Optimisation of VAMPIRE for billion-atom simulations of magnetic materials on ARCHER

eCSE 0709 Technical Report

Rory Pond and Richard F L Evans

Department of Physics, University of York, Heslington, York, YO10 5DD

Abstract

We have successfully ported the open source VAMPIRE software package to the UK ARCHER supercomputing service. During the project we implemented major changes to the data input and output routines in the code removing the previous bottlenecks of generating snapshots of the atomic scale magnetic configuration from a simulation. The new output code utilizes the full capabilities of the parallel file system on ARCHER achieving effective output bandwidths in excess of 30 GB/s. We have demonstrated the new capabilities of the code by simulating ultrafast magnetic domain wall dynamics in a system of over 918,000,000 Fe and Gd atoms.

Introduction

Magnetic materials are essential to a wide range of technologies, from data storage to cancer treatment to permanent magnets used in wind generators. New developments in magnetic materials promise huge increases in performance of devices but progress is limited by our understanding of magnetic properties at the atomic scale. Atomistic spin dynamics simulations[1] provide a natural way to study magnetic processes on the nanoscale, treating each atom as possessing a localised spin magnetic moment. The localised nature of the spins allows the simulation of a range of complex physical phenomena such as phase transitions, laser heating, antiferromagnet dynamics in complex systems such as nanoparticles, surfaces and interfaces. However, such approaches are computationally expensive, requiring parallel computers to perform simulations of more than a few thousand atoms. The VAMPIRE code [1,2] is an open source software package to perform parallel atomistic simulations of magnetic materials. The code is written in a mixture of functional and object oriented C++ with a modular structure to allow new features to be added to the code.

The aims of this eCSE project were to optimise the VAMPIRE code on the ARCHER system and improve the data input/output routines to enable configuration data to be extracted from the simulation to see the time evolution of the atomic spin configuration in time.

Porting to ARCHER and compiler optimisation

The first stage of the project was to compile the VAMPIRE code on ARCHER and compare performance of Cray, Intel and GNU compilers. The code is written in standard compliant C++ with no external libraries other the MPI and required no changes to compile correctly with the different compilers. The code was also verified and produced identical results independent of the compiler suite used and level of optimization. The results of compiler optimization found that for optimal settings the GNU g++ compiler was around 17% faster than Cray and 10% faster than Intel for small problem sizes, with the difference narrowing to 4% for both for larger problem sizes. The code is not particularly well vectorised, perhaps suggesting that the Intel and Cray compilers slightly over-optimise. This reflects our previous experience when comparing other compilers, also finding that g++ generates optimal performance. To make it as straightforward as possible to compile the code on ARCHER we have added a tuned make target in the makefile so that loading the GNU compiler environment and typing `make archer-parallel` is all that is required to generate an optimal binary on the ARCHER system. Compilation takes around 1 minute using parallel make.

Previous implementation of data input and output

Prior to this eCSE project the output was naively implemented as a one-file-per-process using C++ streams as implemented in the C++ standard library. Each processor in a simulation would create a unique output file and output the atomic spin unit vectors in plain text format. While robust and easy to use, C++ streams have generally high processing overheads that make them unsuitable for high performance data input and output. In the VAMPIRE code the processing overheads meant that a typical data rate for output to disk was around 5kB/s on optimal hardware, for example a Solid State Drive (SSD). This poor performance was also hindered by the use of `std::endl` C++ constructs, which cause a flush of the output buffer to disk after every line of text. Atomistic simulations tend to be relatively compact in terms of atomistic level data, with computational complexity coming from the requirement for small integration time steps (0.1 fs) and long integration times (nanoseconds). This limits most typical simulations to between 10,000 and 100,000 atoms with larger simulations in the 10 million atom range, with associated configuration data sizes of between 24kB and 24MB in total for each snapshot. The MPI parallelisation of the code shows good scalability to thousands of cores (for tens of millions of atoms scale problems), but here the small amounts of data and large number of output processes made the one file per process output spectacularly inefficient, preventing the output of configuration data from the simulation.

The input data for the code consists of a small number of free format plain text files containing keywords and values for running the program. The files include information such as crystal structures, magnetic material parameters, system sizes,

runtime parameters and details of data output. In the previous implementation these input files were read by all processors to initialise the information on each process. For small core counts the overheads were trivial, but for large core counts the opening and processing of two text files per process took longer than the simulation, leading to a large bottleneck for large scale simulations on large core counts (6144 cores).

New implementation of data input

To resolve the poor scalability of data input, we developed a new method to read the plain text files on a single master process and broadcast its contents to all processes in the simulation. In C++ text data is typically handled with std::string objects, which have automatic memory management and text processing functions built in (for example conversion to uppercase, stripping out un-needed characters). However, std::string objects are not directly compatible with the MPI library and so the data had to be converted into a character array prior to an MPI_BROADCAST call. Our implementation takes advantage of the C++ standard library streams which allows a stream of text from a file (fstream) or a text string (sstream) to be treated identically.

The first part of the implementation is to replace the previous call to open the input file:

```
// file stream to load contents of input file as a stream of text
std::ifstream inputfile;

// open input file
inputfile.open( filename );
```

with an updated function call to get_string(filename) which instead returns a stringstream which can be processed in an identical way and requires no other code changes to be made:

```
// string stream to contain contents of input file as a stream of text
std::stringstream inputfile;

// fill contents of string stream with a string obtain
// from the get_string() function
inputfile.str (get_string( filename ) );
```

The implementation of the get_string() function then only reads the input file on the root process, converts the string stream to characters and then broadcasts the contents to other processors. The generality of this code can easily be used in other packages and codes (licenced under the permissive BSD licence).

```
std::string get_string(std::string const filename){

   const int root = 0; // define root process id

   int len; // number of characters in string (needed by all processes)
   std::vector<char> message; // character array (needed by all)

   // Read in file on root process
   if (vmpi::my_rank == root){

      // ifstream declaration
      std::ifstream inputfile;

      // Open file read only
      inputfile.open(filename.c_str());

      // Check for opening
      if(!inputfile.is_open()){
         err::vexit();   // exit code with an error
      }

      // load contents of file into string
      std::string contents( (std::istreambuf_iterator<char>(inputfile)),
                            std::istreambuf_iterator<char>() );

      // copy string contents to character array
      std::copy(contents.begin(), contents.end(),
                std::back_inserter(message));

      len = contents.length(); // get length of file in chars

   }

   #ifdef MPICF
      // broadcast character length to all processors
      MPI_Bcast(&len, 1, MPI_INT, root, MPI_COMM_WORLD);

      message.resize(len); // resize message on all processors

      // broadcast file contents to all processors
      MPI_Bcast(&message[0], message.size(), MPI_CHAR,
                root, MPI_COMM_WORLD);
   #endif
   std::string str(message.begin(),message.end());
   return str; // return message as string

}
```

The new input file implementation now scales to thousands of processors and since other parts of the initialisation code are fully parallelised the system initialisation now only takes a few seconds for typical-sized problems.

New implementation of data output

Unlike the new implementation for data input, the data output routines required a complete rewrite from scratch, including both atomic coordinate data (outputted once at the start of the simulation) and atomic spin data (a unit vector for each atom outputted in a series of snapshots during the simulation). Prior to performing the output modifications the data output code was reorganised into the new modular structure for the code which has directly contributed to the sustainability and future maintainability of the code. This enabled a more straightforward implementation of the new configuration output module in the VAMPIRE code.

The original proposal was to implement a new output method using MPI-IO routines found in the MPI library. However, in light of more recent data from the ARCHER team showing that file-per-process can have optimal performance for certain file sizes[3], we opted for a more flexible approach and have implemented three different output parallel output methods within the VAMPIRE code: *file-per-process*; *single-shared-file* with MPI-IO; and *file-per-node*. The advantage of this flexible approach is the ability to tune performance for different problem sizes and numbers of processor cores. In addition, there is no requirement for a parallel file system allowing the improvements to be used on a wide range of different hardware from desktop machines through to national scale resources. Each of the output methods is user configurable from the main program input file allowing easy selection of the output method at runtime.

File-per-process

The first method is a slight modification of the original output scheme where each process outputs its own data to disk but now in binary mode rather than as text using C++ streams. In this mode a peak output bandwidth of 7.5GB/s was achieved for output on two nodes (48 cores) for a large problem (3GB total data size). A particular limitation of this method for simulations using VAMPIRE is the generally small problem size, meaning that the data per process is only a MB or so for typical problem sizes with a reasonable number of total time steps. Therefore, the file-per-process output scheme is only practically useful for running large problems on a small number of processor cores. However, this would be regarded as an unusual simulation pattern and not typically requiring ARCHER and being better suited to regional or local resources. Nevertheless, comparing to the original output method the new binary mode is over one million times faster than the previous implementation, and so at least enables good performance for I/O on a range of different resources.

Single-shared-file using Message Passing Interface Input and Output (MPI-IO)

The second method implemented in the VAMPIRE code uses collective MPI-IO routines to implement a single shared file for each configuration. The order of the data is not important for data processing and so the implementation uses collective mpi_write_at_all() function calls with a predefined offset for each process. In order to have only a single write call the data is first copied from the main spin data arrays

to a linear buffer. In the main code the spin data is stored as three separate arrays for *x, y* and *z* spin data to optimize cache performance. In addition, the user can specify a subset (slices) of data to output during a simulation and so not all spin data for the atoms in the simulation are necessarily outputted to disk. The buffer therefore contains serialized spin data for only the atoms to be output in *xyz* order to facilitate easier data processing. Since the number of atoms to be outputted from each process are fixed for the whole simulation the MPI offsets for each process are calculated once at the beginning of the program. The performance of the single-shared-file is generally good, up to 2GB/s for the largest data set size benchmarked (3 GB in total) with of course a minimal number of files (1 per snapshot) to process for data analysis. Interestingly for a fixed data size the output data rate seems independent of the number of output processes. While performance is *good*, this unfortunately means that the output is not *scalable* for strong scaling problems, and so the output time is around 2 seconds per snapshot for any number of cores. However, as a fraction of total runtime this is still small given a typical number of 500 snapshots.

File-per-node

The final output method we implemented is a hybrid approach, manually combining data within a subset of processors before outputting to disk with standard C++ write functions. The file-per-node method has a number of advantages over the file-per-process and single-shared-file methods. The first is that the number of output nodes can be freely varied from one (as in the MPI I/O implementation) to the total number of processes, which can be used to achieve a balance between high performance and a manageable number of output files. The second advantage is the ability to select either binary or text formatted output for greater portability. The distributed nature of the files means that the file-per-node method can utilise distributed scratch storage on lower tier clusters where high performance parallel file systems are uncommon. Finally, the file-per-node method is also expected to have optimal performance on Blue Gene systems due to the provision of dedicated I/O nodes serving file requests from a large number of compute blades. Our implementation takes advantage of custom communicators in the MPI library to define groups of processors (MPI_COMM_IO) which co-operate to produce a single output file. A master process in the IO group collates the data from the other processes and is responsible for writing the file to disk. As with the single shared file implementation, the number of atoms from each process does not change during the simulation, and so after some initial book keeping during initialization the implementation simply requires a single MPI_Gatherv call to collect the buffers from the different processors in the IO communicator before outputting to disk. The file-per-node method achieved the best performance, with average bandwidth in excess of 30 GB/s for the optimal number of I/O tasks – the peak available from the hardware.

Performance tests on ARCHER

To evaluate the performance and scalability of the new IO functionality we have performed a number of tests for different problem sizes. As stated in the introduction, atomistic spin dynamics simulations are typically bounded by the large number of integration time steps required, from 10^6 – 10^8 (100 ps – 10 ns real time) being typical numbers. This requirement for large numbers of steps limits the ability to simulate large systems to obtain useful science. We have therefore focused performance tests on strong scaling of medium and large scale systems likely to be suitable for running on ARCHER. The first test uses a small number of processors on 1-2 nodes to assess the relative performance of the different output methods. The benchmarks consist of three different problem sizes: small (1M atoms, 24 MB total), medium (27M atoms, 648MB total) and large (125M atoms, 3GB total). The simulations are run for a small number of time steps and output 10 snapshots of the spin configuration, with 24 bytes per atom. The bandwidth to disk and time for IO operations are recorded for each snapshot and the mean values are calculated. Note that only the actual write operation is timed – the reported bandwidth is without overheads such as copying to the data buffer. File striping is left at the default value of 4 with a block size of 1 Mb which may not be optimal. For the MPI-IO method there is only a single file which is reported as one output process, but in reality all processes in the simulation are calling the MPI-IO routine. The file-per-node method can scale the number of I/O processes in the simulation from 1 to N_{procs}, and so for small scale simulations one can vary the number of output processes independent of the number of simulation processes. The average bandwidth achieved as a function of number of IO processes for different problem sizes is presented in Fig. 1.

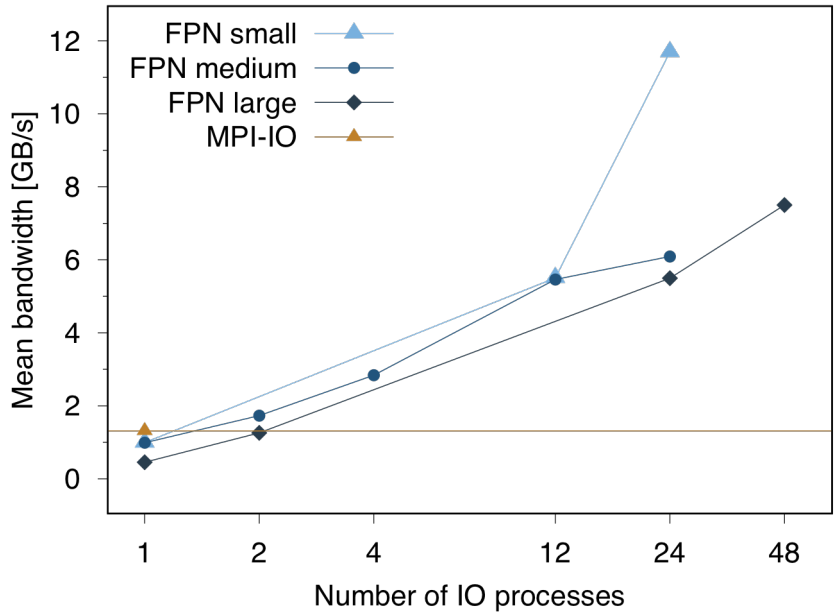

Figure 1 | Scaling of mean output file bandwidth on 1-2 nodes of ARCHER for different problem sizes.

The first thing to note about the data is the general good performance of all the IO for a range of different problem sizes and different numbers of IO processes. Considering first the large data set (3 GB total file size per snapshot), the MPI-IO method achieves around 1.3GB/s bandwidth, which is a factor 3 higher than that achieved for a single file written from a single process. However, increasing the number of output processes leads to a significant increase in performance around 7GB/s for 48 processes. Performance seems to be slightly better for the smaller problem sizes, reaching 12GB/s for a small single node simulation and 24 output processes. It is interesting that, even considering such a small amount of data there are perceptible increases in performance for more output processes. The conclusion for small simulations is therefore the more output processes the better, and fully populated *file-per-node/file per process* methods are optimal and 4-8 times faster than the MPI-IO method depending on problem size.

The second test compares the scalability of MPI-IO and *file-per-node* methods to high core counts (up to 12288 cores for 512 nodes). The advantage of the *file-per-node* over *file-per-process* method is the vast reduction in the number of files generated during the simulation, which can run into millions of files of the latter for a typical 500 snapshots and large core counts. This is particularly problematic in the strong scaling regime, where the data per process decreases linearly with core count. Therefore, we have assumed a single output process per node, but for higher core counts it may be desirable to reduce the number of output nodes to maintain a reasonable number of larger files while still maintaining high performance. Fig. 2 shows the scaling of IO bandwidth as a function of the number of nodes (I/O processes) in the simulation.

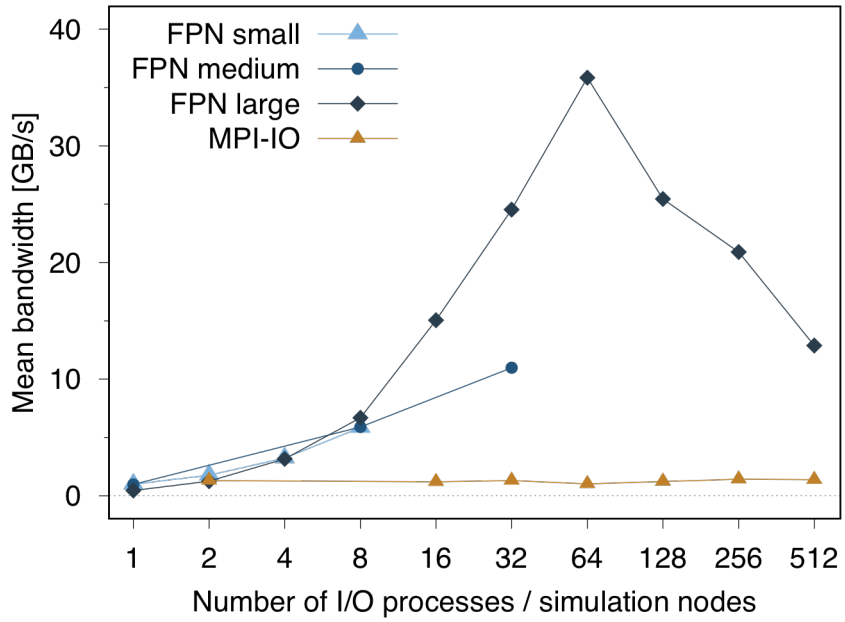

Figure 2 | Scaling of mean output file bandwidth to high core counts on ARCHER for different problem sizes.

The data show much higher performance can be achieved, approaching the peak capability of the hardware for large problem sizes. Considering the MPI-IO method for the large problem size, the output bandwidth is essentially independent of the number of processes in the simulation. The MPI-IO method is universally worse performing than the *file-per-process* method and so in general seems to be sub-optimal for the problems typically encountered with the VAMPIRE code. The reason for the much lower performance is curious, and could be because the MPI-IO library is overly conservative and accumulating too much data before outputting to disk, where one would expect that the output bandwidth would no longer scale. The *file-per process* method scales well to 64 nodes achieving a peak mean bandwidth of 35GB/s. However, for this simulation the distribution of bandwidth for individual snapshots was large, and in some cases bandwidth over 70GB/s was reported. Clearly this implies caching is being used, where the data is buffered in memory before output. However, this is still reflected as an advantage in the simulation since once the data is buffered the simulation can continue, and so the effective I/O performance is excellent. Above 64 nodes the performance decreases monotonically, indicating increasing contention for file system resources given the large number of files being opened and decreasing amount of data per process. Therefore it is likely optimal to have a maximum of 64 I/O nodes for larger core counts. In terms of time, each snapshot takes around 2s with the MPI-IO method, and around 0.1s with the *file-per-process* method with an optimal number of files. It is clear that with the new I/O methods the VAMPIRE code is no longer limited by I/O bandwidth, with 500 snapshots taking between 0.1% and 2% of a typical simulation taking 20 hours compared to over 99% of the runtime time with the previous implementation prior to this project.

Finally we tested the performance of the code for a large scale simulation with realistic inputs. The simulation models the dynamics of magnetic domains during ultrafast heat induced magnetic switching in the material GdFe. The simulation generated just over 918M atoms which was parallelised over 12,288 cores on ARCHER. During the simulation a subset of the atoms were outputted using the *file-per-node* method using 64 output processes, with around 12MB of data per output process/snapshot. A total for 421 snapshots were written to disk over the course of a 2 hour simulation with a total data size of 300GB. The output bandwidth for each snapshot as a function of snapshot number is shown in Fig. 3.

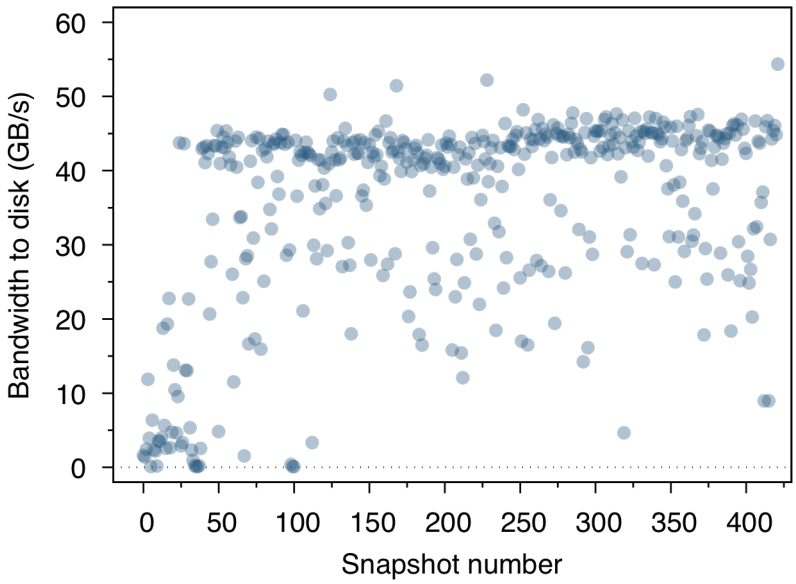

Figure 3 | Output bandwidth as a function of snapshot number for a large scale simulation of 918M atoms.

The data show excellent effective bandwidth for the majority of the simulation snapshots given the small file size of 12MB, with I/O operations completed in around 0.1s for most of the snapshots, allowing the simulation to proceed quickly. As seen in previous results, this is greater than the theoretical peak performance of the file system and so is probably due to caching the I/O operations in memory. There were a two regions of constrained operations around snapshot numbers 30 and 100 which took significantly longer than the average, with around 8s IO time for each snapshot in the worst case. This is likely due to contention in the file system but was generally a rare occurrence over the whole simulation. The realistic inputs also highlighted two segments of code which are not well-parallelised and caused a much longer initialisation time than expected, around 30 minutes. The first of these calculates the distribution of Gd and Fe in the simulation which is fast for small systems, but for large systems is slow. The second is the calculation of the halo data which is currently implemented as a loop over all atoms looping over all processors. For large systems and large core counts this is also a significant bottleneck in the program initialisation.

Having identified these problems we plan to implement code improvements to address them in the near future.

Data processing utility: VAMPIRE Data Converter (VDC)

The new output routines lead to a natural diversity of output put data in terms of the number of files, their format and structure. To make data analysis and visualisation straightforward for users we have developed a new data processing utility (Vampire Data Converter, or VDC) which can read any kind of data generated by the code and output it as flattened data in text format or as renderable files with the PoVRAY ray tracer[4]. The utility works by reading a single metadata file for each snapshot which identifies the type and format of the data. The utility then interprets the data in the appropriate way, loading the data into memory. Once loaded the data can be manipulated to select a subset of data based on geometric or based on the spin configuration for example. In future we intend to add new functionality to the conversion utility, supporting a wider range of file formats and output options.

Documentation and online tutorial

Documentation of how to compile and obtain optimal performance of vampire on ARCHER has been included into the upcoming version 5 of the code, due for release later in 2017. In addition, during the project we merged the LaTeX source code for the VAMPIRE manual into the main GitHub repository so that documentation can be written alongside new features and is always up to date with the features of the code. This will significantly improve the sustainability of the VAMPIRE code going forward. Details of the new output methods in the code (file-per-node, file-per-process and mpi-io) have also been written into the software manual, with descriptions of how to use the new methods and guidance on how to maximise performance on different hardware. For novice users we have also drafted a tutorial for the next iteration of the project website (to be published alongside version 5 of the code), walking through a simple simulation and submission script on ARCHER using the new output methods.

Conclusion

In conclusion, we have implemented major improvements to the input and output routines in the VAMPIRE code, showing over 1,000,000 times improvement over the previous implementation. Supported by new user documentation and training, this has enabled unprecedented large scale simulations of magnetic materials with atomic scale resolution and will allow

Acknowledgement

This work was funded under the embedded CSE programme or the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] R. F. L. Evans et al, *J. Phys.: Condens. Matter* **26**, 103202 (2014)
[2] http://vampire.york.ac.uk
[3] http://www.archer.ac.uk/documentation/best-practice-guide/io.php
[4] http://www.povray.org