HJCFIT Archer eCSE end of project technical report

Jens Hedegaard Nielsen

Raquel Alegre

Remis Lape

James Hetherington

Lucia Sivilotti

Abstract

This report details the implementation of the Archer eCSE project "Parallelization and porting of single-channel analysis tools to the high-performance computing platform". In this project HJCFIT, which is a part of the DCProgs suite of tools, has been transformed from a single-process library running on desktop computers to a multi-precision library that can utilise a full Archer node and is thus 14 times faster than the original serial version. We have implemented multi-precision arithmetic, made the code easier to use on high-performance systems and made several other improvements to the overall codebase.

Introduction

HJCFIT is a library for the maximum likelihood fitting of kinetic mechanisms to entire sequences of single channel open and shut intervals. The HJCFIT method estimates the Q-matrix, a table of the rate constants describing each transition between pairs of kinetic states. The likelihood is calculated using an exact correction method for events that are missed because they are shorter than the temporal resolution of the recording. In a typical run of the software, the likelihood is calculated on the basis of an initial guess of the Q-matrix and optimised using a standard simplex algorithm to find the maximum likelihood Q-matrix. The optimisation may run for several minutes to hours or even days, depending on the complexity of the kinetic model. This makes it clear why running the code in parallel and on a queue based system is attractive.

Speed related optimisations

This section describes the work performed to complete work-packages 2, 3, 4 and 5. In the proposal three different levels of possible loop parallelization were identified. The log-likelihood calculation consists of a sum of log-likelihoods for independent experiments, typically obtained at different concentrations of an agonist. Parallelisation over these independent experiments seems attractive, but the number of experiments is typically limited to 4-5 in real world examples, limiting the parallel speedup.

The log-likelihood of each individual experiment is a sum of log-likelihoods for one or more bursts. A single burst is a series of one or more openings (note that in this report we will use the term burst to cover groups of openings at all concentrations, eg to include also what the channel literature would refer to as clusters). In an experiment each burst is separated from the next by shut intervals longer than an appropriately chosen critical length. The division into bursts is done to ensure that all openings within a burst are from the same ion-channel. However, it's not possible to ensure that different bursts are from the same channel, as channels may stay closed for a long time while another channel is opening and closing rapidly.

The log-likelihood calculation for an individual burst is independent of other bursts and the log-likelihood for one experiment was previously implemented as a serial loop over all bursts. Finally the log-likelihood calculation for each burst is a multiplication of a series of matrices: one for each individual opening and closing, plus the initial and final vectors.

Parallelisation can be implemented at both these levels- over bursts (mentioned in WP 4) and inside bursts over matrix multiplication (mentioned in WP 3). Most straightforward is the loop over bursts as these are fully independent. The matrix multiplications must naturally be done in the same order as in the serial case complicating matters slightly. As we shall see below it is beneficial to parallelise over both the bursts and individual openings to obtain good load balancing.

A third level of paralelism, within the individual matrix multiplications, can be found. It would be easy to exploit the parallelization built-in to the linear algebra library Eigen (http://eigen.tuxfamily.org/) used in HJCFIT. Eigen has built-in support for parallel linear-algebra. However, the matrices used in HJCFIT are small (on the order of 10x10 elements) so the parallel speedup within the matrix multiplications is very small. No performance difference was found with and without Eigen's parallel linear algebra enabled. We therefore chose to explicitly disable Eigen's internal parallelization so as not to interfere with our higher level parallelization.

In the original proposal we suggested rewriting the code to merge and flatten the two loops over experiments into one, in order to make the load balancing more efficient. However, this would result in a significant change to the code which will make the implementation differ significantly from the existing one. Instead we chose the strategy documented below to optimise the two levels using a hybrid OpenMP/MPI in which we combined the loop over bursts with the parallelization of individual burst likelihood calculations.

We have not integrated performance regression in the test suite as we had originally suggested in work-package 2. This method is most suitable for monitoring performace of individual functions. On consideration, we chose to benchmark and optimize the performance of complete simulations, as this is more likely to give us useful information on improving performance of the parallel code.

Low level OpenMP parallelization

We chose to implement the parallelization over bursts and the matrix multiplication within bursts using OpenMP. The parallelization over bursts involves a fairly minimal amount of communication between threads as:

- the burst data can be pre-distributed
- the only parameter needed to calculate the log-likelihood for a burst is the Q-matrix
- the total log-likelihood is a sum over the likelihoods of individual bursts

The parallelization of log-likelihood calculation over an individual burst requires more communication as the individual matrices calculated by individual threads need to be multiplied in the reduction step. As this is the innermost level of parallelization, it is most natural to parallelise this using openMP which also simplifies the sharing of matrices.

Ideally, in order to reduce any load imbalance, we want the runtimes for each experiment to be as similar as possible, so that we get the maximum speed-up from performing the log-likelihood calculation for each experiment in parallel. By benchmarking the runtime of a single evaluation of the likelihoods for the individual experiments, we have obtained an understanding of how well balanced the load is in these calculations in the first instance and have subsequently performed a benchmarking of them with low level parallelization applied.

Measurements of the performance time of a single likelihood calculation for the individual experiments in serial implementation were: 5.9 ms, 6.7 ms, 5.8 ms and 4.7 ms (benchmarked locally on a Macbook Pro 2.8 GHz QuadCore). We note that while the runtimes are of similar magnitude, they still vary significantly. We will return to this question of the runtimes of the individual experiments after the OpenMP parallelization.

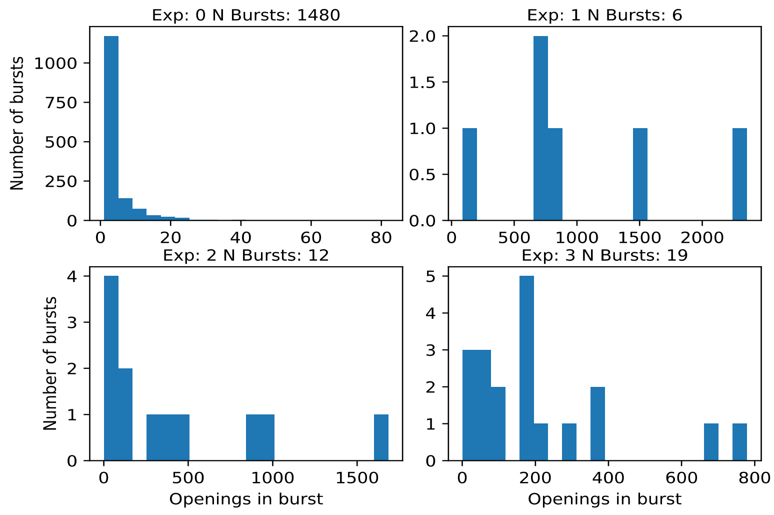

Figure 1 Example of the distribution of burst lengths in 4 experiments used in a likelihood calculation.

It would be easy to implement the parallelization only at the lower layer, over the calculation of likelihood for individual bursts. However, by looking at the histograms of bursts length in Figure 1, it is clear why this is unlikely to work very well.

In this particular example, two distinct types of experimental records can be identified. The first record (Exp 0) consists of a large number (1480) of very short bursts (less than 10 transitions) while the other three records have much smaller number of bursts (<20) consisting of hundreds to thousands of transitions each but with a rather large spread of burst-length. This makes it difficult to obtain good load balancing by only relying on the parallelization over bursts as the runtime for these experiments will be dominated by the few long bursts.

On the other hand, the parallelization over the matrix multiplication within the individual burst is unlikely to yield good results, unless the series of Q-matrix products is significantly longer than the number of threads. After benchmarking two different implementations- parallelisation over bursts or over intra-burst matrix multiplication- individually, it become evident that the parallelization over matrix multiplications results in a significant slowdown, if performed for all series with runtime increasing with the number of threads. On the other hand the parallelization over bursts results in a significantly smaller speedup when the number of bursts is small, as the runtime is completely dominated by the longest bursts.

Thus, we must control which of these parallelisms we use, depending on the problem at hand: finding the best parallelisation, while avoiding the unnecessary slowdown from attempting to use low levels of parallelism. Note that the optimal solution will depend on the properties of the data and the size of the Q-matrix, something that may change in future work.

After a number of experiments, we settled for a solution where we selectively parallelise over either the bursts, the matrix multiplications or perform no OpenMP parallelization. This is implemented using the standard OpenMP `if` pragma.

Based on the current example the likelihood calculation is parallelised over the bursts if there are more than 100 bursts in a record. The switch-over is selected to separate Experiment 0 in Figure 1 from the rest of the experiments. Better performance may be obtained for other datasets by tweaking the switch-over parameter. Then, if the outer loop is not parallelised, we call a parallel implementation of the internal matrix multiplications. This parallelises the calculation of the matrices, as well as the matrix multiplication.

As the matrix multiplication is non-commutative, the multiplications must be done in the same order. To this end we implemented the parallel multiplication by storing the partial matrix products for each OpenMP thread in a C++ std::vector which is subsequently multiplied by the initial and final vectors. There is an additional overhead associated with the creation of a vector of matrices and the two step matrix multiplication, so we perform this in parallel only when the number of matrices to multiply is larger than 100.

This matrix multiplication is often likely to overflow the standard floating point numbers. The existing serial code has a simple functionality to prevent overflow within the matrix multiplication. Following each matrix multiplication, the maximum element is found and provided that it is larger than 1050, the matrix is divided by 1×10^{50} and the exponent stored separately.

This simple solution does efficiently prevent overflow in the likelihood calculation, but does come with the cost of reduced precision. When the matrix multiplications are performed on several threads the final multiplication of individual thread result matrices may still overflow even if the maximum element of each matrix is less than 1×10^{50}.

To prevent this, the new parallel implementation keeps each matrix smaller than 1×10^{20} and this is sufficient to prevent an overflow when the calculations are performed within a single Archer node (typically using 4-8 OpenMP threads combined with 3-5 MPI processes; see below). However, developing a better solution should be considered in the future.

To summarise, the parallelization using the combined burst and matrix multiplication approach described above resulted in runtimes of 2.6 ms, 3.0 ms, 2.8 ms and 2.8 ms for the 4 experiments respectively (benchmarked locally using 4 OpenMP threads on a QuadCore MacBook Pro 2.8 GHz). Relying only on the parallelization of over bursts results in run times of (2.5 ms, 3.4 ms, 3.7 ms and 2.7 ms) respectively, with a significantly worse load balancing.

MPI parallelization over experiments.

As mentioned above, the log-likelihood over individual experimental records can be calculated independently. In the existing codebase, the loading of data and individual log-likelihood calculations are performed in the Python layer of the code. To keep the code flexible with respect to loading of data for multiple experiments, we chose to keep this loop in the Python layer. We have experimented with multiple high-level parallelization solutions in Python. We first implemented the parallelization using IPython parallel (http://ipyparallel.readthedocs.io/en/stable/) which is internally based on communication using ZeroMQ (http://zeromq.org/). This provides a nice high-level interface for the implementation of parallelization over the experimental records. However, we found that the communication overhead was too large. When running examples on Archer within a single node, roundtrip time for communication was around 1 ms, which is of the same order of magnitude as the runtime of individual likelihood calculations. We did not further explore possible solutions to speed up the communication using ZeroMQ.

Instead we chose to implement the parallelization using MPI and MPI4Py (http://pythonhosted.org/mpi4py/), which yielded significantly better performance. The resulting speedup can be seen in Figure 1 .

Improved dynamic memory allocation.

HJCFIT uses Eigen (http://eigen.tuxfamily.org/) for its linear algebra calculations. Eigen is well-regarded, established library that allows us to write modern, readable and highly performing C++ code. However, during the profiling of the code for the implementation of parallelization in work packages 3, 4 and 5, we found that a significant amount of time in realistic examples was spent within memory allocation in Eigen.

This naturally prompted us to investigate the various options for memory allocation in Eigen. Eigen supports either fixed size matrices with size defined at compile time on the stack, or dynamic matrices with runtime-defined sizes allocated on the heap. The dynamic matrices are allocated at runtime and this results in a significant performance drop. HJCFIT heavily uses runtime-defined matrices sizes as internal math is performed on matrices with sizes given by the number of open/closed states in the particular model being solved, which is naturally a runtime parameter, as it depends on the specific experiment.

Two solutions for optimising the memory allocation are possible. When refactoring the code, it is possible to reduce the number of dynamically allocated matrices by reusing already allocated matrices. In the present code, matrices are allocated within the likelihood calculation. To refactor this would require significant changes to the codebase. The second option is to statically allocate matrices sufficiently large for all relevant problems. Eigen has built-in support for dynamic sized matrices with a compile time fixed maximum size. Therefore, one can easily define a type of matrices which act fully as dynamic allocated matrices, provided that they are smaller than a compile time defined constant. These will then act as drop-in replacement for the existing dynamically allocated matrices.

Straightforward implementation of this option gave a substantial performance increase. However, statically allocating matrices with sizes larger than needed does come at a cost of higher memory usage. In the case of HJCFIT the memory requirement is fortunately fairly low for models being currently fitted, with a typical single threaded memory usage of less than 50 MB. Thus, we were confident in implementing static allocation of matrices in the mainline codebase. Initially we chose to allocate matrices with a static size of 50x50.

However, it is simple to change the size and recompile the code with either a different static size or even with fully dynamic memory allocation. Furthermore, verification of the number of open and closed states in the model is added, and this ensures that a meaningful error message is raised, if solution of a model with too many open and closed states is attempted. The performance gain from the static memory allocation can be seen in Figure 2. The options for dynamic memory allocation can be found in the Eigen documentation (https://eigen.tuxfamily.org/dox-devel/group__TutorialMatrixClass.html).

Benchmarking

The fitGlyR4.py job, used for benchmarking HJCFIT performance, represents a typical example of a maximum likelihood fit. The dataset contains open and shut time intervals from four separate single channel records at different agonist concentrations. The first record (mentioned as Experiment 0 above) is obtained at low agonist concentration and is composed of a large number of short bursts of single or multiple openings. The other three records obtained at higher concentrations of agonist contain much smaller numbers of bursts. Bursts at high concentration are much longer and are formed of thousands or even tens of thousands of openings. This selection of experimental records gives the most information about ion channel behaviour and is optimal for understanding channel sojourns in different kinetical states across a range of agonist concentrations. The kinetic scheme in the script has been successfully used to describe glycine receptor behaviour. The scheme incorporates 10 states with 22 rate constants for transitions between the states, but it is constrained by the requirement of maintaining microscopic reversibility, and therefore only 14 free parameters are estimated directly. This scheme is of medium complexity and more complex schemes (with a number of free parameters up to 27) are routinely tested.

Figure 2 shows the number of likelihood evaluations performed per second by the original library, after improved memory allocation and after improved memory allocation, MPI and OpenMP parallelization. The improved memory allocation gave a speed-up from 19 to 39.3 likelihood calculations per second (approximately 2-fold). On top of this, we applied the OpenMP parallelization over bursts and matrix multiplications. The OpenMP parallelization resulted in a speed-up from 39.3 to 85.9 likelihood evaluations per second when using 6 threads (approximately 2.2-fold). We noted that the OpenMP implementation does have significant overhead when running the calculations on a single thread which drops the performance from 39.3 to 35.4 likelihood evaluations using a single thread relative to no OpenMP. This is most likely due to the additional overhead of the two step approach, where we create a one element vector of partial matrices, which is then multiplied by initial and end vectors.

MPI parallelization over 4 experiments or 4 processes without any OpenMP parallelization results in an increase from 39.3 to 119.8 likelihood evaluations per second giving a 3.0-fold speedup. We observed that the total speed-up due to parallelization going from purely serial code (at 39.2 evaluations per second) to 4 MPI processes and 6 OpenMP threads is 6.9-fold, but the product of the 2 individual speed-ups given above is only 6.6-fold (2.2 x 3). This is due to the improved load-balancing and more similar runtimes of the individual experiments when running with OpenMP parallelization.

Finally, the total speed-up was from 19 to 272.3 likelihood evaluations per second giving a total combined 14-fold improvement on a single Archer node.

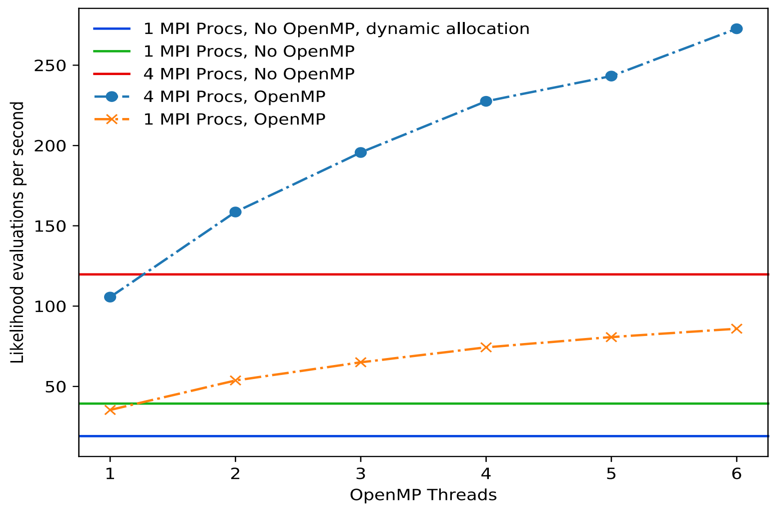

Figure 2 Benchmark of running fitGlyR4 on one Archer node, showing the number of likelihood function evaluations per second. The combined effect of improved memory allocation and hybrid parallelisation is a speedup from 19 to 272 function evaluations per second,

Accuracy related changes

Fortran issues

The HJCFIT library is a re-write of a previous Fortran 77 code, begun by the UCL team prior to this project. We noticed that, despite the identical result obtained for a set of single likelihood calculations, the two codes often produced different results for long running likelihood calculations. The likelihood is optimised over a multi-dimensional Q-matrix with usually more than 10 free parameters and it is expected that the likelihood surface might have several local maxima. As the likelihood is evaluated, slight differences in the calculation may result in different optimisation pathways being selected and different end-results obtained. This can be due either to differences in the optimisation algorithm used or to differences in the likelihood calculation. While this was not strictly a part of work-package 6 and 7, we felt it was important to ensure that the code behaves as expected and that no new issues were introduced compared with the Fortran code.

The optimisation is done using a downhill simplex algorithm in both cases. In the new library a SciPy (http://scipy.org/scipylib/index.html) implementation of the Nelder-Mead algorithm is used whereas the Fortran code base uses its own custom implementation which it slightly tweaked for the specific likelihood problem. This might have potentially resulted in different maximum likelihoods being obtained: the different optimisation algorithms made it more difficult to establish where differences arise in the likelihood calculation. To investigate this, we therefore calculated the likelihood manually using both implementations focusing on points where the likelihood optimisation diverged.

We found examples were the log-likelihood values do differ after about 8 to 9 digits. After careful inspection of both codebases and interactive debugging to step through the implementation, we established that this divergence was due to differences in the floating point precision used in the two implementations. While both implementations perform all calculations in double precision, the Fortran version used a number of parameters stored in single precision which were being automatically cast to double precision during the calculation. The value of stored digits beyond single precision is undefined and random: the differences in results can therefore be traced to this bug in the Fortran code.

For example the initial and final CHS vectors (initial and final state occupancies) depend exponentially on the temporal resolution (the shortest resolved time interval) and the critical time interval used to separate individual bursts. The values of these two parameters were cast from single to double precision in the Fortran code.

Multi-precision

In a number of cases, HJCFIT log-likelihood calculations failed as a result of floating point precision issues. In the existing codebase, this is prevented by the optional use of long-double floating point numbers rather than regular doubles. This typically gives an extended precision from 64 bit floats to 80 bit, but the implementation of long-doubles is compiler- and platform-dependent. Specifically, long-doubles are identical to doubles when compiling with MS Visual studio on Windows. In addition, there are two other issues with this solution. There is no guarantee that 80 bit long-doubles have sufficient precision to resolve all floating point inaccuracies, and re-implementing the entire code using long-doubles has a significant impact on performance.

In work-package 7, we therefore implemented an alternative solution. We focused on a specific issue where the log-likelihood calculation fails. A root finding is performed as part of the approximate survivor function used for missed event correction. This progresses through multiple steps, first identifying the upper and lower bounds of all the roots.. Subsequently the individual roots are bracketed and exactly identified. It has been observed that the root-finding may fail for specific guesses of the Q-Matrix. This may be because two roots are within floating point precision of each other or because the bracketing or bounds location fails. We identified a particular instance where root finding was failing and used it as a test to implement a more robust solution.

The particular error we investigated happened only when compiling with GCC (not Clang) and was sensitive to the exact iteration strategy implemented when the roots are bound from above and below. In brief, finding the bounds involves finding the eigenvalues for a matrix that depends on the guesses for the upper and lower bounds.   This may fail if the matrix is close to being singular. It is therefore in principle possible to avoid this particular issue by picking other guesses for lower and upper bounds. However, this is not a robust solution as the issue is likely to reoccur during the optimisation process.

Increasing the numerical precision can resolve the issue but, as argued above, it is too computationally expensive to do this in general. We have therefore implemented automated fallback to higher precision as suggested in work-package 7. As the existing code raises an exception when Eigen fails to find the eigenvalues, it is possible to wrap the call to the standard double eigenvalue solver in a try block and calling a multi-precision version of the same algorithm. This means that the runtime cost is minimal, as most of executions of the code will go through the standard algorithm with only an extra try block. The number of likelihood evaluations that will require the expensive multi-precision calculations is a small fraction, so little performance is lost in these fall-backs, but this will prevent failure of a long running optimisation at random points in the process.

The evaluations are performed using GNU MPFR (http://www.mpfr.org/) which implements multi-precision floating point arithmetic on top of GMP (https://gmplib.org) "The GNU Multiple Precision Arithmetic Library" which implements multi-precision integer arithmetic. In practice we are using Eigen's support for multi-precision (https://eigen.tuxfamily.org/dox/unsupported/group__MPRealSupport__Module.html) which depends on the nice C++ interface to MPFR (http://www.holoborodko.com/pavel/mpfr/).

So far, we implemented multi-precision support only for this specific algorithm, but as more specific issues are identified, it will be easy to add multi-precision support in a similar way to other specific sections of the code, making the code fall back to the multi-precision if needed. We expect all issues caused by floating-point precision to be of the form where an exception is raised which can be caught and where fallback to a multi-precision implementation is feasible.

We did not port the entire code to run with GMP and MPFR, as suggested in work-package 6. Early benchmark of simple operations indicated that this is likely to result in a very significant increase in runtime of the order of 10 times or more. Unfortunately, Eigen does not yet implement multi-precision for all the linear-algebra algorithms used. Thus, we chose to focus on the application of multi-precision to individual functions as explained above.

Note that the code can still be compiled without support for multi-precision on platforms that do not support it. GMP is not supported on Windows but MPIR (http://mpir.org/) should in principle be a suitable replacement which works on Windows. We have however not tested it as part of this project. The code is written in a way that ensures that it can be compiled conditionally without multi-precision support and remains supported in this form on Windows.

Minor improvements

CI on Windows and Travis

HJCFIT has a fairly extensive test suite implemented directly in the C++ layer and has behaviour-driven tests using Behave (http://pythonhosted.org/behave/) in the Python layer. At the start of the project the tests were executed on the UCL RSDT Jenkins instance (http://jenkins.rc.ucl.ac.uk/) and on an instance of the UCL cluster Legion. However, the CI did not test Python3 compatibility and neither did it test the long-double implementation.

In addition, the code was written to be Windows compatible but was not automatically tested on Windows.

As part of the eCSE project the CI has been extended to Windows on the RSDT Jenkins as well as running automatically on Travis-CI (https://travis-ci.org/DCPROGS/HJCFIT). The CI run on Travis ensures that the code works with both Python 2 and 3 and compiles with long-double support.

Automatic Documentation build

At the beginning of the eCSE project, HJCFIT had quite good documentation consisting of a user guide written in restructured text using Sphinx (http://www.sphinx-doc.org/en/stable/) and Python API documentation using Sphinx autodoc (http://www.sphinx-doc.org/en/stable/ext/autodoc.html) along with C++ API documentation using Doxygen (doxygen.org) and integrated into Sphinx using Breathe (http://breathe.readthedocs.io/en/stable/). In addition, the repository contained a number of Jupyter (http://jupyter.org/) notebooks demonstrating the use of HJCFIT's Python API. An HTML version of the documentation has been manually built and deployed to github repository (http://dcprogs.github.io/HJCFIT/). In addition build and installation of the code was documented using the github wiki.

As part of this eCSE project three significant changes were made to the documentation infrastructure.

1. All Jupyter notebooks were moved to the exploration subdirectory of the repository where they are automatically executed, converted to restructured text (rst) and included in the documentation. This makes it simple to write examples that are automatically included in the documentation and can contain prose, code and graphs. The automatic execution of notebooks as part of the CI run ensures that examples are compatible with both Python 2 and 3 as well as with the long-double build of the code.
2. The documentation in automatically build and deployed to github pages as part of the CI run. Documentation for the latest commit in the master branch is deployed to HJCFITmasterdocs (https://dcprogs.github.io/HJCFITmasterdocs/) and documentation reflecting the latest commit to the development branch is deployed to HJCFITdevdoscs (https://dcprogs.github.io/HJCFITdevdocs/).
3. The build documentation was transferred from the github wiki into the main documentation.

This implements a part of work-package 9. As work-package 8 was not implemented there are no notebooks documenting MCMC.

Deployment on Archer

As part of the project some work has been done to streamline and document the deployment of the code on Archer. This is part of work-package 1.

Markov-chain Monte Carlo

Work-package 8 was not implemented due to lack of time.

Input/output formats (work-package 10)

The code is most effectively invoked via Python scripts, providing for effective management of IO.

Documentation (work-package 11)

The new online documentation at https://dcprogs.github.io/HJCFITmasterdocs/ is supplemented by this report along with changes to the documentation reports the changes made. We intend to produce a paper describing the improvements to HJCFIT and publish in suitable journal. This report will form a base of a first draft.

Conclusion

In the scope of this Archer eCSE project we ported HJCFIT to Archer and parallelised the code to produce a 14 times speed-up. We have added support for multi-precision floating point libraries that solve long standing issues within the code. Furthermore, we have streamlined the documentation of the code with additional examples build from Jupyter notebooks and automatic deployment.

Acknowledgement

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