# Hybrid parallelisation for the CRYSTAL code

Barry G.Searle

February 15, 2019

**Abstract**

The two-electron and exchange-correlation integral routines in CRYSTAL have been modified to reduce the use of global variables and make them thread-safe. OpenMP tasking directives have been added to these routines to produce a hybrid parallel version of the code, so potentially improving the performance and reducing the memory footprint of the code. The modified version of CRYSTAL can now run a 3360 atom $Al_2O_3$ grain boundary problem using all the cores on an ARCHER node with 45% speed-up.

## 1 Introduction

The aim of the project was to improve the performance and reduce the replicated memory requirements of CRYSTAL by the addition of OpenMP directives. This should allow users to make use of unused cores on the Archer service when the memory requirements of a job have resulted in the number of MPI processes on a node being underpopulated.

The algorithm used in CRYSTAL to compute the ground state energy can be summarised as follows

1. From a given density matrix calculate analytically the Hamiltonian matrix elements in the Atomic Orbital (AO) basis

   *a*. Calculate the two-electron, Coulomb and exchange, terms

   *b*. Calculate the one-electron terms

   *c*. Add the DFT exchange-correlation components obtained from quadrature. (Not needed in Hartree-Fock calculations)

   and generate the total energy.

2. Fourier transform the Hamiltonian to its reciprocal-space representation.

3. Compute and solve the transformed Hamiltonian by direct diagonalisation at each k-point.

4. From the eigenvalues calculate the Fermi energy.

5. From the Fermi energy and the eigenvectors calculate the new density matrix.

6. Repeat steps 1.-5. until convergence in the total energy is achieved. Each repeat is an iteration of the Self-Consistent Field (SCF) method.

Due to the time constraints of the eCSE the work plan involved two phases to address steps 1a and 1c. Step 3 uses library routines, LAPACK in the serial/data replicated code and ScaLAPACK in the distributed data code. As modern implementations of these libraries on ARCHER provide support for threaded versions the diagonalisation step should show some performance improvement without modification. Steps 1b, 2, 4 and 5 were not addressed in this project, which will limit the maximum speed-up achievable for an SCF iteration. Step 1b is usually the most computationally expensive of the unmodified steps. For geometry optimisation jobs there is also no change to the calculation of forces that follows the SCF iterations.

Each phase of the project required modernising the code to move variables from global storage, common blocks or modules, into local variables. e.g.

```
SUBROUTINE DFAC4
USE NUMBERS
IMPLICIT REAL(FLOAT) (A-H,O-Z)
COMMON/TCOMM/ACCFAC
```

becomes

```
SUBROUTINE DFAC4(ACCFAC)
USE NUMBERS
IMPLICIT NONE
REAL(FLOAT), INTENT(IN) ::  ACCFAC
```

In cases where there were multiple calls to a subroutine this had the added advantage of simplifying the code. e.g.

```
OLDACC = ACCFAC
ACCFAC = ACCFAJ
CALL DFAC4
ACCFAC = ACCFAK
CALL DFAC4
ACCFAC = OLDACC
```

becomes

```
CALL DFAC4(ACCFAJ)
CALL DFAC4(ACCFAK)
```

In some cases related variables were grouped into derived types to reduce the length of the subroutine parameter list and reduce the risk of mistakes in its use. Once these global variables were made thread safe, OpenMP directives could then be added to run tasks on any unused cores. This also required finding test cases that covered the routines being modified to make sure the changes had been tested and were correct.

It is worth also mentioning that the integrals, step 1 in the above scheme, are the most memory consuming part of most large scale runs of CRYSTAL. This is due to many of the required data structures being fully replicated on each process to simplify the code due to the irregular nature of the data accesses required in these subroutines. Thus the memory use of this portion does not decrease with increasing number of processes, and indeed the memory use per node increases as a larger number of processes are placed upon it. Use of threads reduces the number of processes required to populate the node while still allowing full use of all the cores, and thus should allow large calculations to use larger number of cores *on a node*. One can also contrast this with the other portions of the code which are fully data distributed. Thus large structures are shared among the nodes, and in practice the memory usage in these sections is less important.

## 2 2-electron integral OpenMP modifications

The driver routine for the 2-electron integrals is `SHELLXN`. The 2-electron integrals require 4 Gaussian basis functions, an initial and final state for each of the electrons. The outer loop of `SHELLXN` runs over a screened pair of basis functions. Inside this is a loop over translated copies of one of the pair of functions. OpenMP tasks are generated from the code inside these loops which contains loops over than remaining pair of basis functions. The resulting tasks are fairly coarse-grained, but have the advantage of generating independent writes to the outputted data structure, the irreducible Fock matrix, thus avoiding the need for atomic memory access directives. This requires changes to pass information via argument lists to all subroutines, and their children, called by `SHELLXN`. To simplify the parameter passing some of the data is collected into new derived types which are declared locally. These types produce a clearer relationship between some of the data and the integrals calculation. The main part of `SHELLXN`, which is essentially the OpenMP task, is split off into a subroutine which allows most of the local variables to be declared in the task rather than be declared in `SHELLXN` and then copied by the OpenMP directive.

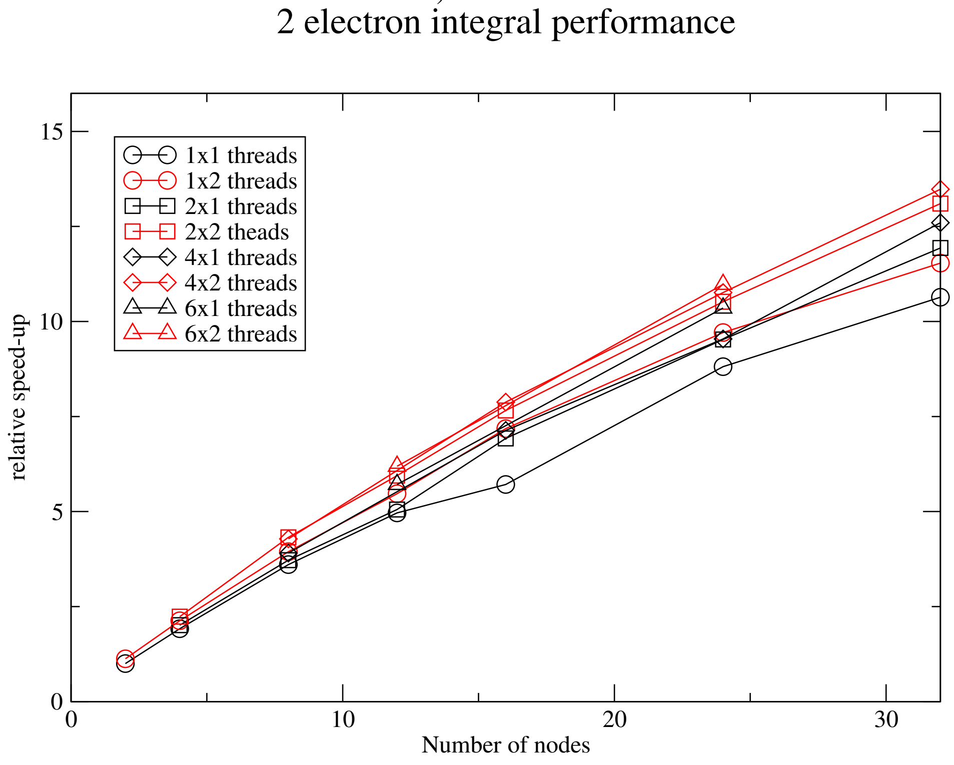
The main complication in these routines are the buffers used for the bipolar approximation to the integrals. Each thread needs it own table, which need to be recalculated when full rather than spilling to disk. If the memory buffer is large enough for the fulltable then all threads can share it and local buffers aren't unnecessary. These dynamically allocated buffers may be the largest temporary data objects depending on their size which can be specified in the input file. The next largest, which are also dyanmically allocated, are the work space for the `VIC5` subroutines. The smaller objects are declared locally and in a few cases are eliminated with the code restructuring.

Similar changes have been applied to the P1 coulomb routine `SHELLXCOUL`. This is a specialised version of `SHELLXN` which is used in very low symmetry systems, as otherwise trying to exploit the (non-existant) symmetry in calculation of the integrals results in significant overheads. This routine has the same structure as `SHELLXN` without the exchange terms, which instead are calculated in `SHELLEXCH`. In this case the order of updates to the irreducible Fock matrix in `SHELLEXCH` may conflict, requiring the use of atomic memory accesses.

### 2.1 Performance

The performance of the modified routines on a $TiO_2$ $4 \times 4 \times 4$ supercell is shown in figure 1 (the data is in table 1 in the appendix). The performance of the code is generally slightly better on the same number of cores with threads, due to the better dynamic load balancing of the OpenMP tasks compared to the pure MPI code. There is also approximately 10% extra performance available when using the virtual hyper-threads available on the Xeon processors. Using hyper-threads is even more important on the Xeon Phi nodes of the Archer test system (see the appendix).

Figure 1: SHELLXN performance for equivalent number of MPI processes $\times$ threads for a $TiO_2$ $4 \times 4 \times 4$ supercell, e.g. 4 nodes is 96x1 or 48x2. Speed-up is relative to 48x1. (In the legend, $n \times m$, is $n$ cores each with $m$ hyper-threads, so 2x2 is 2 cores with 2 threads for a total of 4 threads)



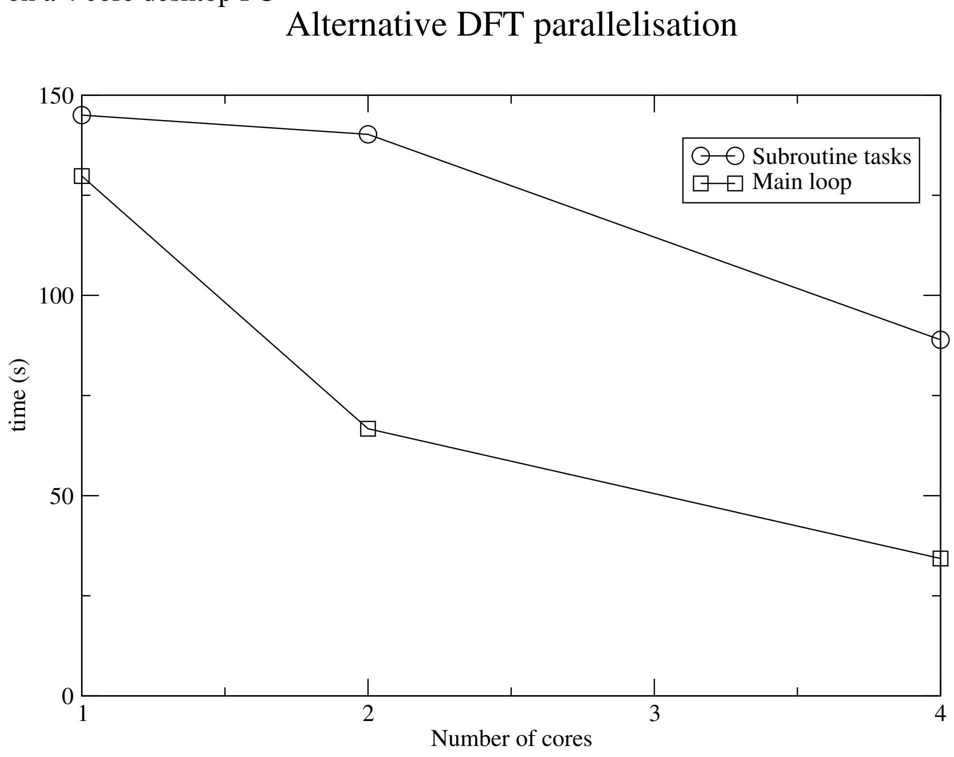
The modified routines are about 1% slower than the unmodified code due to the overhead of copying multiple subroutine parameters into 3 or 4 deep subroutine call sequences. This is easily negated by the extra performance available from the hyper-threads.

## 3 DFT OpenMP modifications

The DFT routines are more modern than the coulomb integrals and use modules rather than common blocks. The subroutine calls are also shallower making it easier to trace the effects of the global variables since a variable isn't used in the driver routine and then 3 levels down requiring it to be passed through the intervening routines.

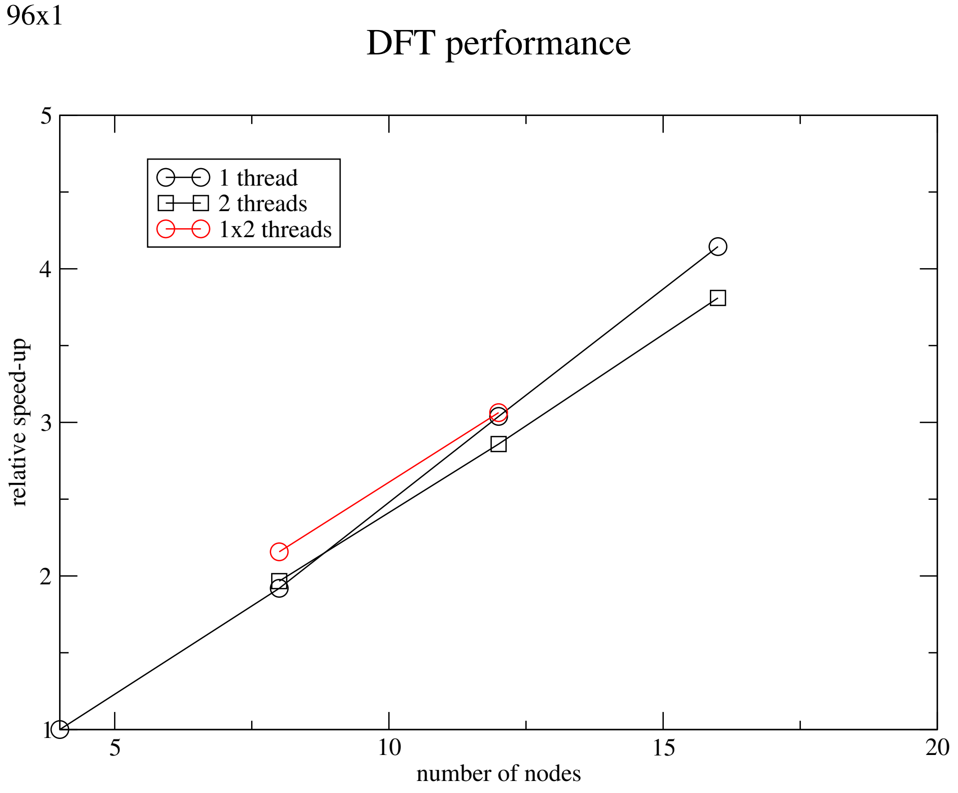
The driver routine contains a main loop over batches of DFT grid points over which the quadrature is performed. This is somewhat similar to the main loop over shell pairs in the integrals and is one possible candidate for generating parallel tasks. The loop over DFT batches of points involves 3 steps. These are the generation of the density on the grid, the evaluation of the functional from this density, and the mapping of the functional into the Fock matrix. Approximately 70% of the time in the loop is in the first step, with 25% in the third step. A second parallelisation strategy considered was to use the loops within these 2 subroutines to generate tasks. Both parallelisation options can have threads simultaneously attempting to write the same location in the Fock matrix in the final step and therefore require the use of OpenMP atomic memory directives. The second option can also have conflicting memory accesses to the density on the grid requiring additional atomic operations. The second option also requires task waits within each subroutine since the functional can't be evaluated until the density is complete. The result is that the first, more coarse grained, parallelisation option has better scaling than the second (see figure 2).

Figure 2: Performance of alternative DFT parallelisation options for an alumina slab on a 4 core desktop PC



The other difference between the two parallelisation options is the memory requirements. There are temporary arrays required by each thread sized by the number of points times the maximum size of an orbital pair (`NPMAXT`) and others related to the number of points times the direct space vectors that contribute to a grid point (`NPICAVER`). The default number of grid points, changed by the `CHUNKS`) keyword is 200. The second parallelisation only requires temporary storage per thread proportional to `NPMAXT`. The first option has temporary storage related to both values and `NPICAVER` is the larger of the two values. The largest value of `NPICAVER` found from examining a variety of test case is 500000. This is about the same size as the temporary storage already used in the integral routines, so we have chosen to use the first parallelisation option due to its better performance. Due to the size of `NPICAVER`, temporary objects of this size are dynamically allocated for each thread, whilst the smaller objects are declared locally.

Figure 3: NUMDFT performance for equivalent number of MPI processes × threads for a $TiO_2$ $6 \times 6 \times 6$ supercell, e.g. 4 nodes is 96x1 or 48x2. Speed-up is relative to 96x1



### 3.1 Performance

The performance of the modified routines on a $TiO_2$ $6 \times 6 \times 6$ supercell is shown in figure 3 (the data is in table 3 in the appendix). The times for the smaller supercell used in the integrals were too small to be reliable (0.5-2 seconds) and are included in the appendix.

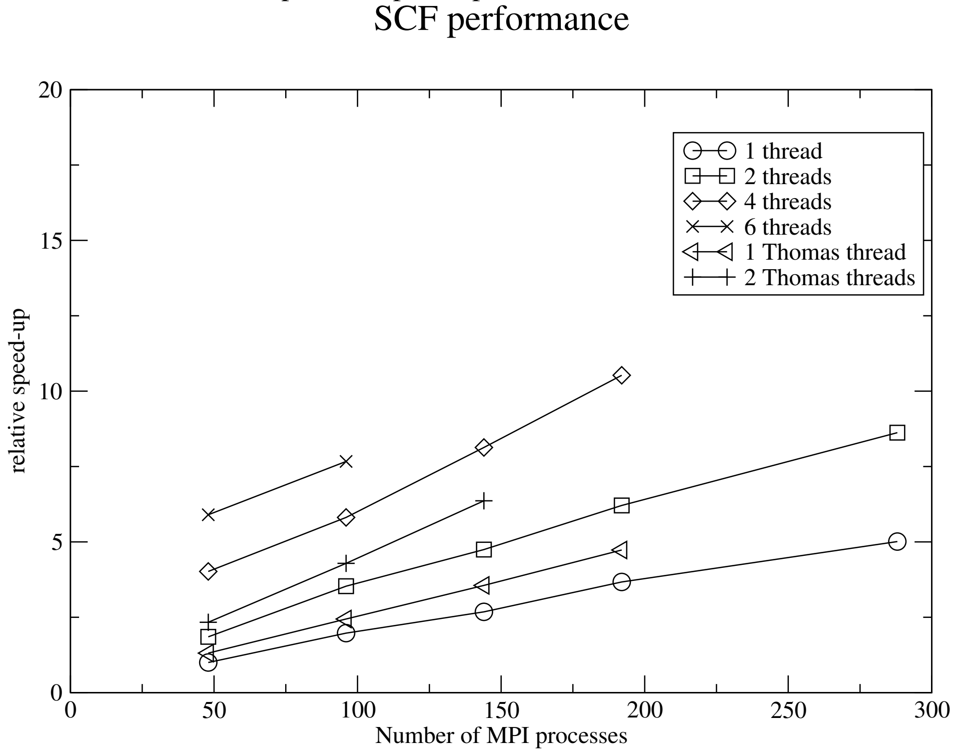
## 4 Overall Performance

The overall performance of an SCF iteration is presented in figures 4 and 5. The Alumina example case is a true test of what we are trying to achieve as it requires 4GB per MPI process and is unable to use all the cores on a node. In this case there is a speed up of approximately 45% relative to the unthreaded code by using the unused cores on a node.

Since not all parts of the code have had OpenMP directives added in this project the overall SCF performance of the threaded code is not expected to be twice that of the unthreaded code with 2 threads. There are also some benefits due to the greater memory bandwidth per core and possibly higher turbo mode clock rates on an under-populated node that contribute to this.

There may also be some performance improvement with hyper-threads on Archer. This needs to be tested for each problem being run since although the integrals perform

Figure 4: Performance of 1 SCF iteration for an equivalent number of MPI processes for a $TiO_2$ $4 \times 4 \times 4$ supercell. Speed-up is relative to 48x1



better, this can be partly cancelled out by reduced performance in the ScaLAPACK calls. One possible solution may be to use Intel's MKL library, where the threading within it can be separately controlled by the `MKL_NUM_THREADS` environment variable, instead of Cray's libsci.

## 5 Possible extensions

The simplest extension to this project is to add OpenMP directives to the one-electron integral calculation (step 1b in the SCF description). These share routines with the two-electron integrals so some of the required code modernisation work has already been done.

Other possibilities are to add threading directives to the CPHF two-electron integral and DFT driver routines or to modernise and thread the calculation of forces for geometry optimisations.

## 6 Summary

We have modernised the two-electron integral and numerical DFT code in CRYSTAL17. OpenMP directives have been added to the modified routines to support hydrid OpenMP/MPI parallelism in the program. The demonstrated speed-up is approximately 45% for an Alumina test case that must be run under-populated on Archer

Figure 5: Performance of 1 SCF iteration for an equivalent number of MPI processes for a model of an Alumina grain boundary. A node with 1 thread is under-populated with MPI processes, e.g. 96x1 requires 8 nodes (192 cores). Speed-up is relative to 96x1

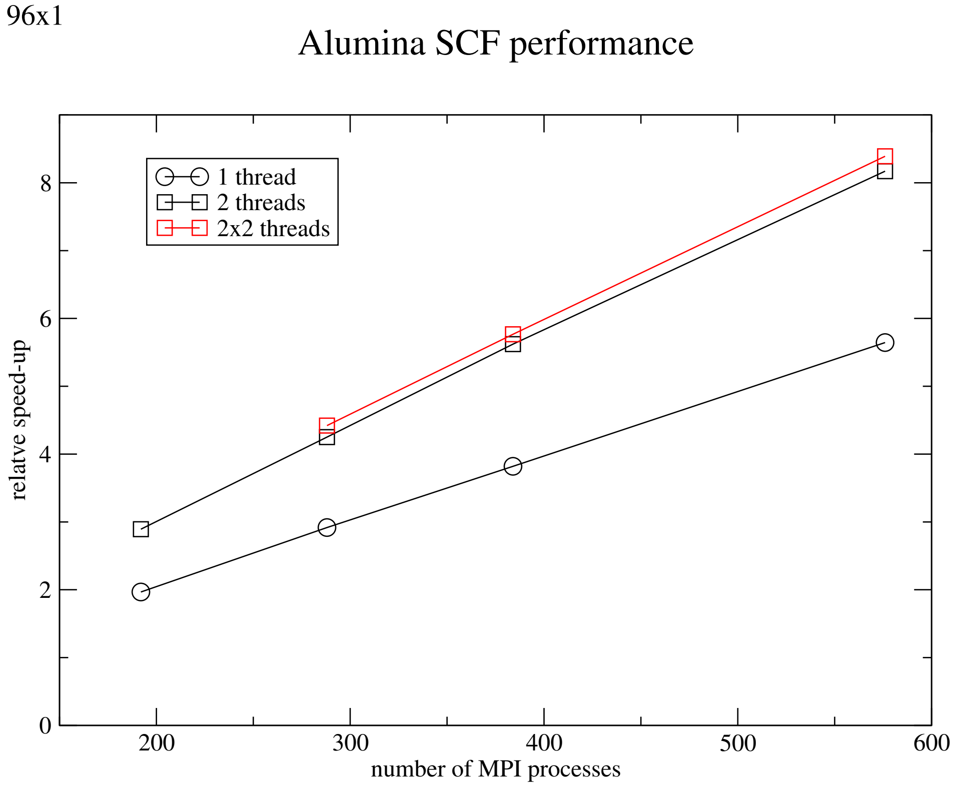


Table 1: SHELLXN performance on Archer Xeon nodes for a $TiO_2$ $4 \times 4 \times 4$ supercell

| Number of nodes | 1 thread (s) | 2 threads | 4 threads | 6 threads | 12 threads |
|---|---|---|---|---|---|
| 2 | 65.64 | - | - | - | - |
| 4 | 34.21 | 32.64 | - | - | - |
| 8 | 18.21 | 17.64 | 16.79 | - | - |
| 12 | 13.22 | 12.99 | - | 11.49 | - |
| 16 | 10.21 | 9.48 | 9.20 | - | - |
| 24 | 7.45 | 6.89 | 6.88 | 6.34 | 6.23 |
| 32 | 6.17 | 5.50 | 5.21 | - | - |

Table 2: NUMDFT performance on Archer Xeon nodes for for a $TiO_2$ $4 \times 4 \times 4$ supercell

| Number of nodes | 1 thread (s) | 2 threads | 4 threads | 6 threads | 12 threads |
|---|---|---|---|---|---|
| 2 | 6.9 | - | - | - | - |
| 4 | 3.42 | 3.65 | - | - | - |
| 8 | 2.07 | 2.31 | 2.17 | - | - |
| 12 | 1.59 | 1.62 | - | 1.79 | - |
| 16 | 1.37 | 1.5 | 1.53 | - | - |
| 24 | 1.33 | 1.37 | 1.32 | 1.38 | 1.37 |

nodes with the original code. This will allow users to make better use of their Archer resources for jobs that are currently limited by the memory requirements per process.

## 7 Acknowledgement

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

## 8 Appendix - Test details

Archer has 24 core 2.7GHz Xeon v2 nodes with 64GB, the Tier-2 system Thomas has 24 core 2.2GHz Xeon v4 nodes with 128GB and the Xeon Phi system has 64 1.3GHz cores with 96GB.

The $TiO_2$ $4 \times 4 \times 4$ supercell has 384 atoms, 8 kpoints and 8064 orbitals. The $TiO_2$ $6 \times 6 \times 6$ supercell has 1296 atoms, 1 k point and 27216 orbitals. The Alumina grain boundary has 3360 atoms, 1 k point and 43680 orbitals.

Archer tests were run with the PrgEnv-intel/5.2.82, cray-libsci/16.11.1, and Intel/17.0.0.098 modules. The Alumina tests were run with `aprun -n 192 -N 12 -S 6 -cc numa_node` (1 thread), `aprun -n 192 -N 12 -S 6 -d 2 -cc numa_node` (threads), and `aprun -n 192 -N 12 -S 6 -d 4 -j 2 -cc numa_node` (2x2 hyper-threads).

Figure 6: NUMDFT performance for equivalent number of MPI processes $\times$ threads for a $TiO_2$ $4 \times 4 \times 4$ supercell, e.g. 4 nodes is 96x1 or 48x2. Speed-up is relative to 48x1

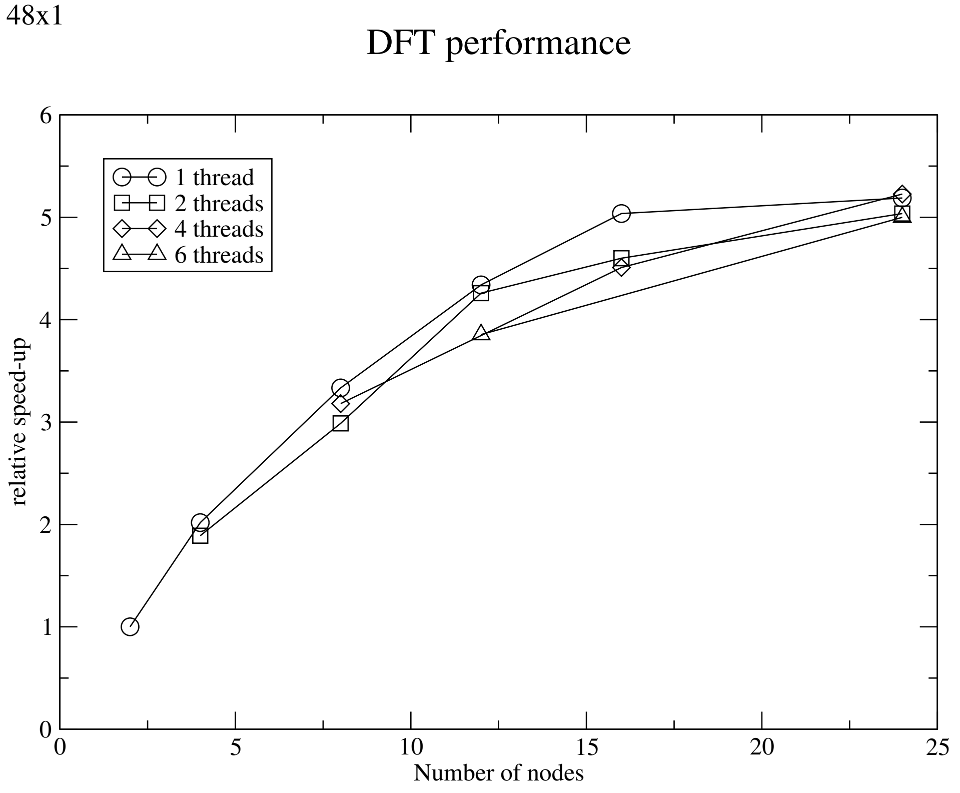


Table 3: NUMDFT performance on Archer Xeon nodes for for a $TiO_2$ $6 \times 6 \times 6$ supercell

| Number of nodes | 1 thread (s) | 2 threads | 4 threads | 6 threads | 12 threads |
|---|---|---|---|---|---|
| 2 | - | - | - | - | - |
| 4 | 22.67 | - | - | - | - |
| 8 | 11.81 | 11.53 | - | - | - |
| 12 | 5.47 | 5.95 | - | - | - |

Table 4: Performance of 1 SCF iteration on Archer Xeon nodes for a $TiO_2$ $4 \times 4 \times 4$ supercell

| Number of MPI processes | 1 thread (s) | 2 threads | 4 threads | 6 threads |
|---|---|---|---|---|
| 48 | 385.75 | 208.33 | 118.86 | 95.96 |
| 96 | 195.48 | 109.22 | 66.38 | 50.3 |
| 144 | 143.94 | 81.28 | 47.44 | 39.5 |
| 192 | 105.04 | 62.14 | 36.65 | - |
| 288 | 76.98 | 44.74 | - | - |

Table 5: Performance of 1 SCF iteration on Thomas Xeon nodes for a $TiO_2$ $4 \times 4 \times 4$ supercell

| Number of MPI processes | 1 thread (s) | 2 threads (s) |
|---|---|---|
| 48 | 294.32 | 165.21 |
| 96 | 157.95 | 89.26 |
| 144 | 108.45 | 60.63 |
| 192 | 81.55 | - |

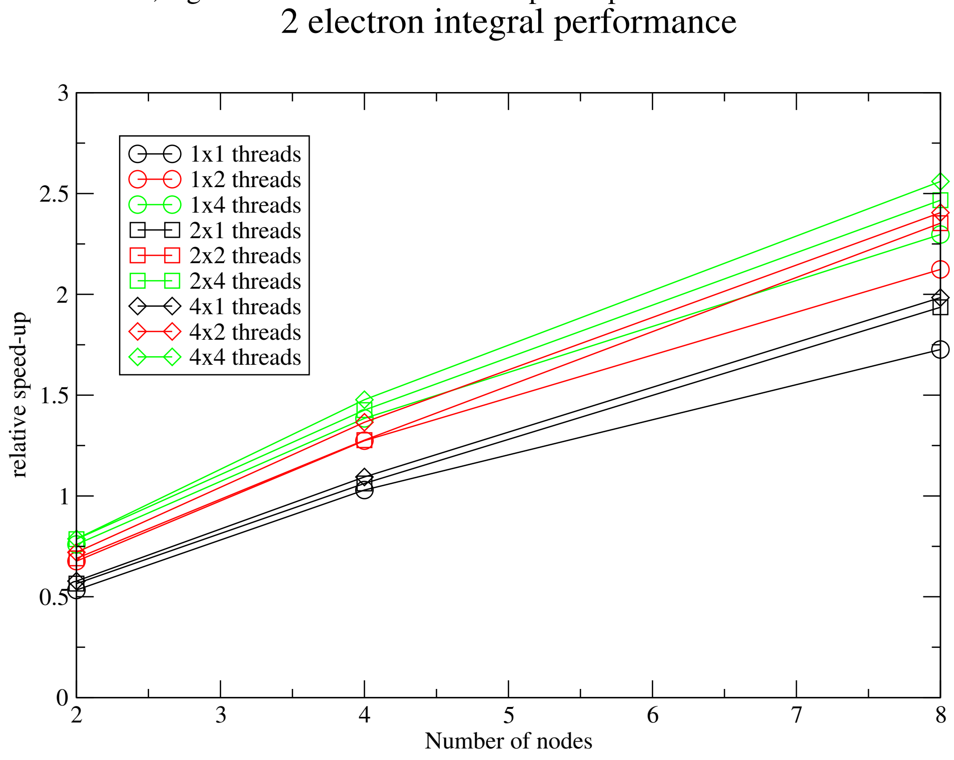
Table 6: Performance of 1 SCF iteration for an equivalent number of MPI processes for a model of an Alumina grain boundary. A node with 1 thread is under-populated with MPI processes, e.g. 96x1 requires 8 nodes (192 cores).

| Number of MPI processes | 1 thread (s) | 2 threads | 2x2 threads |
|---|---|---|---|
| 96 | 5767.8 | - | - |
| 192 | 2932.1 | 1993.9 | - |
| 288 | 1977.4 | 1357.1 | 1304.7 |
| 384 | 1509.9 | 1026.1 | 1000.2 |
| 576 | 1021.9 | 705.9 | 687.4 |

Table 7: SHELLXN performance on Xeon phi nodes for $TiO_2$ $4 \times 4 \times 4$

| Number of nodes | 1 thread (s) | 2 threads | 4 threads | 1x2 | 2x2 | 4x2 | 1x4 | 2x4 | 4x4 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 122.99 | 115.92 | 113.58 | 97.0 | 95.08 | 91.1 | 86.43 | 83.55 | 83.37 |
| 4 | 63.77 | 61.82 | 60.0 | 51.51 | 51.42 | 48.07 | 47.4 | 46.04 | 44.43 |
| 8 | 38.02 | 33.91 | 33.09 | 30.91 | 27.9 | 27.29 | 28.58 | 26.61 | 25.64 |

Figure 7: SHELLXN performance for equivalent number of MPI processes × threads on Xeon Phi, e.g. 4 nodes is 96x1 or 48x2. Speed-up is relative to 48x1 Archer Xeon


2 electron integral performance
relative speed-up
Number of nodes
1x1 threads
1x2 threads
1x4 threads
2x1 threads
2x2 threads
2x4 threads
4x1 threads
4x2 threads
4x4 threads