Full parallelisation and modernisation of the BGS global magnetic field model inversion code

BGS MEME eCSE technical report

Abstract

The British Geological Survey's global geomagnetic model, Model of the Earth's Magnetic Environment (MEME), is an important tool for calculating the earth's magnetic field, which is continually in flux. Whilst the ability to collect data from ground based observation sites and satellites has grown rapidly, the memory bound nature of the code has proved a significant limitation in modelling problem sizes required by modern science. In this paper we describe work done replacing the bespoke, sequential, eigen-solver with that of the SLEPc package for solving the system of normal equations. This work had a dual purpose, to break through the memory limit of the code, and thus support the modelling of much larger systems, by supporting execution on distributed machines, and to improve performance. But when adopting SLEPc it was not just the solving of the normal equations, but also fundamentally how we build and distribute the data structures. We describe an approach for building symmetric matrices in a way that provides good load balance and avoids the need for close co-ordination between the processes or replication of work.

Performance and scaling characteristics are explored on ARCHER, a Cray XC30, where we achieved a speed up for the solver of 294 times by replacing the model's bespoke approach with SLEPc. This work also provided the ability to model much larger system sizes, up to 100,000 model coefficients, which is also demonstrated. Some of the challenges of modelling systems of this large scale are explored, and mitigations including hybrid MPI+OpenMP along with the use of iterative solvers are also considered. The result of this work is a modern MEME model that is not only capable of simulating problem sizes demanded by state of the art geomagnetism but also acts as further evidence to the utility of the SLEPc libary.

Introduction

The BGS global geomagnetic model inversion code is used to produce various models of the Earth’s magnetic field. It is essentially a mathematical model of the Earth's magnetic field in its average non-disturbed state. The input consists of millions of data points collected from satellite and ground observatories on or above the surface of the Earth which are used to identify the major sources of the magnetic field – the core, crust, ionosphere and magnetosphere. The magnetic field is then solved for the so-called Gauss coefficients, which describe the magnetic field as weighting factors for spherical harmonic functions of a certain degree and order (i.e. spatial wavelength). In addition, the Gauss coefficients have a temporal dependence requiring the solution of weights for a sixth-order B-spline function. The output is thus a set of around 10,000 coefficients describing the spatial and temporal variation of the magnetic field from the core to near-Earth orbit over a period of around 15 years. This allows a compact representation of the magnetic field.

The inversion code, although partially parallelised, has critical parts which currently run in serial fashion, in particular the solution of the normal matrix. This serial portion places significant limitations on scaling up the parallel performance of the problem. It has been determined that a fully parallel algorithm is needed, which avoids the various partial-step solutions which are currently used during the computational work. A fully parallel problem would allow the spatial and temporal resolution of the model field to be increased significantly. This should in turn give reduced uncertainties and an increase in the predicted confidence levels in the modelled field. The models are widely used in the scientific community e.g. for studying the core field or the effect of space weather and non-scientific communities, for example in navigation (civilian and military), or for commercial activities in mineral or hydrocarbon exploration (c.f. Figure 1).

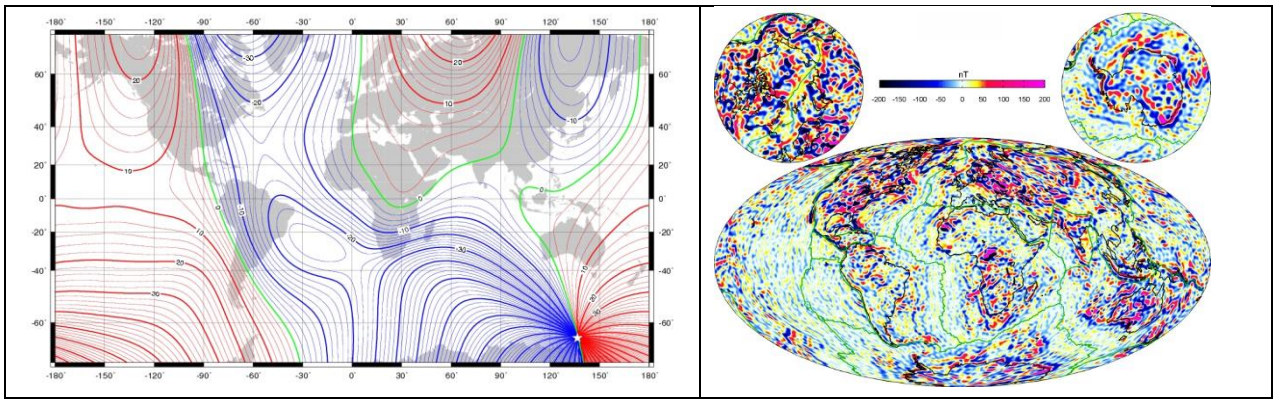

Figure 1 - (Left) Magnetic Declination angle for 2015.0 for the Earth's main field to degree and order 13. (Right) Lithospheric magnetic field strength for degree and order 16 to 133 (in nanoTesla).

From the BGS code, the community produces a number of models each year both for research and for non-scientific users. One such model is the International Geomagnetic Reference Field (IGRF). This model is widely accepted as a standard low spatial resolution model of the Earth's magnetic field. One user community is solar-terrestrial physicists who use it for their magnetic coordinate systems. Another community studies the history of the Earth's magnetic field over thousands of years in order to understand the underlying physics generating the field. The IGRF is updated generally only every 5 years and BGS has contributed since its inception in the late 1960s. Another widely-used model is the World Magnetic Model (WMM). This model is used in civilian and military navigation and positioning systems, including mobile phones, and is updated every 5 years by BGS with their counterparts in the USA. Other models are produced annually by BGS for detailed navigation where very accurate values of declination are required, or to look at rapid time variations of the magnetic field for scientific study.

Performance and scaling characteristics of existing model

Previously a partial parallelisation of this code was undertaken with MPI which concentrated on the building of the matrix and RHS vector, all other aspects (such as the Eigen solve and IO) remained serial. This parallelisation worked by allocating the entire data structures on every process (e.g. the entire matrix and RHS) and decomposing on the input data. Contributions from each input data are additive, and as such each process would calculate the contributions for its subset of data across the entire matrix and RHS, before all individual processes' data was collected at rank 0 and added together.

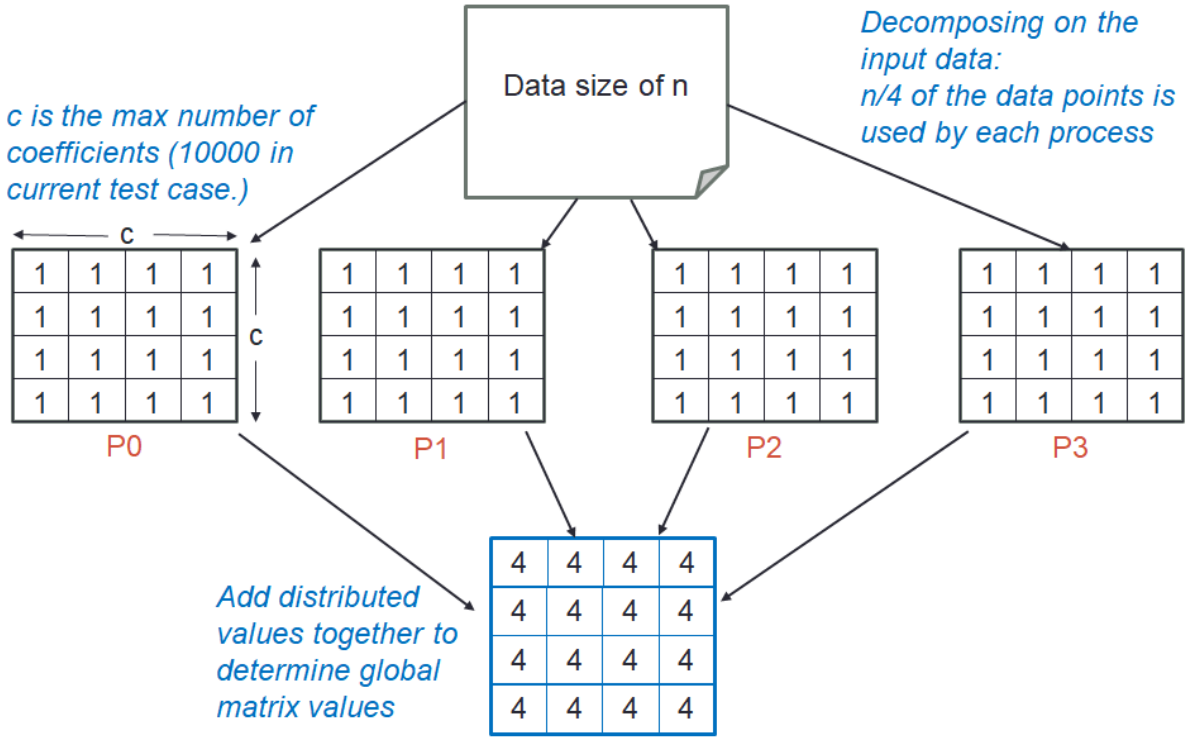

Figure 2 - Illustration of existing parallelisation of matrix and RHS building by decomposing on the input data

Figure 2 illustrates the existing parallelisation in more detail, where input data size of *n* is decomposed across the processes and then the entirety of the data structures (in figure 2 only the matrix is illustrated) is built locally for that subset of data. Once local building has been completed these values are collected at a root process (rank 0) and added together to form the final matrix values.

There are a number of limitations with this approach and implementation:

1. Because the entire global matrix has to be held on each process, the memory limit of the machine is quickly reached. In figure 2 this is illustrated by c, which is the number of model coefficients (or parameters.) Common parameter sizes are 10000 which results in a matrix of 10000 by 10000 which, bearing in mind this is double precision, is 800MB of data per process. The community would like to extend this to much larger system sizes, but the amount of memory required increases exponentially as the number of model parameters increases. For instance moving to 20,000 parameters would require 32GB memory per process. This is important because model users are currently forced to throw away significant amounts of data (for instance they can only use 1 in 20 data points collected by EASA SWARM satellites due to the memory limits of the code.)
2. This parallel implementation of this assembly uses point to point MPI communications (blocking synchronous sends and blocking receives) to gather up data from each process for the addition on the root rank. This would be far better as a reduction.
3. There is a work imbalance between the processes in calculating the normal equations and RHS where there was a 37% difference in the runtime between the slowest and fastest process in our test run of the matrix and RHS building. Due to the implementation of the MPI communication, this results in excessive stall time for process, where the slowest rank dictates the overall run time.

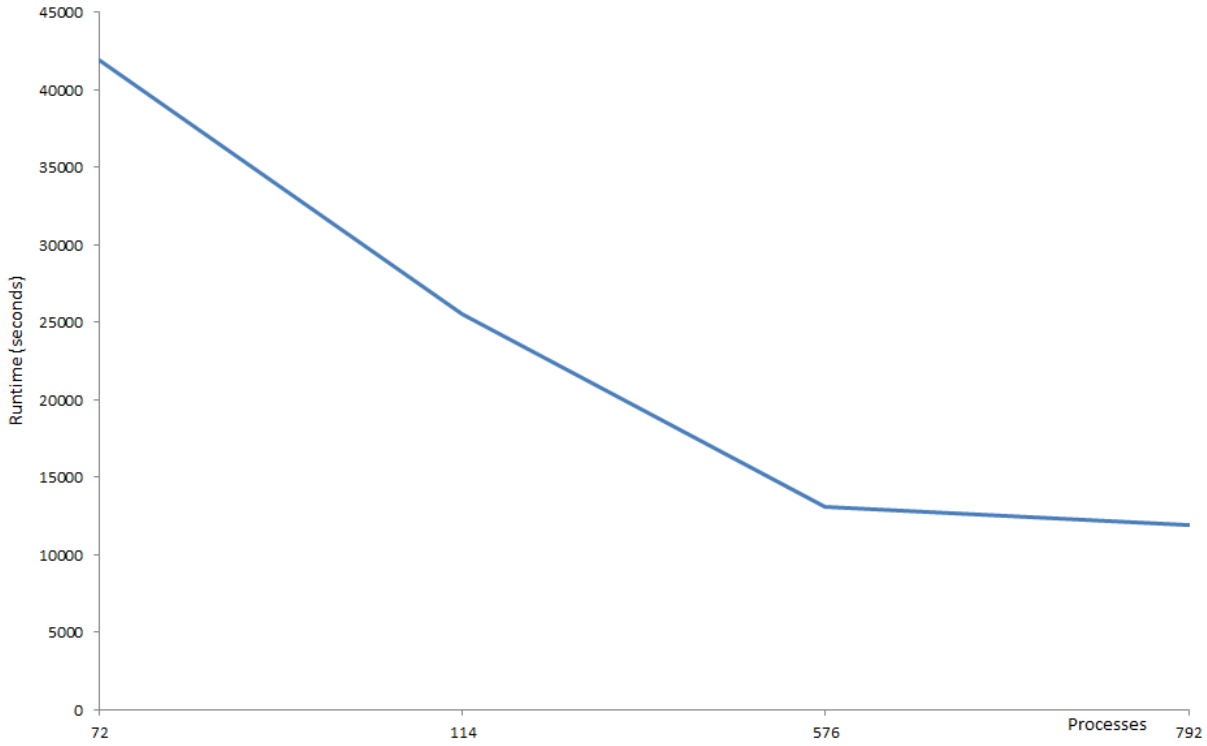

Figure 3 - Strong scaling experiment of previous code, 10000 model parameters, with 4.6 million data points

Figure 3 illustrates strong scaling of the previous code on ARCHER with 4.6 million data points and 10000 model parameters. It can be seen that, up until 576 processes (24 nodes), a modest speed up is achieved (3 times compared to 72 processes whereas a speed up of 8 times is theoretically possible.) At 576 processes 28% of the overall runtime represents execution running in parallel and the remainder is running in serial. This very significant serial fraction, combined with the naive use of MPI communication calls, therefore limits any further speed up and it can be seen that there is little improvement at 792 processes (33 nodes.)

Description	**Percentage of runtime**	**Actual runtime (seconds)**
IO (reading and writing)	0.09%	126
Determine distribution	1.5%	207
Build matrix and RHS (in parallel)	28%	3777
Calculate Eigen values and vectors	62%	8188
Other Eigen solver activities	4%	583
Calculate deviation of solution	1.6%	215
Misc	2.31%	302

Table 1 - Profiling of runtime over 576 processes

Error! Reference source not found. illustrates profiling of the runtime over 576 processes, it can be seen that by far the most significant amount of time is spent (in the serial part of the code) calculating the eigenvalues and vectors as part of the Eigen solver. Even though building the matrix and RHS is still parallelised there is still a significant amount of time spent and determining the distribution of particles across processes also consumes a considerable amount of runtime.

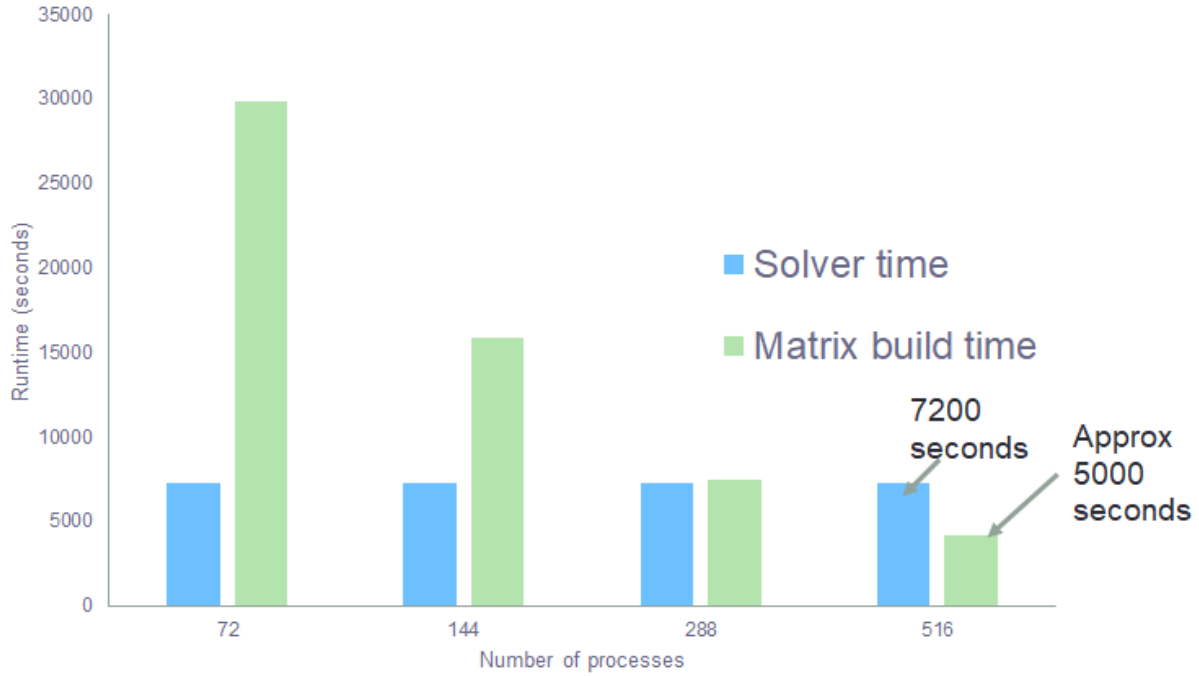

Figure 4 - Solver and matrix building time for 8,000 parameter problem with 4.6 million data points

Figure 4 illustrates this same thing in a different manner for a different experiment of 8,000 parameters and 4.6 million data points. It can be seen here that the solver time is constant, 7200 seconds, irrespective of parallelism due to its sequential nature. The matrix and RHS building time does scale as parallelism increases, but at 516 cores still takes a substantial amount of time.

The solver itself is based around a direct solve where an eigensolve is performed to find the eigen values and vectors of the matrix and then apply these to the matrix and RHS. In the previous code this was using a serial Givens reduction to solve the eigenvalue problem.

Improvement of the model – the proposed plan

The major aspect for the model community was to be able to run larger systems and hence the memory limitation of the current approach was their most critical limitation of the current model. However the runtime is still very important, because these larger systems must run and as such optimisations to the performance hotspots of the model were also critically important. Actually it was our view that these things went hand in hand, work to improve the parallelisation of the model addressing the memory limits should also have a positive impact on the overall model performance.

The specific objectives for this project were:

- The ability for the code to run on at least 1,000 cores with reasonable scaling/parallel efficiency. The runtime of the code on 112 cores of the BGS machine (for a specified data set/test-case) is around 11 hours. This contains a significant serial fraction and if the code was completely parallelized the runtime should be significantly reduced. Strong scaling wise, we are aiming for a baseline of max 8.5 hours runtime.
- Provide the ability to solve over 100,000 parameters which is the number needed to describe in time and space the various geophysical systems we can model with the magnetic field (core, lithosphere, ionosphere and magnetosphere).
- Solve the 100,000 parameter task in 240 hours or less.

- Along with previous performance metrics, results should be in agreement (e.g. within a relative 0.1%) with the results produced by the current code on Archer and BGS facilities.

For reasons of reproducibility with the previous model it was decided to stick with using an eigensolve and to use the PETSc library, where we will take advantage of the distributed memory functionality to build up distributed MPI based matrices and vectors. The Scalable Library for Eigenvalue Problem Computations (SLEPc), which is a library that sits on top of PETSc, provides Eigen solving capability and the solver directly called from the code. The benefit of using PETSc and SLEPc is that we gain the benefits of modern solvers and parallelism, whilst keeping with using an Eigen solver means other parts of the code (such as calculating the formal deviation at the end) which rely on the resulting eigenvalues remains unaffected. After this is working and validated, it will then be possible for the BGS and general community to test iterative solvers that ship with PETSc, such as GMRES, for any likely performance/scalability improvement that an iterative method might produce and at what stability cost in the future.

Driven by how PETSc works, instead of decomposing on the input data points, instead the matrix (and RHS) is decomposed amongst the processes. The idea is that this will still be built in a distributed fashion with each data point contributing to each location in the matrix and RHS and every input data point will exist on every process for this step. Not only does this break the memory limit, as the matrix is now distributed across the processes, but it also allows for the Eigen solve (including the calculation of eigenvalues & vectors) to be performed in parallel.

In conjunction with this a plan to generally modernise the model from a code perspective was also proposed. The previous model was fixed form FORTRAN 77 and fairly messy from a code perspective as numerous people had worked on it over the years. It was decided that, refactoring this to free form Fortran 90 and applying code standards throughout would make the model more maintainable by the community and might even help improve single node performance.

Our new model: Distributed building of the matrix and RHS

Instead of the entirety of the matrix and RHS being resident on each process and the processes then decomposing on the input data, the decomposition is done on the matrix and RHS, with the entirety of the data processed for this subset of the data structure. This is illustrated in figure 5, where both the matrix and RHS structure (which is a 1D array) is decomposed across the processes for *n* model parameters. The challenge with this approach is that the matrix is in-fact symmetrical, so only the diagonal and one half of the matrix actually needs to be computed. It is fairly easy for the previous model to take advantage of this fact, as each local process will only compute the values for the diagonal and upper half of the matrix before communicating these to the root rank (process 0.) This root then sums up the values at each cell of the diagonal and upper half of the matrix from each process, before copying the upper values to their corresponding lower value location in the matrix. Because the building of the matrix and RHS is so expensive (e.g. over 30000 seconds on 72 processes), being able to limit the amount of data explicitly computed is an important saving. This symmetry is illustrated in figure 6.

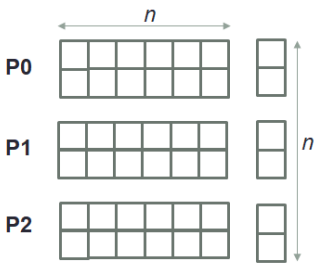

Figure 5 - Decomposition of matrix and RHS

However there is a problem with the decomposition of figure 5 – namely that exploiting the symmetry of the matrix becomes far more difficult because the corresponding cells in the upper and lower parts of the matrix are likely on different processes. There are two naive options, firstly one could ignore the symmetry and instead all processes calculate the points assigned to them or secondly each process holding a diagonal and upper cell calculates this and then MPI communication is used to send the upper value to the process holding the corresponding lower cell.

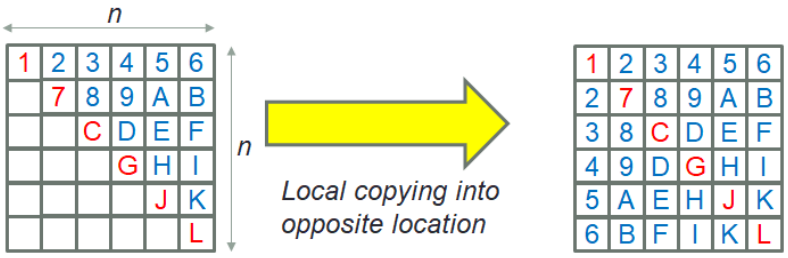

Figure 6 - The previous model only explicitly computes the diagonal and upper half of the matrix, then copying the upper values to their corresponding lower cell

Neither of these options are ideal, with the first option redundant global computation is being performed as explicit calculation of each of the lower cells would be a duplicate of work done elsewhere. The major limitation of the second approach is illustrated in figure 7, which results in a significant load imbalance. For instance on the example of figure 7 process 0 must calculate 13 points whereas process 2 only 3 points. Because of the intensive nature of building the matrix and RHS this would result in process 2 spending much of its time idle.

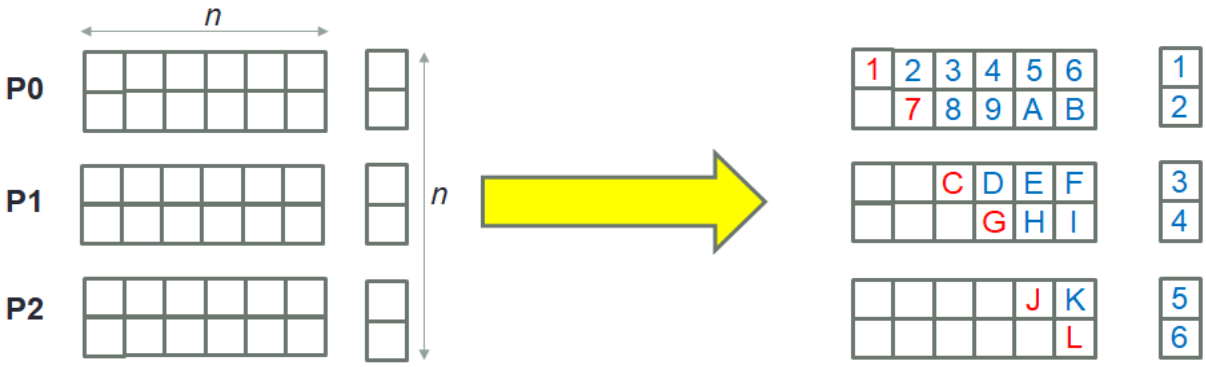

Figure 7 - Naive approach of only calculating diagonal and upper matrix points held locally

It was therefore realised that a different approach was required which would natural balance the load between processes when building the matrix. Therefore, instead of simply limiting the calculation to the diagonal and upper part of the matrix to avoid redundant calculations, we developed an approach where processes would build specific upper and lower parts of their matrix, but crucially without the

global duplication of computation (e.g. if one process is building the values for a lower cell then there is the guarantee that the corresponding upper cell value will **not** be built by the holding process) and minimal global coordination.

$$f = \frac{n^2 - n}{2} + n$$

Equation 1 - The global number of cells that been to be explicitly calculated, where n is the matrix size in one dimension

Equation 1 illustrates the calcul of cells that must be explicitly calculated based on a matrix of neters.) For instance using the example of figure 7, the matrix size is 36 but *f* is 21, so only 21 cell values must be explicitly calculated. For each row we calculate *r=f/n*, which is the base points per row that need to be calculated. In the example of figure 7 this is 3.5 . Then, for each row held locally the rank starts at the diagonal and calculates *r* local points. This is illustrated in figure 8, if *r* is a whole number (*n* is an odd number in this case) then that is all that's required, however if *r* is a fraction (*n* is an even number as in the example here) then slightly more is required. In the case of *r* being a fraction then starting from the first row, for each row the rank processing that row alternates between *r=ceil(r)* and *r=floor(r)*. If the number of rows divided by 2 (*n/2)* is even, then this ordering of ceiling and floor needs to be swapped in the second half of the matrix. This pattern can be clearly seen in figure 8, where the elements per row alternates between ceiling and floor and as 6/2=3 is not even, then no swapping of these should be done.

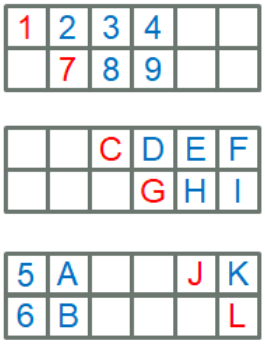

Figure 8 - Balanced building of the matrix

The distributed algorithm building so far provides the ability to build the matrix in a way that requires minimal coordination between the processes to determine which cells to explicitly calculate, avoidance of replicating work and reasonable load balance. The next step is to then send calculated data values in the upper or lower parts of the matrix to the corresponding process which will need to place that value in its lower or upper part of the matrix. Each process calculates the number of values it needs to send to every other process and the number of values to receive from every other process. Based on this send and receive buffers are allocated, cell data is packed into the appropriate send buffer, and non-blocking MPI sends and non-blocking MPI receives are issued.

This is illustrated in figure 9, the data per cell is not just the explicit value, but additionally the zero indexed global row and column of that piece of data. Therefore each cell requires 16 bytes (8 bytes for the double precision value and two 4 byte integers.) This additional global coordinate is associated with each data value to both ensure the correct mapping of values to cells on the receiving process, and also it means that determining the placement of this value locally on the receiver is trivial (just

swap the row and column around and translate the global row to a local row based on the local starting row position.)

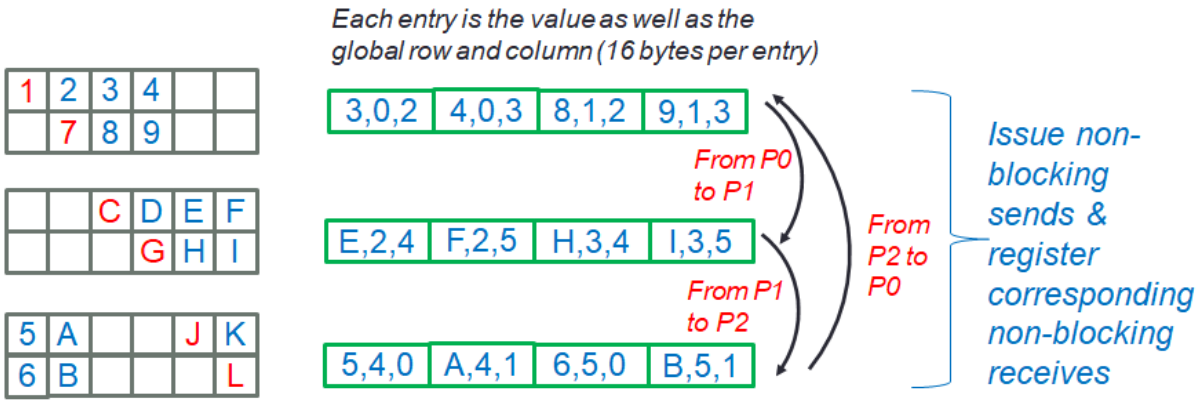

Figure 9 - Sending cell data to corresponding process

At this point copying of values between the upper and lower parts of the matrix held locally occurs (for instance in this example the value *2* will be copied on process 0 from the 1st column of row 0 to the 0th column of row 1.) Next MPI waits are issued for the non-blocking communications and when these have completed the received values are copied from the receive buffers into the appropriate local matrix cell by swapping the row and column and adjusting the row for the local starting point. This is illustrated in figure 10 and represents the last step of the global building of the matrix (and RHS.)

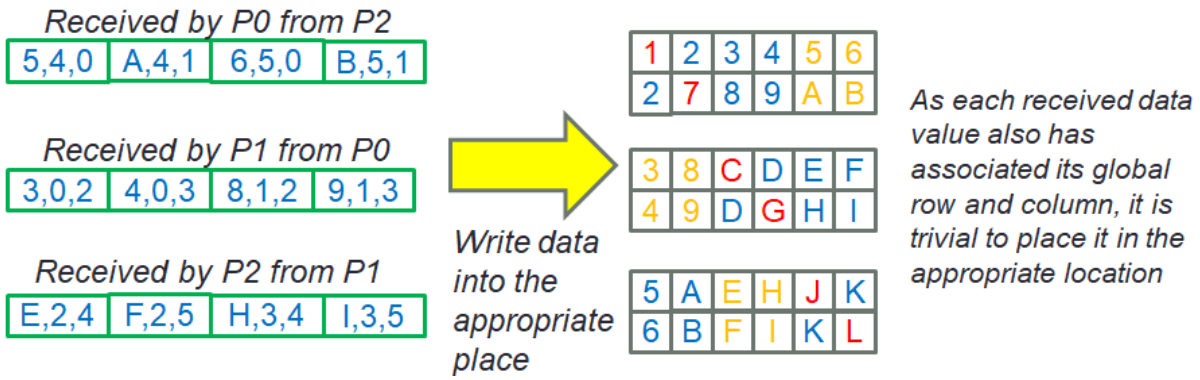

Figure 10 - Receiving values and copying them from the receive buffer into the appropriate matrix cell location

In addition to the modified parallelism described in this section we also reordered and optimised a number of loops in order to enable them to make better use of the cache. There were also a number of loops where we were able to re-order the loop and avoid redundant internal calculations. One example of this was a loop that stepped through the data and for each iteration required calculation of a specific value to apply to the data. By reordering the loop here we were able to calculate this value only once and then loop through all the data elements and apply this value before moving onto the next one. Whilst these sorts of serial code optimisations were not the main focus of this project, we felt that simple modifications that improved performance were worth doing as part of the general code refactoring.

PETSc and SLEPc Eigen solving

Instead of parallelising the existing Givens reduction, we decided to replace this with an approach that utilises the PETSc and SLEPc libraries for the Eigen solve. The Portable Extensible Toolkit for Scientific

Computation (PETSc) is a is a suite of data structures and routines developed for the parallel solution of scientific applications modelled by partial differential equations. Not only does PETSc ship with a variety of highly optimised pre-conditioners and solvers, there is also extensive support for flexibly selecting different modes of execution that include running serially, running over distributed memory machines (using MPI) and even GPUs. Whilst PETSc does not come with any Eigen solvers, the library itself has been designed so that aspects can be used as building blocks by other libraries. As such the Scalable Library for Eigenvalue Problem Computations (SLEPc) sits on-top of PETSc, providing the eigen solver capability but relying on the PETSc eco-system for parallelisation, utility functionality and general program flow. The beauty of using SLEPc is that the code is still written in *the PETSc style*, just the specific solver created is an Eigen solver in the SLEPc library. This is important because not only does it mean that those familiar with the common PETSc library will understand the code, but also it should be fairly trivial (from a code perspective) to swap out the Eigen solver and replace it with a PETSc iterative solver.

The SLEPc libraries comes with a number of Eigen solvers including Krylov-Schur, Arnoldi, Lanczos, GD and Lapack. There are also a significant number of options that can be tuned, either directly in code or via command line arguments to the executable to set specific configurations. After extensive testing we found that for this problem the default, Krylov-Schur, solver is by far the best (most accurate with respect to the previous code and fastest). We spent significant time investigating different configurations, specially the maximum projected dimension which trades off memory usage for redundant computation when running in parallel. However, none of these made a significant difference to performance or memory usage and as-such we concluded that the default settings were optimal for the model as it currently stands.

This raises an important point however, as the MEME model itself is fairly generic with the specifics of the problem, effectively the equations that determine the values held in the matrix and RHS, provided by a driver code which bootstraps the model. As such it might be that other future users of the code, with their specific models dictated by their own driver code might benefit from other SLEPc solvers or options. In the first instance these individuals can experiment with different PETSc and/or SLEPc options via command line arguments to their MEME model executable. To make these more permanent, on a driver by driver basis without polluting the MEME generic library (as it might be used by multiple drivers) we extended the driver code to provide an explicit opportunity for setting driver specific configuration options. One such option in the current model is to tell SLEPc that the problem is generalized Hermitian, this is important because not only does it significantly improve performance in comparison with the assumed non-Hermitian approach but also the previous model makes this assumption and-so this mode of operation most closely matches the results between the new and previous model.

Code refactoring

As part of this work we have also undertaken large scale code refactoring, upgrading the code from fixed form FORTRAN 77 to free form Fortran 90. In addition to this we took the opportunity to apply coding standards and restructure the library layout. In this section we provide an overview of the steps taken.

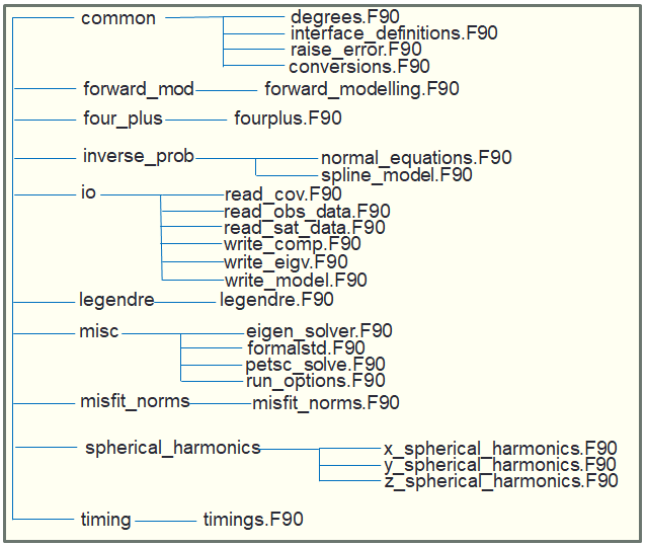

Figure 11 - Revised MEME library structure

Figure 11 illustrates the revised model library structure, where numerous F77 files were concatenated into free form F90 modules. In addition to the code itself the makefiles were also overhauled, with a *Makefile.base* provided and then *makefile_optimised* and *makefile_debug* which call into the base makefile but build the model for debugging or with full optimisation enabled. The Cray, Intel, GNU and Solaris compilers are supported as targets in the makefile. For instance, to build an optimised version of the code on ARCHER with the Cray compiler one would issue *make -f makefile_optimised Cray* . The debug makefile inserts symbols into the executable (*-g)* and also enables error trapping such as indexes going out of bounds and FP exceptions. We don't suggest using the debugging version in production, but was useful during the development work here and also will support user developing their own driver code to ensure that this is correct before the large scale production runs.

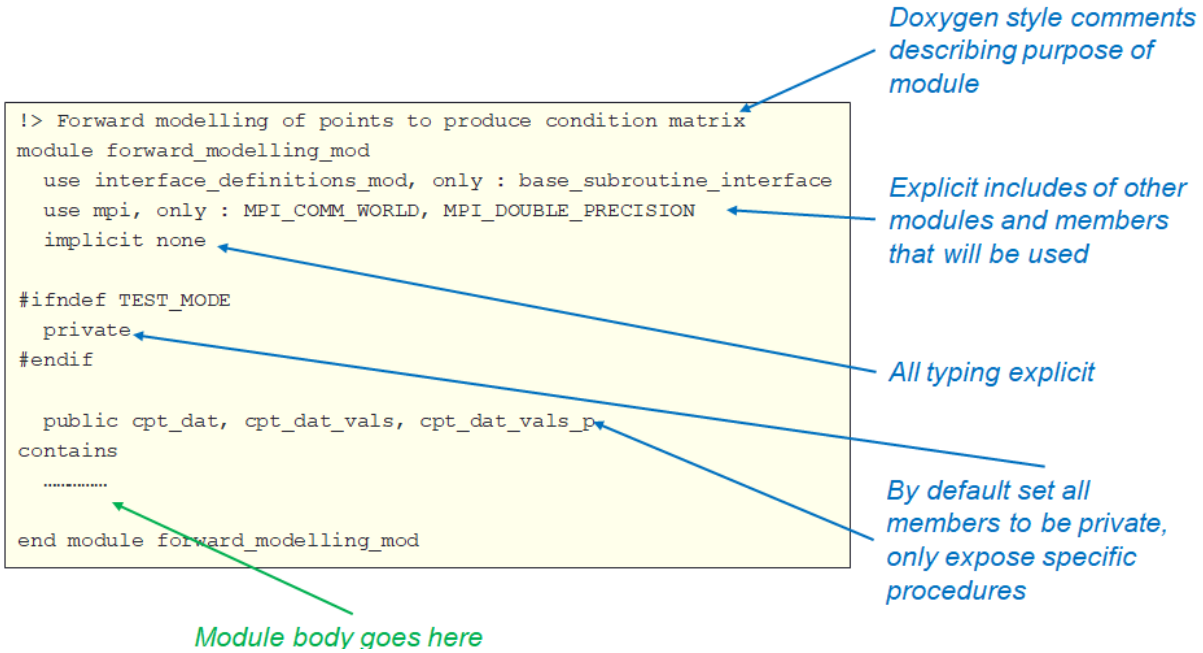

Figure 12 - Illustration of standard module structure used throughout the code

Figure 12 illustrates the standard module structure we have adopted for all the free form F90 modules of figure 11. All modules are named with the *_mod* postfix to make their module status explicit and

Doxygen has been used for document generation and as such comments throughout the model follow Doxygen style mark-up. Modules are by default private and explicitly expose specific public procedures, likewise when using modules we only access the specific procedures that the module requires (e.g. only using *base_subroutine_interface* of the *interface_definitions_mod* module rather than accessing all members.)

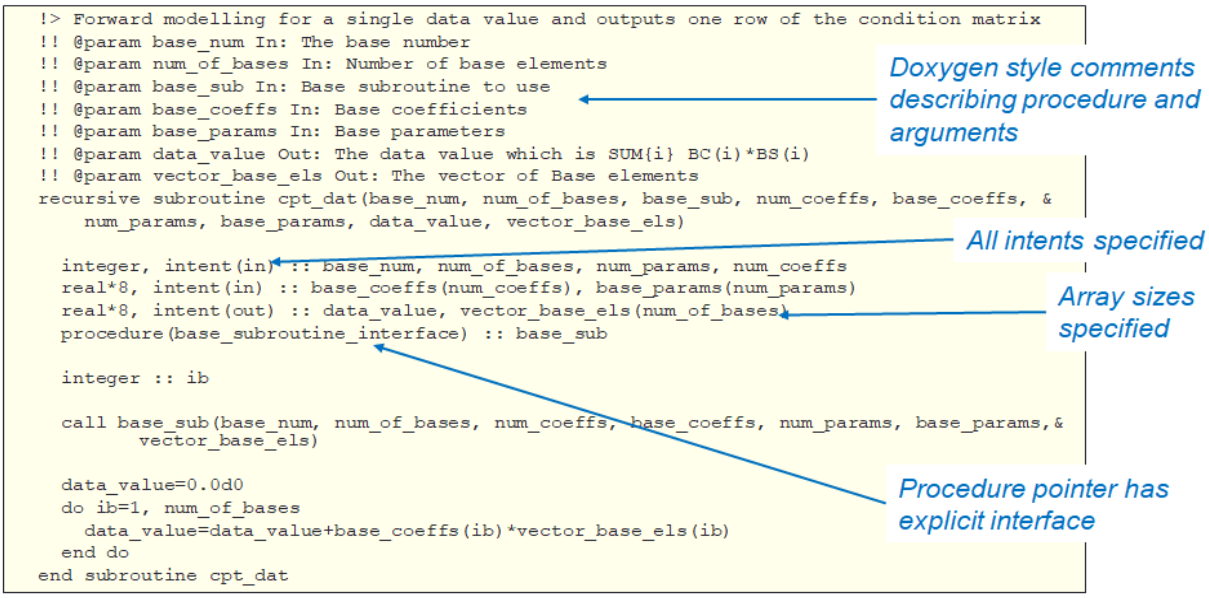

Figure 13 - Illustration of standard procedure

Figure 13 illustrates the standards adopted for each procedure, where Doxygen mark-up is used to provide a high level description of the procedure and then each argument is explained (not all arguments are documented in figure 13 for brevity.) This is a significant step up from the previous model, where the limited code commenting was in French! In the previous model argument names were fairly meaningless and varied from procedure to procedure. Hence in the new model we renamed all arguments to be much more self describing (e.g. *num_of_bases* is the number of bases, *num_coeffs* is the number of coefficients) and these are used as standard throughout the code. Intents are specified for all procedure arguments and the size of all arrays is also provided explicitly rather than inferred which is how the previous model was written. This last point is important, because by making the size of arrays explicit it is then possible to do bounds checking at debug mode, which isn't the case with implicit sizing.

In their driver code the user writes a number of procedures and then provides pointers to these as arguments to the main MEME model library code. The majority of the execution time resides in the MEME library itself, calling out to these user defined procedures via the procedure pointers for problem specific functionality such as the equations which govern the values held in the matrix and RHS. In the previous version of the code these were just passed without any definition and as such no error checking was possible on the arguments. We have explicitly defined interfaces for all of these procedures in the *interface_definitions_mod* module which governs their signature and tightens this aspect up. Hence if the user writes their own driver code and their procedures do not match the signature expected by the model then a compile error will be generated straight away.

From these standards, and especially the Doxygen mark-up style comments applied to all procedures and modules, documentation is auto-generatable and scripts have been included in the repository for regenerating the documentation in the future as the community continues to further develop the model.

Performance, scaling, accuracy and memory usage

Performance and scaling comparisons have been done on ARCHER. All experiments have been compiled with the Cray compiler version 8.6.5, PETSc version 3.8.4 and SLEPc version 3.8.3. In this section we concentrate on a system size of 10, 339 coefficients and 4.3 million input data items, contrasting the performance of our new code using PETSc and SLEPc for solving the normal equations, against the previous model. The chosen system size represents the limit of the capability of the previous model. The results of this experiment are illustrated in Figure 14 (runtime, on the vertical, is log scale), where it can be seen that the performance of the newly developed model as part of this eCSE project is significantly better than that of the previous code (vertical runtime is a log scale in the graph.)

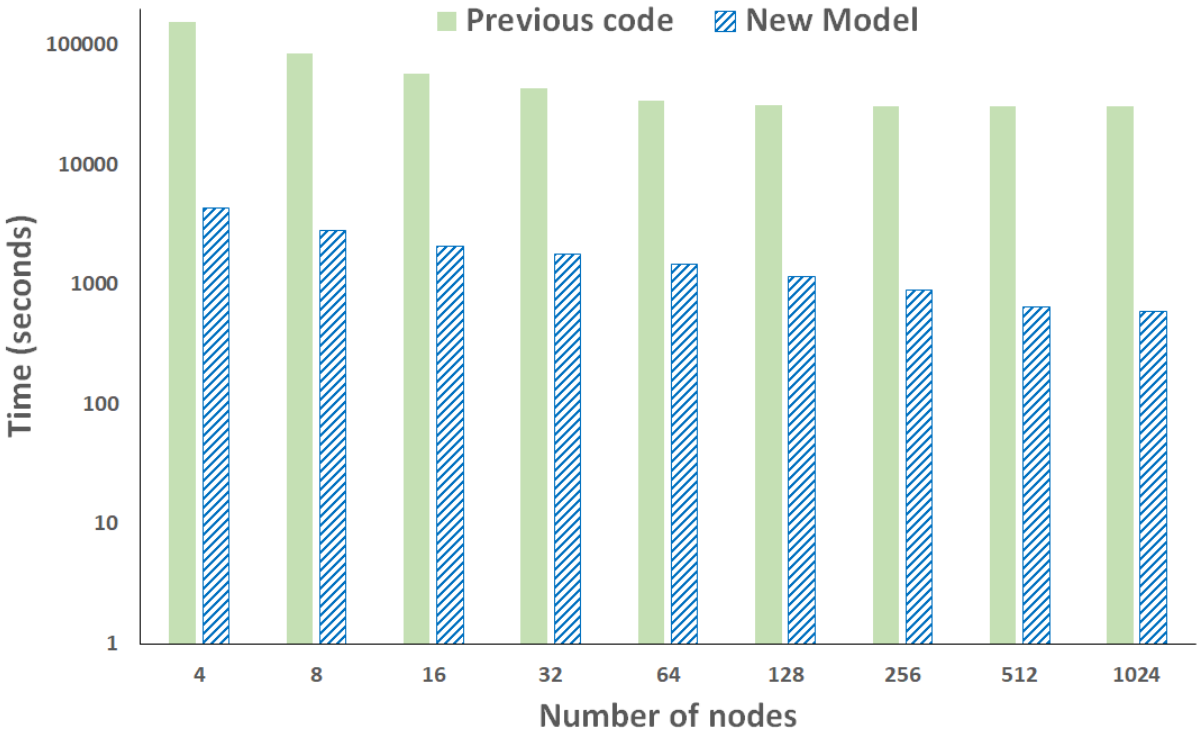

Figure 14 - Performance comparison of previous and new model with system size of 10,339 parameters and 4.3 million data points. The new model is 51 times faster at 1024 nodes than the previous model.

Figure 15 illustrates the breakdown of timings for the results in Figure 14 between the normal equation build time and the solver time (again, vertical axis run time is log scale). It can be seen that, as expected, the solver time of the previous code is constant at 28,500 seconds irrespective of the parallelism due to its sequential nature. The run times for the normal equation building in the new code, along with the solve are significantly smaller than that of the previous code. Figure 15 it can be seen that the most significant difference in terms of performance between the two models is in the solver time, where our new code takes 194 seconds over 4 nodes and 97 seconds over 64 nodes in contrast to 28,500 seconds for the previous model regardless of parallelism. This represents a speed up for 294 times and illustrates one of the major benefits to using SLEPc and PETSc over the bespoke serial solver in the previous code, both in terms of raw computational performance and also the ability to leverage parallelism. In terms of solver performance, 97 seconds over 64 nodes was the fastest that it ran, with the solver run time slightly increasing beyond this point, e.g. to 130 seconds over 1024 nodes. This is because, with 10,000 model coefficients, we are hitting the limits of strong scaling and the problem size was not big enough to take advantage of the increased parallelism.

This really illustrates the benefit of using SLEPc and PETSc over the bespoke serial solver of the previous code, both in terms of raw computational performance and also the ability to leverage parallelism. Having said that, the building of the matrix and RHS is also substantially faster in the new code compared to the previous code. This is a combination of serial and parallel optimisations as described previously.

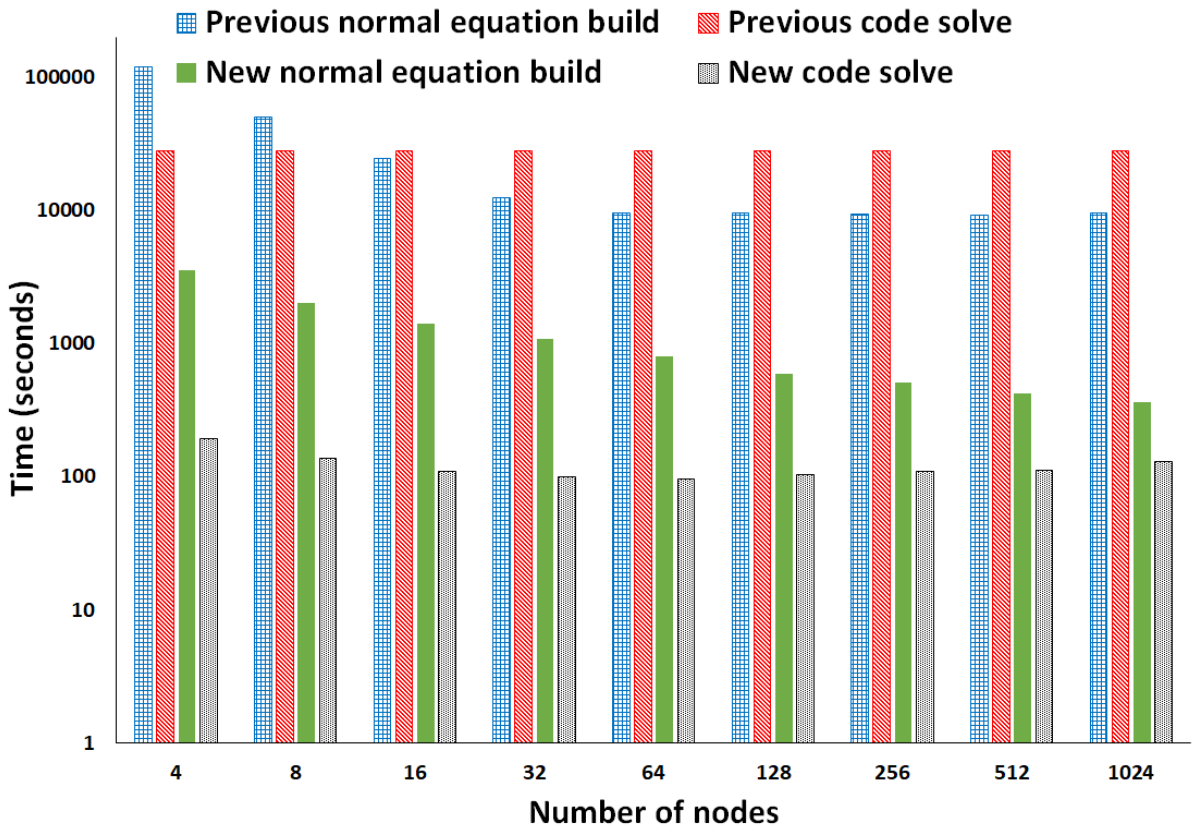

Figure 15 - Amount of run time for solver and building of normal equations for previous and new code as the number of nodes is scaled. Strong scaling effects also play a part here, but at the optimal number of nodes (64) the SLEPc approach is 294 times faster than the previous model's bespoke solver and at 1024 nodes our new normal equation building is 26 times faster than the previous model's approach.

Table 2 illustrates the percentage difference between results generated by the previous code and the new model in the experiment of this section over 32 nodes. To do this comparison we wrote a single Python script, which is in the repository, and this compares the different types of textual results files for the two model runs. In this work we have not modified the format of the output files, as the community has other tools downstream that processes these files. However because we have changed so significantly how the matrix and RHS is built, along with a completely different approach to solving the Eiven value problem, there was a significant question around how the models would compare and we had a target of 0.1% mean difference. Table 2 contains the minimum, maximum and mean difference for each result file (Ffit to XYZfit_c(3)) and it can be seen that the mean difference is well below this 0.1% difference target. The *Ffit* and *Fobsfit* results contain one result for each element, whereas for the *XYZfit*, *XYZobsfit* and *XYZfit_c* results there are three separate result elements. In the later case we only report elements 1 and 3 in table 2 because element 2 is exactly equal, a zero percent difference, throughout all the results between the two models.

Metric	Minimum % difference	Max % difference	Mean % difference
Ffit	0.000019	0.000053	0.000026
Fobsfit	0.000014	0.000029	0.000019

XYZfit (1)	0.000010	0.034425	0.000091
XYZfit (3)	0.000010	0.312869	0.000204
XYZobsfit (1)	0.000010	0.000081	0.000033
XYZobsfit (3)	0.000010	0.182673	0.000130
XYZfit_c (1)	0.000010	0.007966	0.000073
XYZfit_c (3)	0.000010	0.011187	0.000085
Model coefficients	0	43	0.048114

Table 2 - Percentage result difference between results generated by previous model and new model with experiment of figure 14 over 32 nodes

The *model coefficients* of table 2 are slightly different to the results illustrated in this table. Instead, these are raw generated co-effieicents that come directly from the Eigen values and vectors, before they are then applied to the input data to generate the results. Whilst a maximum difference of 43% seems very significant (especially in comparison to the 0% minimum and 0.048% mean), it should be noted that all the differences in this large range represent very tiny numbers smaller than 1e-30. After discussions with the BGS around this it was determined that, whilst percentage wise these differences might seem significant, in reality because those numbers are so tiny they make no significant difference to the overall results.

Generally speaking it surprised us how closely the results match between the previous model's Givens reduction approach and SLEPc. One of the strong reasons for the Given's reduction was its stability and perceived accuracy, and we have shown that actually this can be replaced by a much faster, parallel method, which has no qualitative impact on the accuracy of the overall results. This is an important point because there are significant advantages to SLEPc and, in this context, we can benefit from these without sacrificing any degree of accuracy which the geomagneticists were worried about.

Scaling up beyond 10,000 model coefficients

One of the major reasons for doing this project was to have the ability to scale the code beyond the current limit of 10,000 model coefficients. From the results in figure 14 and 15 it can be seen that for the new model, whilst as we increase the amount of parallelism the runtimes are dropping, the scaling is somewhat limited. There are two possible reasons for this – firstly we could be at the limits of strong scaling and as such there isn't enough computation to offset the overhead of extra parallelism. Secondly the SLEPc and PETSc Eigen solver is fairly tightly coupled and as such efficient parallelisation of this at large scale might become more of a challenge. In this section we explore both these aspects in the context of larger systems.

Before doing so however it is important to discuss the issue of memory usage in the new model. As described previously we have explicitly decomposed on the matrix and RHS in order to break through the memory limit barrier that had been limiting the previous code. However, the Eigen solve in SLEPc is a direct solve and this requires the allocation of a significant data structures during this direct solve on each proecss. We did significant experimentation with different solvers, settings of the *maximum projected dimension* (which trades off memory usage and computational recalculation), and even experimented with the MUMPS parallel direct linear solver. Irrespective, SLEPc required the allocation of this memory on each process, which does not decrease as one increases the amount of parallelism and as such this had now become the memory limit of the new model.

Number of parameters	Direct solve size (MB)
10,000	1498
20,000	9543

30,000	23908
40,000	38200
50,000	57300
60,000	85950
70,000	114600

Table 3 - Memory usage requirements of direct solver by the number of parameters

A major limit of the current code was it's inability to scale beyond 10,000 model coefficients due to memory limitations. Having decomposed the matrix and RHS data structures across processes this limitation has been mitigated. However the direct eigen-solve in SLEPc requires the allocation of a significant data structures during this direct solve on each process, especially when searching for all eigenvalues and vectors as we require here. Whilst there are options for experimenting with the maximum projected dimension (which trades off memory usage and computational recalculation), and replacing the solver with the MUMPS parallel direct linear solver, from experimentation we found that these had no impact on the very large memory usage of SLEPc.

Table 3 illustrates how the memory usage of the direct solver grows as the number of model coefficients is increased. This memory requirement is in addition to other data structures including the matrix and RHS, and is not parallelisable. Therefore irrespective of the number of processes, this amount of memory must be allocated per process and is a very serious limitation. Bearing in mind most nodes in ARCHER, the Cray XC30 used for this work, have 64GB RAM in total and 32GB per NUMA region, ways round this needed to be found. Whilst there is a trend to increase the amount of memory per node with more modern machines, memory usage is still a limitation however as we reach large coefficient sizes.

Hydridising the code with OpenMP

Based on the experiments done in figure 15, the building of the matrix and RHS scales well as we increase the amount of parallelism. However, the direct solve doesn't scale quite so well and scaling from 16 to 32 nodes in figure 15 only decreases the runtime of the solver by 8% but the matrix and RHS building by 28%. Therefore, as the memory limits of the direct solve are on a process by process basis, it made sense to hybridise the code using OpenMP as a reduced number of processes can run per node, but within each process we can still take advantage of all the cores using thread level concurrency. The approach adopted was to apply OpenMP to all aspects of the code, including the matrix and RHS building. Whilst there has been some work done in PETSc to support hybrid OpenMP/MPI parallelism this isn't particularly mature and-so we didn't enable this in the code but instead the solver will just solve on a process by process basis without any thread level concurrency. One of the nice things about PETSc is the fact that it is trivial to modify the type of parallelism by selecting the appropriate data structure, so in the future if the hybrid versions become more mature then only minor code changes are needed.

In order to support this we stuck with the rule that the matrix and RHS are decomposed based upon processes, i.e. 10 processes means that these data structures are split into 10 chunks, one per process. But the input data itself is decomposed across the thread of a process and each thread will work on the processes' data structure chunk but with a separate subset of the input data to process.

Figure 16 sketches the hybrid OpenMP code for one of the kernels involved in building the normal equations. All OpenMP calls are loop based directives, and in this instance we are performing an OpenMP reduction between the threads once local iterations of the loop have completed. As before, the matrix and RHS data structures are decomposed on a process by process basis and when building

```
!$omp parallel do private(j, dw, il) reduction(+:rhs_norm_equations)
    do j=1, lines_in_y
      ....
      do il=1, number_of_rows
        rhs_norm_equations(il)=rhs_norm_equations(il)+.....
      end do
    end do
!$omp end parallel do
```

Figure 16 - Illustration hybridised OpenMP/MPI kernel

the normal equations, the input data is split up between the OpenMP threads. Each thread works on the processes' data structure chunk but with a separate subset of the input data to process. This is important, because it means we can start MPI in *funnelled* threading mode rather than the much slower *multiple* mode. It should also be noted that, due to the non-associativity of floating point arithmetic, using OpenMP reductions can generate slightly different results from one threading level to another. However, this is still well within tolerance and we also provide additional code to perform the reduction in guaranteed order, via local variables and explicit addition but with some minor degradation in performance, if this is critical to the end user.

Nodes	2 threads PP	4 threads PP	6 threads PP	12 threads PP
40	42.99	29.52	24.59	20.33
36	44.19	30.69	25.86	21.72
27	48.61	34.39	29.69	25.96
18	56.05	41.82	37.28	34.49
9	78.95	64.17	60.55	59.72
5	127.4	109.47	104.34	111.05
3	245.82	189.65	195.11	190.12

Table 4 – Matrix and RHS building time changes wrt threads Per Process based on 500,000 data points & 8000 parameters

Nodes	2 threads PP	4 threads PP	6 threads PP	12 threads PP
40	55.7	44.8	40.29	36.68
36	54.67	43.76	38.72	39.71
27	49.32	42.07	40.4	41.63
18	48.04	29.84	43	39.62
9	42.07	42.19	48.01	68.15
5	61.72	66.64	82.3	122.34
3	103.36	113	144.35	212.99

Table 5 - Solver time changes wrt threads Per Process based on 500,000 data points over 8000 parameters

Nodes	2 threads PP	4 threads PP	6 threads PP	12 threads PP
40	2.95	2.6	2.56	2.46
36	2.76	2.63	2.69	2.45
27	2.89	2.54	2.71	2.83
18	3.07	2.86	2.94	2.51
9	3.2	3.03	2.6	2.86
5	2.94	3.05	2.68	2.86
3	2.74	2.65	2.84	2.89

Table 6 - IO time changes wrt threads Per Process based on 500,000 data points over 8000 parameters

Nodes	2 threads PP	4 threads PP	6 threads PP	12 threads PP
40	119.02	88.05	76.94	66.76
36	118.62	88.89	76.42	71.27
27	117.41	90.94	81.96	77.73
18	124.13	85.55	92.63	83.97
9	140.69	120.6	120.51	138.5
5	208.76	190.75	199.14	244.41
3	369.42	317.71	352.72	414.8

Table 7 - Total runtime changes wrt threads Per Process based on 500,000 data points over 8000 parameters

We found that utilising threading made a difference to overall performance with the model and ran a smaller experiment than those detailed previously, with 500,000 input data items and 8000 model coefficients. Table 7 illustrates how the run time of the model changes with respect to the number of threads per process for this experiment. For each row, for instance the 40 nodes row, the overall number of cores remains unchanged (960 in this case), and with two threads per process we have 480 process each running over 2 threads, out to twelve threads per process resulting in 80 processes each with 12 threads. It isn't as simple as saying one level of threading is *better* or *worse* than another. Instead, it depends heavily on the overall amount of parallelism (number of nodes) and it can be seen that when using 40 nodes the run time can be almost halved by the simple configuration change of going from 2 threads per node to 12 threads per node. However over three nodes going from 2 to 12 threads per node reduces performance by almost a third.

To break it down further, Tables 4, 5 and 6 show the run time, for each configuration, for building the normal equations, solver time, and IO respectively. Based on the solver run times it can be seen that we are hitting the limits of strong scaling with this experiment size as, at small node sizes it is advantageous to use more parallelism and at larger numbers of nodes less, for this problem size this factor dominates at smaller numbers of nodes. For the normal equation building, irrespective it is advantageous to use more threading where possible and this is because it reduces the amount of MPI communications needed, with this problem size this factor dominates at larger numbers of nodes. As all processes potentially need to communicate with all other processes, this can be an expensive.

The code itself will automatically pick up the number of threads based on the OpenMP number of threads environment variable (i.e. OMP_NUM_THREADS), so no additional arguments need to be passed to the code.

Scaling to 100,000 model parameters

Based on this hybridisation we can now run the new model with problem sizes much larger than the 10,000 coefficients that the previous model was limited to. Figure 17 illustrates an experiment, over 40 nodes of ARCHER, as we scale the number of model coefficients up to 100,000. There are a number of caveats and noteworthy points to be highlighted here. It can be seen that there is a sharp increase in run time at 40,000 coefficients and this is where we had to switch, due to memory limits, from a process per NUMA region to a process per node in order to fit into memory. At 50,000 coefficients we started using the large memory nodes on ARCHER, which contain 128GB of RAM, and provided some extra headroom for these runs. In terms of modern computing, 128GB of RAM isn't a particularly large amount, and once we reached 70,000 coefficients this memory was exhausted. In order to model problem sizes of 70,000 coefficients and above, we split the problem in two when it came to eigen-solving, first finding the *n/2* largest eigenvalues in magnitude and applying these and

their corresponding eigenvectors to the RHS, and then finding the *n/2* smallest eigenvalues in magnitude and applying these and their corresponding eigenvectors to the RHS. This worked well, and actually resulted in a slight decrease in run time because at that stage two smaller solves was faster than one larger one.

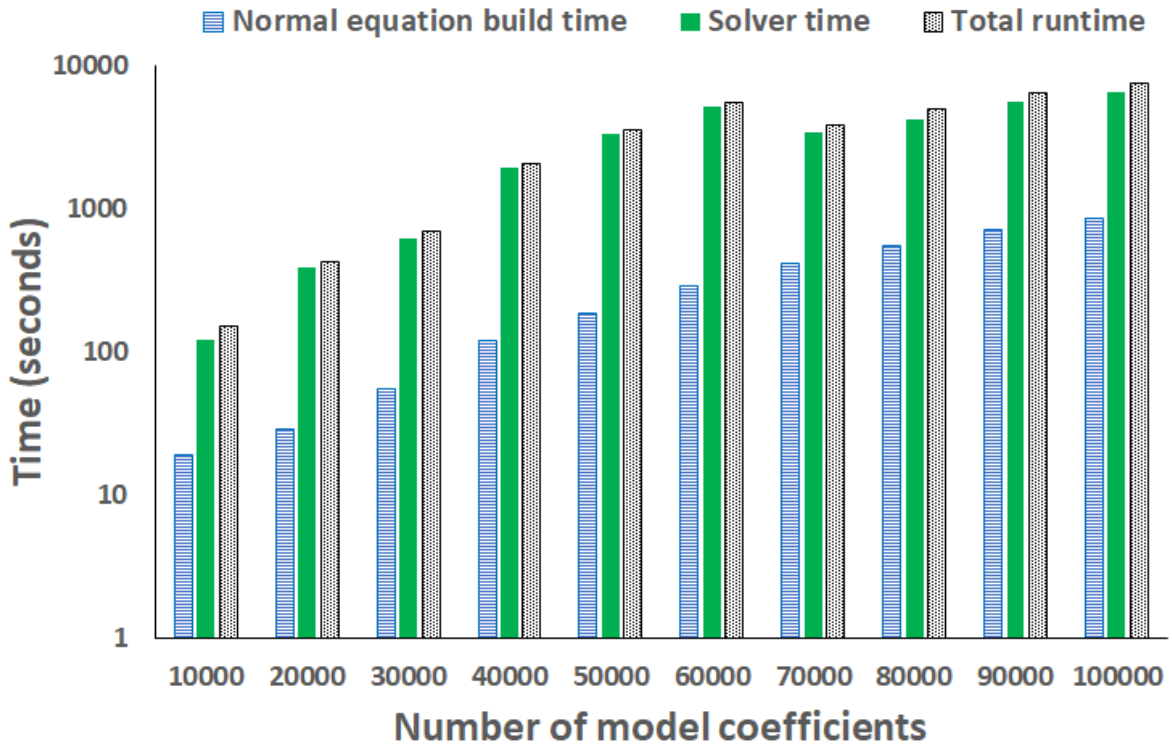

Figure 17 - Scaling the number of model coefficients over 40 nodes of ARCHER

It should be noted that we tried a number of approaches to finding different portions of the spectrum, such as those closest to a target and then iteratively increasing the target. None of these worked very well and the only reliable approach was to split the solve in two and finding the *n/2* largest and *n/2* smallest eigenpairs. In our mind this is the current limitation of our model and, as it currently stands, when the problem size reaches a point where the two separate solves run out of memory, most likely at around 120,000 coefficients on ARCHER further investigations will need to be performed. There are a number of points to bear in mind here, in addition to the obvious point that larger amounts of memory are becoming much more commonplace which mitigate this issue somewhat. Firstly we are finding all the eigenvalues and then throwing away the smallest ones beneath a threshold. At 100,000 model coefficients this represents a large number that are being discarded and, as the direct solver memory usage is determined by the number of eigenvalues being searched, we could likely reduce the number of eigenvalues being searched because of the threshold and improve the situation. Secondly, as we reached 70,000 coefficients PETSc had to be compiled with 64 bit matrix indices, this isn't a problem in itself but does illustrate the significant size of the matrix and general problem we are working with here.

Parallelising the damping values

An important assumption made until this point is the fact that there is only a single damping value required to run the model with. Once the matrix and RHS has been built, a damping value is applied to it and then the solver run for these resulting values. However, fairly often the scientists wish to apply multiple damping values one after another. In this case there is still a solve and results write out

for each individual damping value, but crucially there are multiple solves and result file writes (one per damping file) performed. As the handling of each damping value is embarrassingly parallel with any other damping value, the previous model, whilst the solver was sequential, distributed the *d* damping values across the processes and ran *d* sequential solvers concurrently to effectively solve for each distinct damping value at the same time. We have changed the model to parallelise the solver, which is very important as it allows larger problem sizes, but crucially geometrically decomposing the matrix and RHS and running the solver in parallel across all processes now means that each damping value must be solved sequentially.

As described previously, once at a specific level of parallelism, adding extra to the experiments of figure 15 only provided very limited speed up to the solver. Therefore it made sense to support a multi-level approach, where the matrix and RHS are split up into regions and a specific rank of a region

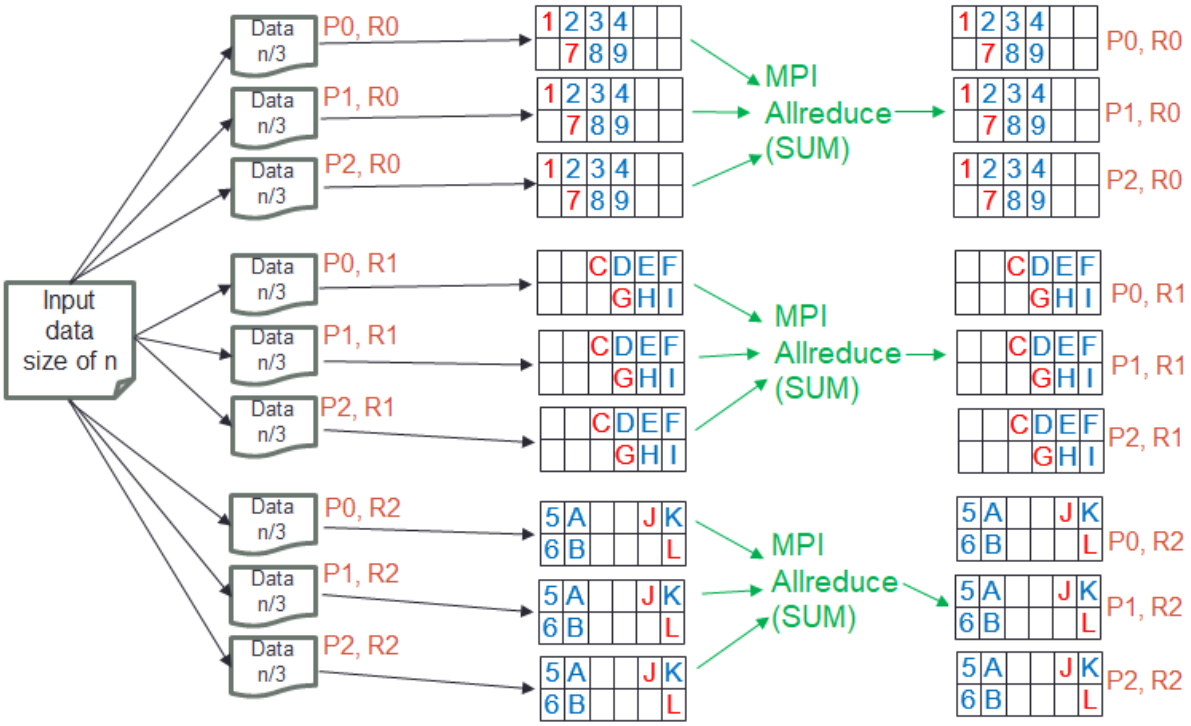

Figure 18 - Modified matrix and RHS building to support regions for parallel damping values

(e.g. rank 0 in each region) works together to perform the solve for a specific damping value and other ranks (e.g. rank 1) then work together to perform the solve for another damping value.

This required modifications to the matrix and RHS building, which is illustrated in figure 16, where processes are part of a region (an MPI communicator) and the input data is split up across these processes (and can then further be split up inside a process using threading.) Each process will work on the region's subdomain and contribute its chunk of input data before an all reduce is performed within the region (across the region MPI communicator) to sum up the values in the region across all processes. The result of this is that all processes within a region then end up with the same chunk of matrix (and RHS) all with the same matching values and then each process can apply separate damping values to its chunk. In the example of figure 16, process 0 of region 0 (P0, R0) will work with process 0 of region 1 (P0, R1) and process 0 of region 2 (P0, R2) to solve for the first damping value, concurrently process 1 of region 0 (P1, R0) will work with process 1 of region 1 (P1, R1) and process 1 of region 2 (P1, R2) to solve for the second damping value etc.

If there are more damping values than there are processes in a region, then these will wrap around and if there are more processes in a region than there are damping values then the extra process will be idle. The number of regions (or number of damping values to process concurrently and from this the number of regions is calculated by the code) is a runtime command line argument. Effectively, the default option of processing 1 damping value at a time (as assumed in previous sections) is where the number of regions equals the number of processes and each region contains just one single process.

Conclusions

In this technical report we have explored the use of PETSc and SLEPc in solving a system of normal equations for modelling the earth's magnetic field. We have demonstrated significant benefits to replacing the bespoke eigen-solver of the previous MEME model with that provided by SLEPc, and have shown that, at no qualitative different to the output result accuracy, very significant improvements in performance are possible by leveraging this popular library. In terms of problem size, the new model is capable of modeling systems very significantly larger than those that were previously attainable. However when it comes to large numbers of coefficients, and more generally using SLEPc for finding very many eigenpairs in a large system, there are caveats and limitations around memory usage and a number of mitigations are required to work around these. ARCHER, the XC30 used for this work is over five years old now and inevitably more modern machines with larger amounts of memory will be impacted less by this, but still it is important to bear in mind as scientific ambitions of this model and use of SLEPc in general will only continue to grow.

Adopting PETSc and SLEPc has had implications far beyond the specific solver itself and we have also described our approach for building the symmetric matrix of normal equations in a distributed fashion. We demonstrated an approach that requires minimal process coordination, gives reasonable load balance and avoids any replication of computation.