taken - until the loop's terminal condition is reached. The number of branches taken in the outerloop version of the code is ~2.5 times lower, the total number of branch instructions is about 2.8 times lower and there are over 2.1 times fewer branches mispredicted. For both code versions not even 0.4% of the taken branches were

mispredicted. Reducing the number of branch instructions had a positive effect of reducing the execution time, but easily predictable branches usually take only 1 cycle anyway. The total number of floating-point operations is over 8 times lower in the outerloop version and the total number of floating-point instructions is about 4 times lower.

Table 5 shows some of the collected HW counters for different numbers of OpenMP threads per MPI processes: 128 MPI x 1 OMP, 32 MPI x 4 OMP, 16 MPI x 8 OMP and 8 MPI x 16 OMP. Overall, the number of TLB misses is lower when fewer MPI processes are used, however, the number of L2 data cache hits and the number of instructions per cycle is the highest for the combination of 32 MPI processes and 4 OpenMP threads. Using more OpenMP threads means increasing the risk of false sharing, and using 4 threads per MPI processes means confining the threads belonging to each process to single CXX on ARCHER2, resulting in the best memory re-use.

	128 MPI x 1 OMP	32 MPI x 4 OMP	16 MPI x 8 OMP	8 MPI x 16 OMP
Execution time	85.2 secs	63.5 secs	90.8 secs	143.5 secs
TLB_DM	0.012G/sec	0.008G/sec	0.007G/sec	0.005G/sec
TLB_IM	0.002G/sec 136,423,798.969 misses	0.002G/sec 112,647,635.125 misses	0.001G/sec 110,034,465.125 misses	0.815M/sec 117,019,893.375 misses
TOT_INS	3.650G/sec	4.571G/sec	4.778G/sec	4.545G/sec
TOT_CYC Instructions per cycle	1.36 inst/cycle	1.57 inst/cycle	1.49 inst/cycle	1.41 inst/cycle
L2_DCH	0.056G/sec	0.073G/sec	0.057G/sec	0.048G/sec
L2_ICH	0.033G/sec	0.024G/sec	0.015G/sec	0.009G/sec

Table 5 - Selected performance HW counters collected for the blob2d-outerloop model executed on a range of different MPIxOpenMP configurations.

Conclusions

Simply using the hybrid MPI + OpenMP parallelisation doesn't necessarily provide a good scaling behaviour. It is often necessary to play with the settings to find the optimal run configuration for a use case of interest. The best setting on ARCHER2 for both models (elm-pb and blob2d) and code versions has been observed by using the combination of 32 MPI processes and 4 OpenMP threads. Using 4 OpenMP threads per MPI process

allows for efficient use of the cache memory available on the system. The refactored code shows better scaling and benefits more from the hybrid parallelisation.

The initial poor scaling of the hybrid MPI and OpenMP parallelisation was due to a combination of factors, including the adopted domain decomposition (MPI used in the x and y directions and OpenMP used in the z direction), relatively small problem size (typical in the field) and extremely fragmented OpenMP coverage. The adopted data structures are correctly initialised across OpenMP threads so the memory locality is taken into consideration.

The refactoring of the code to use only one work kernel loop per timestep proved to be very beneficial to the performance. The code is significantly faster for the tested case study. This is both due to the reduced number of operations and better reuse of the memory, reflected in a smaller number of TLB and cache misses and higher rate of instructions executed per cycle. Although only blob2d model has been tested, it is expected that other models implemented in BOUT++ would benefit from the refactoring as well. Using only one work kernel per timestep should result in better memory use for all models.

Acknowledgments

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk).