Enhancing scalability of the gyrokinetic code GS2 by using MPI Shared Memory for FFTs

Lucian Anton*, Ferdinand van Wyk†‡§, Edmund Highcock†, Colin Roach‡ and Joseph T. Parker¶
* Cray UK, Bristol, BS1 4DJ, UK
† Rudolph Peierls Centre for Theoretical Physics, University of Oxford, Oxford, OX1 3NP, UK
‡ CCFE, Culham Science Centre, Abingdon, OX14 3DB, Oxon, UK
§ STFC, Daresbury Laboratory, Daresbury, WA4 4AD, UK
¶ STFC, Rutherford Appleton Laboratory, Harwell Campus, Didcot, OX11 0QX, UK

***Abstract*—GS2 [1] is a 5D initial value parallel code used to simulate low frequency electromagnetic turbulence in magnetically confined fusion plasmas. Feasible calculations routinely capture plasma turbulence at length scales close either to the electron or the ion Larmor radius. Self-consistently capturing the interaction between turbulence at ion scale and electron scale requires a huge increase in the scale of computation.**

We describe a new algorithm for computing FFTs in GS2 that reduces MPI communication using MPI 3 shared memory. With FFT data local to a node, the new algorithm extends perfect scaling to core-counts higher by almost a factor of 10 (if the load imbalance is small). For larger FFTs we propose and analyse the performance of a version of this algorithm that distributes the FFT data in shared memory over a group of nodes.

I. Introduction

GS2 is an initial value code which computes the time evolution of the five dimensional perturbed distribution function, g, by solving the gyrokinetic equation (GKE) for each species, and it is used to simulate low frequency electromagnetic turbulence in magnetically confined fusion plasmas. So far gyrokinetic simulations have routinely only captured plasma turbulence at length scales (perpendicular to the magnetic field) that are close either to the electron Larmor radius, or to the ion Larmor radius (which is approximately 60 times larger). The turbulence arising at these two length scales are likely to interact, and therefore a major development objective for GS2 is to model the turbulence at both scales simultaneously. This is considerably more demanding, by a factor of $\approx 60^3$, than independent calculations for the separated scales.

The GKE contains linear, non-linear and collision terms, and the non-linear term is computed more efficiently at each time step by exploiting fast Fourier transformations (FFTs). GS2 is parallelised using MPI, with the data array corresponding to g partitioned over the MPI ranks. The most computationally efficient data layout and domain decomposition of g for the various operators in the GKE differ substantially, and GS2 exploits these optimal decompositions at the cost of introducing a number of transpose operations between data layouts at each time step. In the current version of GS2 different data layouts are used for the computation of the FFTs, the collision operator, and the linear advance step. On top of this, when using a large number of MPI ranks more data communication is needed for a given computation, e.g. multidimensional FFT. The transforms between layouts involve all-to-all MPI communications which severely limit GS2's parallel scalability.

The FFT algorithm [2] is one of the most useful computational tools in scientific and engineering numerical codes, but in order to use its power in distributed-memory parallel computing models one has to overcome the poor scalability of parallel multidimensional FFT which results from the all-to-all communication pattern that it involves. Data traffic needed by FFTs can be reduced by designing layouts with optimally low communication requirements, and by using suitable programming models such as OpenMP or shared memory for modern HPC nodes with multicore processors.

While hybrid programming with MPI and OpenMP is widely used to extend parallelism and reduce the need for data movement or replication at the node level, in the last decade UK HPC community has started to use shared memory (SHM) and interprocess communication tools (provided by System V) for the same purpose in several MPI applications [3], [4]. MPI+SHM has the advantage over MPI+OpenMP that SHM does not need to be implemented homogeneously across the whole code in order to achieve the performance required using a fixed number of cores. At a first glance one can argue that shared memory data structures should be an efficient solution for better parallel scalability in codes that use static global data structures with regular data layouts and with a data access pattern that requires a low level of synchronisation. In general, when using SHM it is important to guard against race conditions and inconsistent views of memory which can occur in a shared address space. SHM is supported in MPI 3, which provides a consistent synchronisation mechanism for one sided remote memory access. It has been shown that this approach provides a significant performance increase for several standard parallel algorithms [5].

In this paper we present an algorithm that allows 2D FFT computations, presently computed in the memory of a single MPI task running on one core, to be computed in the shared memory either of one or of a group of compute nodes. In Section II we describe the single node algorithm and the code modifications made in GS2 to allow the FFT data to be stored in the shared memory associated with a single node. We assess its performance using the FFT benchmark and the scaling tests

provided in the GS2 distribution. In Section III we extend this algorithm further to allow the 2D data for each FFT to be distributed across shared memory in a group of nodes, and present preliminary results of its performance using a stand alone test code. The performance results reported here were obtained using the Cray XC30 system ARCHER, the UK HPC national facility [6]. This system has 2 Ivy Bridge processors with a total of 24 cores per node running at 2.7 GHz and 64 GB of memory. The nodes are linked with the Aries interconnect. Unless otherwise specified the timing presented here were obtained with codes compiled with the default version of Intel compiler.

II. Accelerating GS2 FFTs with shared memory

At each time step GS2 uses the FFTW library [7] to compute ≈ 100 2D FFTs in a subspace of the 7 dimensional distributed global data array. Owing to other algorithmic constraints, the 2D FFT plane is in the subspace spanned by the 3rd and 4th dimension indices of the data array, where (in the Fortran array convention used here) the first dimension index has the minimum stride in memory. For this data layout, the most efficient way to compute the FFTs is to ensure that every 4D sub-array spanning over dimensions 1 to 4 is contained within one MPI rank, so that all 2D FFTs can be computed locally. If, however, the total number of 4D subarrays is not a multiple of the number of MPI ranks, the domain decomposition in GS2 splits the 4D subarrays over MPI ranks and requires MPI communication in the computation of each FFT. This is generally less efficient than computing the 2D FFT directly using memory local to a single core, which is referred to, in GS2 parlance, as "accelerated FFTs". The downside of the layout where the FFT data is local to a core is twofold: reduced parallelism because the largest number of MPI ranks that can be used in the computation is the product of the last 3 dimensions of the global array; and the set of numbers of MPI ranks being restricted by the FFT data locality condition.

We propose here an alternative solution to enhance computation parallelism, while avoiding the expensive all-to-all MPI communications. Using SHM each 4D subarray is placed inside a shared memory segment on the compute nodes. In this way, the number of MPI ranks that can compute the FFT in parallel on every 4D sub-array can be increased by a factor equal to the number of cores per compute node, which is typically in the range 20–40 on current HPC systems. If the node has a non-uniform memory access architecture (NUMA) it may useful to use SHM segments on each NUMA domain in order to improve data locality at the cost of reducing scalability. In the following we define a shared memory node (SHM-node) as the subset of MPI ranks from a compute node (CN) that share data in a set of SHM segments. Shared memory FFTs were implemented in GS2 by making the following changes to the original code:

- writing API wrappers around MPI 3 subroutines which map the shared memory segments to GS2 datatypes;
- extending the GS2 data layouts by adding the facility to distribute the GS2 data as 4D subarrays over nodes instead of MPI ranks, with load imbalance where necessary (NB load balancing is perfect for runs where the number of 4D subarrays is a multiple of the number of compute nodes);
- changing three GS2 array variables to pointers to shared memory segments. (NB two of these arrays are internal to the GS2 FFT module);
- defining two new FFTW3 plans.

Now we describe a few more details of the implementation.

A good part of the implementation effort went into a Fortran module that interfaces GS2 with the MPI 3 shared memory subroutines. Inspired by the work done in Ref [3], a number of generalised utility subroutines are provided to help GS2 developers to manage the access to and synchronisation of shared memory segments (as well their allocation and pointer mapping) in the complex context of a "real world" application. Information about the node MPI communicator used to create the SHM segment is stored in a public derived data type, and detailed information on the allocated SHM segments is stored in a private linked list which can be accessed and modified using specialised subroutines.

Most modifications to the GS2 source code related to extending the data layouts descriptors and adding query subroutines to access the 4D sub-arrays allocated in SHM segments. In a given run the N_4 4D sub-arrays are distributed across the N_{SHM} SHM-nodes as follows. $\mathrm{Int}(\mathrm{N}_4/\mathrm{N}_{\mathrm{SHM}})$ 4D sub-arrays are distributed to each SHM-node, and where there is a finite remainder r, one additional sub-array is allocated to each of the SHM-nodes $0, \ldots r-1$: i.e. if $r > 0$ we introduce work imbalance in order to avoid MPI communication. In the current implementation of GS2 SHM can be turned on at the build time with a preprocessor flag, thus allowing the code to be built with earlier versions of the MPI library.

The performance of the MPI+SHM implementation was tested using the FFT benchmark and scaling test that are included in the GS2 distribution. The data array has 384 4D subarrays of dimension $53 \times 2 \times 128 \times 128$, which is a typical grid size used in current GS2 computations. With SHM the FFTs are computed for NUMA SHM-node and CN SHM-node versions. Fig 1 shows representative scaling results (a) for the FFT benchmark, and (b) for the scaling test run of GS2. Fig 1a shows that NUMA SHM-node version follows the baseline at low MPI rank counts and performs significantly better above the point at which baseline switches from accelerated FFTs to distributed FFTs (384 MPI ranks). The CN SHM-node version is less efficient than the baseline in the accelerated FFTs range, it becomes comparable to the baseline up to 3,000 MPI ranks and it is more efficient beyond this point. We note that the drop in CN SHM-node efficiency close to $7,000$ MPI ranks is due to a severe load imbalance where some nodes are allocated two 4D sub-arrays and other nodes one. The SHM efficiency fully recovers at the higher core count ($\approx$ 10,000 MPI ranks)) with exactly one sub-array per node, and is not significantly impaired where the imbalance is more modest, e.g. at 144 MPI ranks. NUMA SHM-node has a similar behaviour but with the number of compute nodes scaled be a factor of $1/2$.

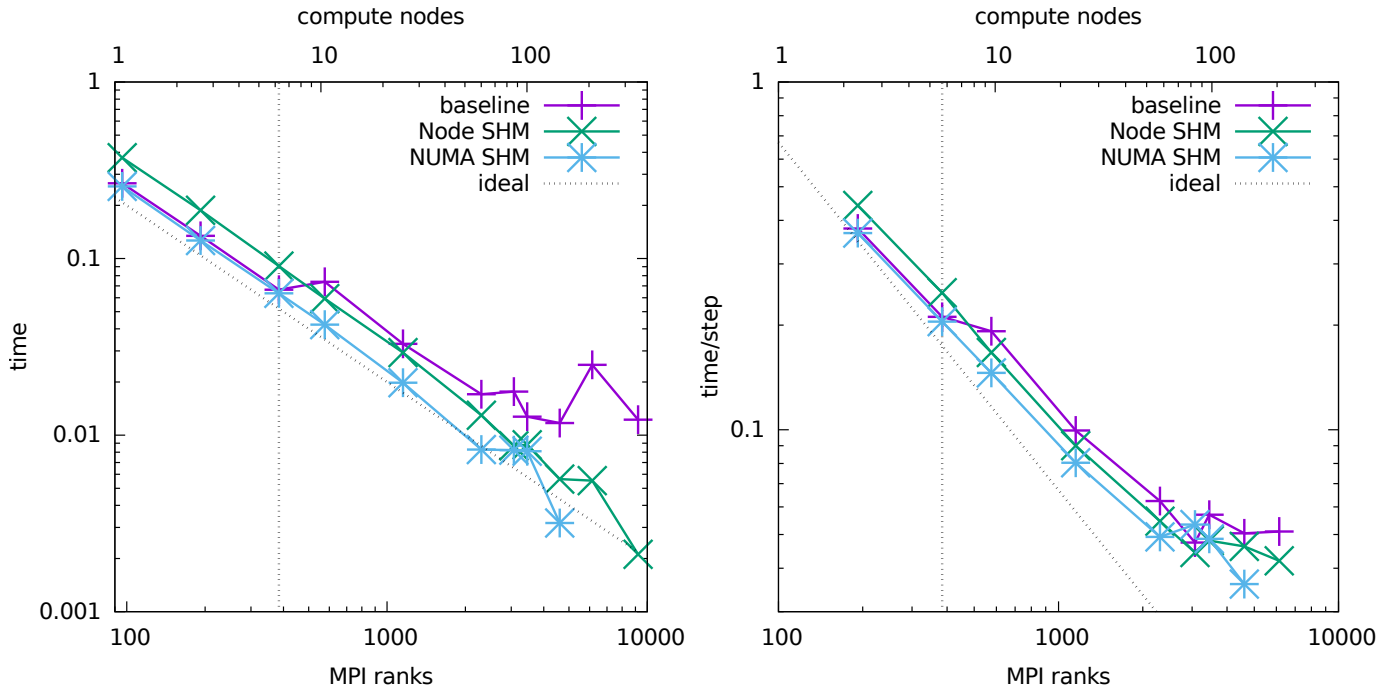

Fig. 1. Comparing the performance of the FFT+SHM algorithm vs baseline FFT algorithm, for the typical GS2 FFT benchmark (a) and the GS2 test run (b). The vertical line marks the point where baseline version switches from the accelerated FFTs to distributed FFTs. Time is in seconds.

In Fig 1b we show the average time (over 100 steps) spent to compute a simulation step versus the number of MPI ranks for a collisionless gyrokinetic test case (i.e. solving the gyrokinetic equation using GS2 in the limit where the collision term does not need to be computed). The timings are for the two SHM versions discussed before (SHM-nodes sharing one NUMA domain or the whole compute node) and the baseline. The behaviour at low core counts is similar to the FFT benchmark, at higher core counts the scaling saturates for all versions with different plateau values. The NUMA SHM-node case speeds up this computation by around 25% within the range $\approx 384 \ldots 2000$ and higher that 4,000 MPI ranks. The compute node SHM-node case reaches similar performance above 4,000 MPI ranks. We have noticed that around 3,000 MPI ranks all version have close timings, this region needs further investigations.

In summary, as the scaling of GS2 typical computations saturates long before the average number of 4D sub-array per compute node is close to one, our performance data suggest that it is more efficient to use NUMA restricted SHM-nodes for a wide range on MPI ranks ($\approx 400 \ldots 4000$).

III. Distributed sub-blocks

In this section we investigate an extension of the SHM FFT algorithm which would allow us to access yet higher core counts by distributing the 4D sub-arrays across nodes. We believe that explorations in this direction may facilitate GS2 simulations that consistently capture turbulence spanning electron and ion scales, which require spatial grids in the 2D FFT plane that are at least 100 times larger than those used currently, with the typical 2D FFT grid size increasing to $O(1000) \times O(1000)$. This is a first step in seeking optimal layouts for locally distributed arrays that reduce the expense of and need for global transposes to compute each of the operators in the GKE.

For simplicity we aggregate the first two dimensions of the GS2 4D sub-array into a single dimension of length N_t, so that the distributed arrays can be considered as 3D with global sizes N_t, N_x, N_y. The 2D FFTs are to be computed for all 2D layers spanning the second and third dimension (x,y). This 3D domain is partitioned into chunks in y that are distributed equally over N_{nodes} nodes. Inside the node, the arrays holding the domain data are allocated to SHM segments, and are therefore accessible to all nodes' ranks. Also we investigate the variation which uses NUMA SHM-nodes. This should improve the memory access speed in exchange for MPI communication inside the compute node.

For the 1D domain partition we use, the FFT computation can be done with the distributed version of the FFTW library. The main steps of the algorithm are as follows:

1) inside each SHM segment the (x,y) planes are divided uniformly among MPI ranks,
2) each MPI rank copies its assigned data from the SHM segment into a local buffer of size required the by distributed FFTW,
3) the FFT is executed with one call to FFTW library using

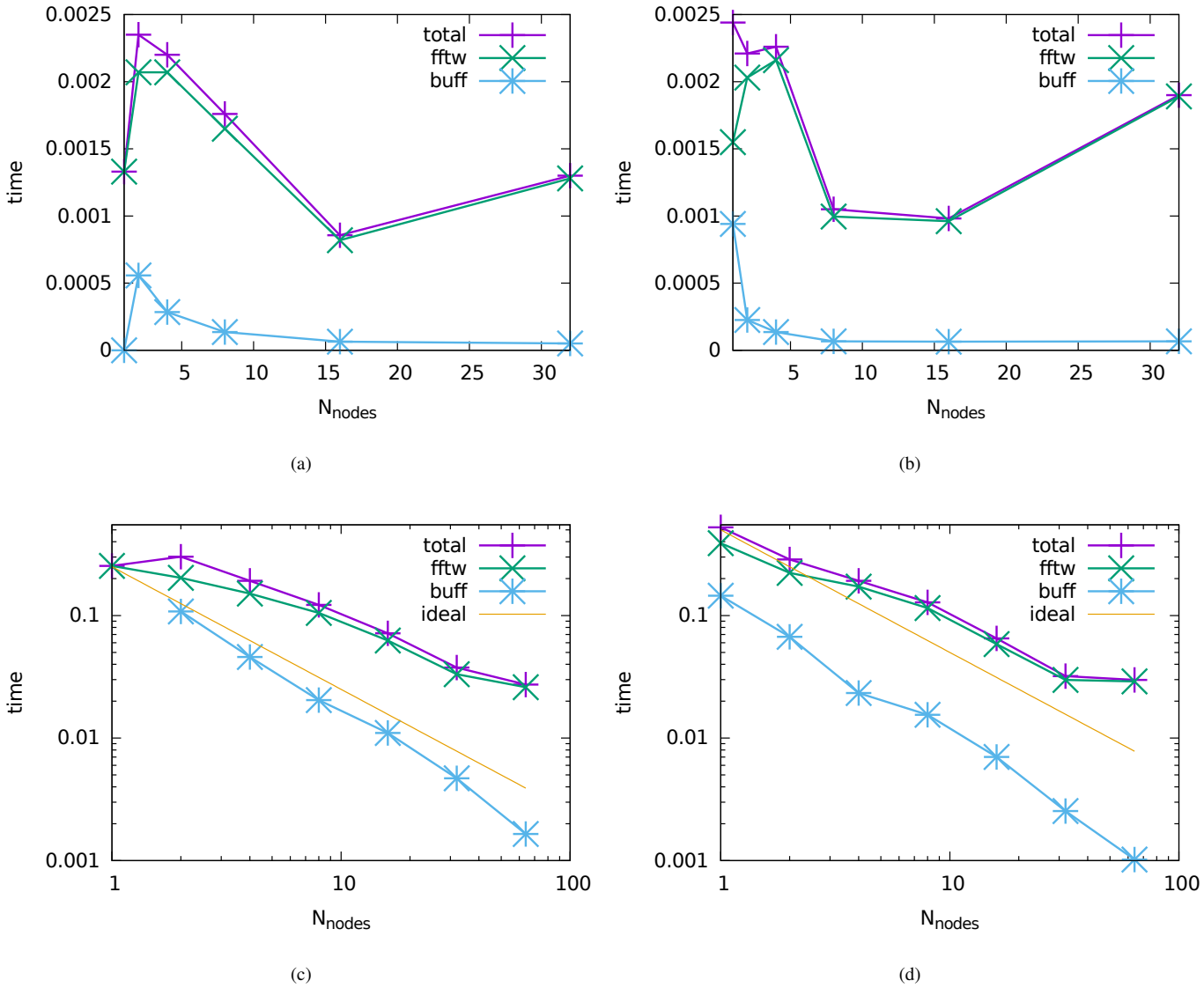

Fig. 2. 2D FFTs average time in seconds vs the number of compute nodes when using one SHM segments per compute node (a,c) or per NUMA domain (b,d). The top row, (a,b), is for a computation with current grid sizes, $96 \times 100 \times 128$, the bottom row, (c,d), is for a grid sizes projected to be used by a multiscale simulation, $96 \times 1000 \times 1024$. The average was done over 40 runs. A compute node has 24 MPI ranks.

a 2D distributed plan defined among the corresponding MPI ranks across the SHM nodes,

4) the FFT result data is copied back to the SHM segment.

We have chosen to transfers the data into a local buffer on each MPI rank because the distributed FFT may require more local memory than the size of the local array and also this allows to use FFTW_alloc that guarantees data alignment for the optimal FFT computational speed. We point here to a marginal advantage of MPI+SHM over MPI+OpenMP. The above algorithm can be implemented with OpenMP threads, but this would require a thread-safe version of the MPI library (which typically has lower performance) because the MPI communication would be taking place inside an OpenMP region.

The performance of the distributed SHM algorithm has been measured for a typical current grid size, and for a grid that is $\sim$100 times larger in the FFT plane, i.e.: $96 \times 100 \times 128$ and $96 \times 1000 \times 1024$. A small number of tests were carried out on grids with slightly different sizes in x to check that the scaling results are not sensitive to this particular grid choice. For simplicity the y size was kept a power of 2 in order to have an equal partition over the compute nodes. The test code was instrumented to collect separated timings for calls to FFTW and for buffer to sub-array data transfers.

Fig 2 shows the complex FFTs computing time (averaged over 40 runs on ARCHER) versus the number of nodes for the small (a,b) and large (c,d) grids and with SHM segments allocated either across the whole node (a,c) or restricted to a

NUMA domain (b,d). The test code was compiled with Cray compiler 8.4 with the default optimisation level. Runs with an Intel compiled version showed differences up to 10%, but we do not present a detailed compiler comparison here.

Figs 2a, 2b shows that splitting the smaller sized block over multiple nodes brings no speed up benefit. Fig 2c shows that the time to execute the large block FFT on one node is ≈ 190 times longer than for the smaller size block, and that splitting the larger sub-arrays across nodes increases the computation speed but with low efficiency. The large block FFT computation can be accelerated ≈ 10 times by using 64 times more computational power. The distribution over two nodes is particularly slow, mainly because of the large buffering time, but above 4 nodes the scaling improves and is controlled by the scaling of FFTW. Restricting SHM to span NUMA regions is detrimental for a single compute node and it has a small impact at higher node counts, see Fig 2d. The buffer transfer time is negligible except for the 2 compute nodes case. When the SHM segment is constrained to one NUMA domain the buffer transfer time decreases at the cost of slowing down FFTW, as one would intuitively expected. Although the scaling efficiency of this algorithm is modest it could be valuable if the locally distributed sub-arrays can be used to reduce the number of transposition used by GS2 current algorithm.

IV. Conclusions

In this work we have described an algorithm and an implementation of MPI 3+SHM in GS2 for the computation of FFTs. We have shown that shared memory allows perfect scalability for FFTs in GS2 to be extended towards 10,000 MPI ranks on current grid sizes with a small change in the current data layout. The collisionless GS2 test show an improvement in performance of approximately 25% when using SHM restricted to NUMA domain for the FFTs at moderate and high MPI rank counts ($400 <$ # MPI ranks < 6000). Using groups of shared memory nodes we have shown that, for the grid sizes suitable for multiscale plasma simulation, FFT scaling can be extended by approximately one order of magnitude but with a low efficiency ($\approx 10/64$). This is a first step in exploring new GS2 data layouts that eliminates the need for all-to-all communication patterns that limit the scalability of the current version of the code.

Acknowledgements

This work was partially supported by the Plasma HEC Consortium [EPSRC grant number EP/L000237/1], CCP Plasma [EPSRC grant number EP/M022463/1], the UK Engineering and Physical Sciences Research Council (EPSRC) through the Software Outlook Programme, and the RCUK Energy Programme [grant number EP/I501045].

This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] GS2 website. [Online]. Available: https://sourceforge.net/projects/gyrokinetics/

[2] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Mathematics of Computation, vol. 19, no. 90, pp. 297–301, 1965, doi: 10.1090/S0025-5718-1965-0178586-1.

[3] I. J. Bush, "New Fortran Features: The Portable Use of Shared Memory Segments," HPCx Consortium, Tech. Rep., 2007. [Online]. Available: http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0701.pdf

[4] L. Anton, "Improving the parallelisation and adding functionality to the quantum Monte Carlo code CASINO," HECToR: UK National Supercomputing Service, Tech. Rep., 2009. [Online]. Available: http://www.hector.ac.uk/cse/distributedcse/reports/casino/casino_final_report.pdf

[5] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory," Journal of Computing, May 2013, doi: 10.1007/s00607-013-0324-2.

[6] Archer website. [Online]. Available: http:www.archer.ac.uk

[7] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE, vol. 93, no. 2, pp. 216–231, Feb 2005, doi: 10.1109/JPROC.2004.840301.