Hybrid OpenMP and MPI within the CASTEP code

E.J. Higgins[1], M.I.J. Probert[1], P.J. Hasnip[1], K. Refson[2], and I.J. Bush[3]

[1]Dept. of Physics, University of York
[2]Dept. of Physics, Royal Holloway, University of London
[3]Oxford e-Research Centre, University of Oxford

September 15, 2015

Abstract

This eCSE aimed to reduce the memory usage of CASTEP by using a hybrid OpenMP+MPI approach, and to reduce calculation time on large systems by implementing a parallel matrix diagonaliser. By implementing OpenMP and with selected use of threaded libraries, the memory per node on ARCHER can be reduced to less than 10% without a significant penalty in calculation time. Using a parallel diagonaliser, the wall time of large calculations has also improved dramatically.

Introduction

CASTEP[1, 2] is an *ab initio* density functional theory (DFT) code developed in the UK, capable of modelling systems of up to a few thousand atoms, and can scale well up to a few thousand cores on ARCHER. It has consistently been in the top 10 codes by core hours across both HECToR and ARCHER[3], using over 5% of the machine over the course of a year[4].

CASTEP describes the electronic states (”bands”) of a material, using a plane-wave (”g-vector”) basis at different points (”k-points”) in reciprocal space. CASTEP can parallelise the work over these 3 dimensions, so that each k point has its own set of bands, and each band has its own set of g-vectors. However, band parallelism is not widely used, and for large calculations there are often only a small number of k-points.

In order to accommodate different sizes of calculations on a range of machines, CASTEP allows for the user to switch between fast, high memory algorithms and slower algorithms that use less memory. However for very large systems, it has sometimes been previously necessary to under-populate the nodes to get sufficient RAM per core. This is because there are certain arrays, e.g. the bands $\times$ bands subspace Hamiltonian, which are allocated on every MPI process.

Whilst a large part of a CASTEP calculation is parallelised some of the work, most notably diagonalisation of large matrices, is performed in serial on all MPI processes. In large (1000+ atom) calculations, this work dominates the runtime, as it scales cubically with the system size. In this project, we proposed the following work packages to address these shortcomings:

Work Package 1: Reduce memory per node The first aim of this project was to reduce the memory usage by sharing the memory within a node using OpenMP. This allows nodes to be fully utilised by using a combination of OpenMP threads and MPI processes. A side effect of this is that use of threaded libraries might now be beneficial. This was also investigated.

Work Package 2: Parallel matrix diagonaliser The second aim of this project was to implement a parallel matrix diagonaliser to distribute the matrix diagonalisation work over multiple nodes using MPI.

Both work packages have been successfully completed, and the resulting code has been merged into the current development version of CASTEP 9.0, for worldwide release later in 2015. Thus, these developments will be available to all ARCHER users once the system modules have been updated.

The benchmarks This report focusses on 2 benchmarks:

- Crambin - A small seed storage protein, containing H, C, N, O and S, (1284 atoms).
- Si_8 7x7x7 conventional supercell, (2744 atoms).

All binaries for the benchmarks are compiled on ARCHER with gfortran 4.9.2, and use FFTW 3.3.4 and Cray's LibSci 13.0.1, with the exception of figure 2 which was run earlier in the project.

WP1: Reduce memory per node

Work Package 1 aimed to reduce the amount of memory per node by reducing the number of MPI processes and replacing them with OpenMP threads.

Memory Improvements

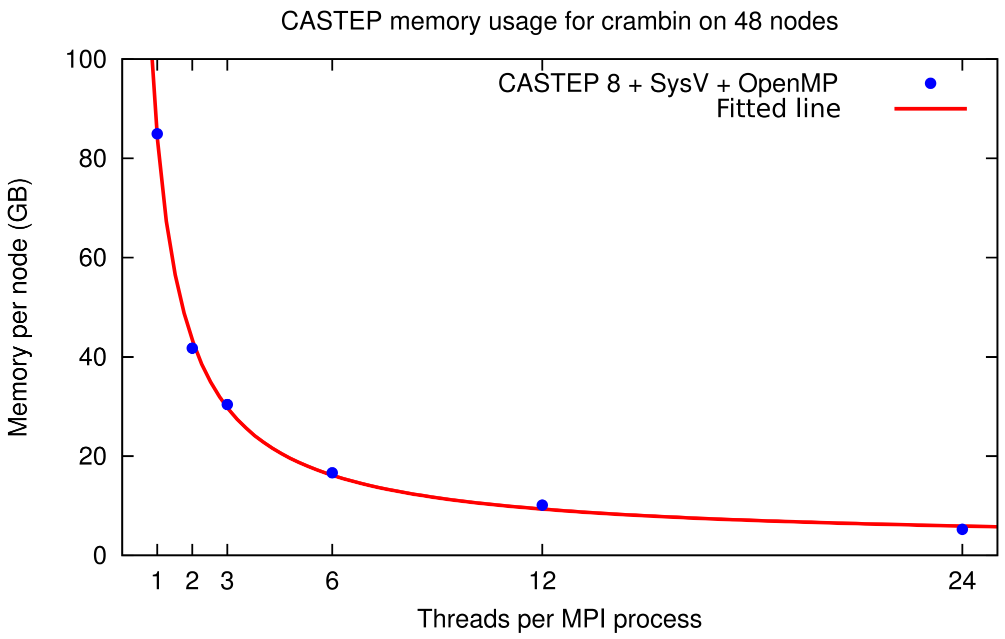


Figure 1: Memory per node as MPI processes are traded for OpenMP threads whilst running on 1152 cores. The line fits the function $24\left(\frac{R}{\#\ \text{threads}} + \frac{D}{1152\ \text{cores}}\right)$, with best fit values of $R = 3.41$GB and $D = 119.5$GB.

Figure 1 shows the memory per node for the crambin benchmark after the completion of WP1, calculated by reading the peak virtual memory from `/proc/self/status` on the root process at the end of the calculation and multiplying it by the number of MPI processes per node. This is an overestimate, since there are arrays only allocated on the root process. It is also unclear whether this includes memory allocated by the MPI library. Nevertheless, it clearly shows a dramatic reduction in the memory per node with the hybrid OpenMP+MPI approach, as MPI processes are traded for OpenMP threads (keeping the number of cores in use constant).

The crambin calculation requires approximately 123GB to run in serial. Of this, 119.5GB is distributed over MPI, and 3.41GB is replicated on every MPI process. If a large number of MPI processes are used, the distributed memory becomes negligible per MPI process. Meanwhile, the replicated data remains constant per MPI process, and therefore dominates the RAM usage per node. Therefore, with ARCHER and 24 MPI processes per node, the replicated data becomes 24×3.41GB $= 81.9$GB per node. This cannot be reduced by increasing the total number of MPI processes. By using a hybrid OpenMP+MPI approach, this replicated data can be shared between OpenMP threads, and hence the total memory per node can be dramatically reduced, as in figure 1.

Threaded Libraries

These memory savings are only generally useful if the performance of the hybrid code is not significantly worse compared to pure MPI. Initially, it was thought that linking with threaded libraries would provide an easy way to utilise threads in CASTEP. In particular, BLAS calls to DGEMM and ZGEMM for matrix multiplications took between 20-50% of the runtime in the crambin benchmark, depending on the number of threads. These are principally called from routines in `ion` and `wave` - see table 1.

Routine Name	# of calls	Time before WP1 (s)
Total runtime		2091.20
`ion_beta_add_multi_recip_all`	6317	485.12
`ion_all_beta_multi_phi_recip`	1390	402.83
`comms_transpose_exchange`	168320	246.96
`ewald_calculate_forces`	1	110.97
`wave_orthog_over_wv_slice`	688	96.15
`ewald_calculate_energy`	1	79.36

Table 1: Performance of crambin on 48 nodes, with 2 MPI processes per node to simulate under-populating a node, using CASTEP 8.0. The times are all averaged over 3 runs.

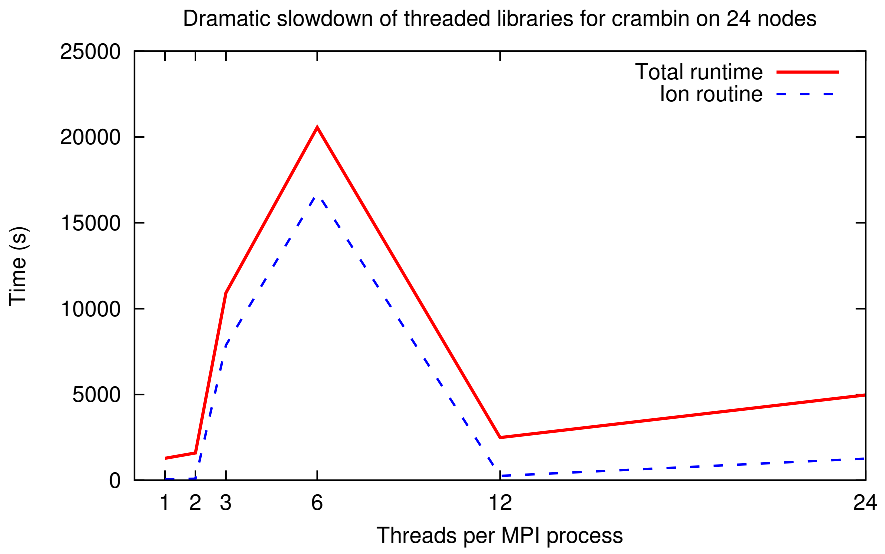


Figure 2: Times for the crambin benchmark running on 24 nodes of ARCHER with a partially threaded version of CASTEP. The dashed line shows the time spent in `ion_all_beta_multi_phi_recip`, where the threaded BLAS slowdown was most significant.

The dramatic slowdown with threaded BLAS, seen in figure 2 was found to be caused by the overhead of forking and joining threads, as the matrix multiplications in CASTEP are typically quite small, but called thousands of times. Hence the time for spawning and killing threads was significant which meant that the library threading performance was inconsistent, and could often increase the program's runtime by up to 10x. Because of this, matrix multiplications were wrapped up into a series of `algor_matmul` routines and the operations threaded explicitly using OpenMP. This allowed them to use existing OpenMP threads, and for control over how many threads to use for each multiplication.

There remains a small overhead associated with OpenMP thread forking/joining in these new routines, but not nearly to the same extent. The advantages gained by threading outweigh these effects. They are however the principal cause of the bump at 3 and 6 threads on the solid line in figure 3.

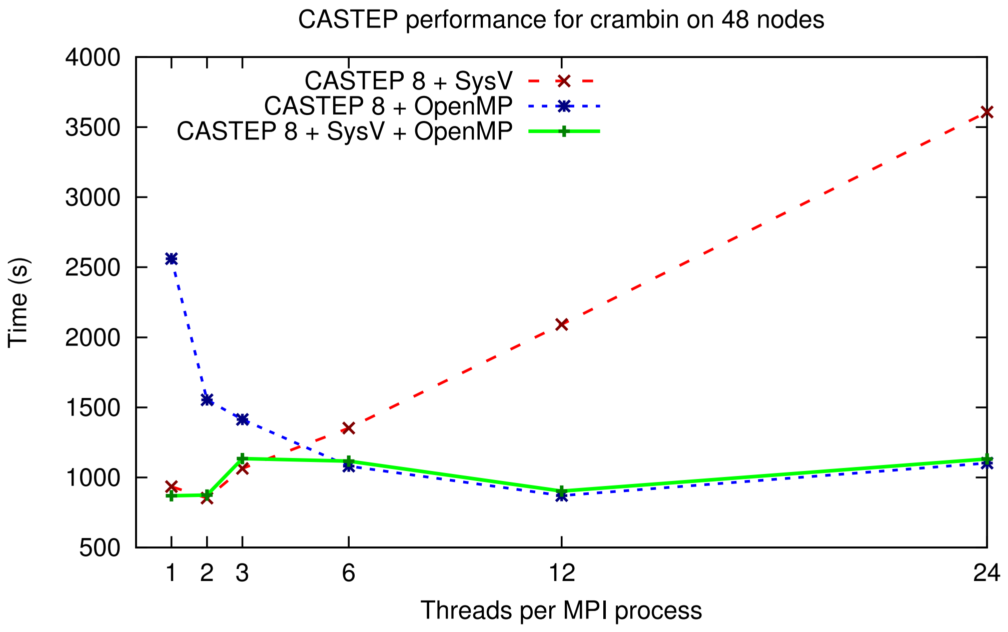


Figure 3: Times for the crambin benchmark running on 48 nodes of ARCHER with CASTEP 8.0, CASTEP 8.0 + OpenMP, and CASTEP 8.0 + OpenMP + System V SMP.

System V Shared Memory

One place that there was the potential to get noticeable performance improvements was in the parallel FFTs, where a number of `MPI_Alltoallv` communications are performed. This is one of the major bottlenecks in calculations on large process counts. By having fewer MPI processes and more threads, this time should be reduced, at the cost of increasing the computation time. This could be mitigated in the future by using a threaded FFT routine.

In CASTEP, this has previously been reduced using System V shared memory segments, whereby all N processes are grouped into groups of m MPI processes. Data for the all-to-all is then collected onto one of the m processes, and that process performs the MPI call. However this is not widely used and could potentially be replaced by using m OpenMP threads per MPI process. Figure 3 shows the performance of CASTEP before WP1, and with and without System V shared memory after WP1. In both cases that use System V, m was selected where possible such that there were 6 CPU cores in each group, for example 6 MPI processes on 1 thread and 3 MPI processes on 2 threads. For 12 and 24 threads, m was set to 1.

Final performance of WP1

Table 2 shows the exclusive time spent in the most time consuming routines in CASTEP 8.0, and their corresponding times after WP1. The performance improvements are due to 2 changes. Firstly, almost all of the time in the `ion` and `wave`, routines, along with a few smaller routines were in matrix multiplications, which have now been threaded in the `algor_matmul` routines, discussed above. Secondly, the `ewald` routines were optimised to take advantage of the separability of the Coulomb interaction in reciprocal space.

Routine Name	# of calls	Time before WP1 (s)	Time after WP1 (s)
Total runtime		2091.20	902.02
`ion_beta_add_multi_recip_all`	6317	485.12	98.21
`ion_all_beta_multi_phi_recip`	1390	402.83	105.71
`comms_transpose_exchange`	168320	246.96	266.83
`ewald_calculate_forces`	1	110.97	1.37
`wave_orthog_over_wv_slice`	688	96.15	10.17
`ewald_calculate_energy`	1	79.36	0.60
`Other`		669.81	419.13

Table 2: Performance of crambin on 48 nodes, 2 MPI processes per node. The final column shows the time with 12 OpenMP threads per MPI process. The net result is a speedup of over 2x by using OpenMP. The 20s difference in `comms_transpose_exchange` is due to timing noise on ARCHER.

WP2: Parallel matrix diagonaliser

Work package 2 aimed to implement a parallel diagonaliser in CASTEP. This was previously done using a serial or threaded LAPACK call to ZHEEVR. However, the scaling of this was limited the number of threads per process. The use of parallel libraries such as ScaLAPACK has been previously investigated. However they do not scale well, since they have their own parallelisation strategies and the cost of redistributing the data is too high.

A parallel Jacobi-like diagonaliser was chosen, based on the BFJ algorithm by Littlefield and Maschhoff[5]. The advantage of writing the diagonaliser within CASTEP is that CASTEP's pre-existing parallelisation scheme can be utilised, and data will not have to be gathered and redistributed.

Jacobi's method

Jacobi's eigenvalue algorithm[6] aims to solve the Hermitian eigenvalue problem

$$\Psi^{+} H \Psi = \varepsilon \tag{1}$$

by iteratively reducing the Frobenius norm of the off diagonal elements:

$$\text{off}(H) = \sqrt{\sum_{i=1}^{n} \sum_{j \neq i}^{n} h_{ij}}\ . \tag{2}$$

This is done by cycling over the off-diagonal elements in the upper triangle

$$h_{ij};\ i = 1 \rightarrow (N-1),\ j = (i+1) \rightarrow N\,.$$

Each time, the i'th and j'th rows and columns of H, and the i'th and j'th rows of Ψ are updated such that h_{ij} is zeroed.

By rewriting equation 1 as

$$(\Psi^{+} H)\Psi = G\Psi = \varepsilon\ , \tag{3}$$

h_{ij} can be zeroed by only updating columns of G and rows of Ψ. This is known as the one sided Jacobi algorithm and allows the columns of G and Ψ^{T} to be updated independently from each other, and distributed across multiple processes.

Parallel Distribution

The order in which the columns of G and Ψ^{T} are updated does not matter, provided each i, j pair is updated once and only once per iteration. One way of doing this is by dividing the columns up into 2 groups, and distributing both groups across all processes. An example of this can be seen in table 3.

	process 1			process 2			process 3		
Group 1	1	2	3	7	8	9	13	14	15
Group 2	4	5	6	10	11	12	16	17	18

Table 3: Parallel distribution of columns for an 18×18 matrix on 3 processes.

For a given iteration, each process updates every column with every other column in each group individually. For example, process 1 in table 3 would update columns (1,2), (1,3) and (2,3) for group 1 and (4,5), (4,6) and (5,6) for group 2.

Next, all inter-group pairs are updated on each process. E.g. process 1 would rotate (1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5) and (3,6).

After this is done, all the groups of columns *except* group 1 on process 1 are cycled anticlockwise in a ring. The result of this is shown in table 4.

	process 1			process 2			process 3		
Group 1	1	2	3	13	14	15	16	17	18
Group 2	7	8	9	4	5	6	10	11	12

Table 4: Parallel distribution of columns after 1 pass.

The inter-group columns are updated on each process as before, but now with the pairs of groups. This action of cycling and updating the columns is repeated until all the columns are back where they started. This completes one Jacobi iteration.

Since this work is replicated across the g-vectors, and since calculations big enough for the work to be significant is generally well parallelised over g-vectors, it was decided that columns should be distributed over the g-vector group.

Performance

While Jacobi's algorithm has more scope for parallelism it is slower in serial than the LAPACK routines, so has not entirely replaced the old diagonaliser in CASTEP. It was found that Jacobi started to beat LAPACK at 24 processes, and on matrices larger than 750x750. CASTEP automatically chooses which algorithm to use based on these criteria. Figure 4 shows the parallel performance of ZHEEV and CASTEP's Jacobi algorithm on varying numbers of cores.

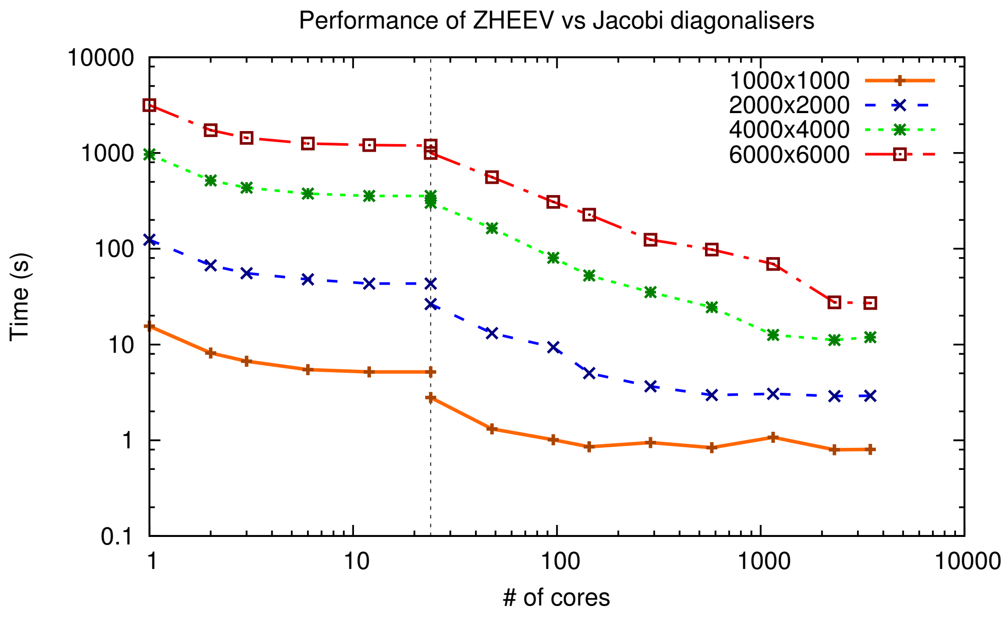
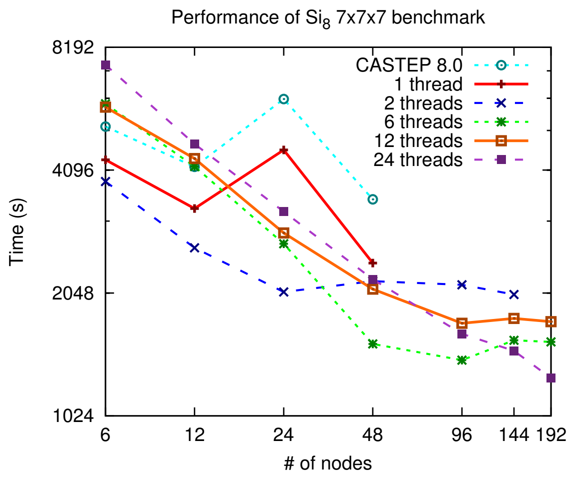


Figure 4: Comparison of ZHEEV (1-24 cores) and Jacobi (24-3456 cores) diagonalisers on various sized matrices. ZHEEV is using LibSci's threading, and Jacobi is MPI parallel. Note the logarithmic scale on both axes.

It can be seen from figure 4 that, while ZHEEV saturates around 6 threads, Jacobi is able to scale up to $\frac{N}{4}$ cores. Beyond this, there are not enough columns to distribute across any remaining processes. Beyond this point, performance does not change significantly as additional cores are not used.

Results



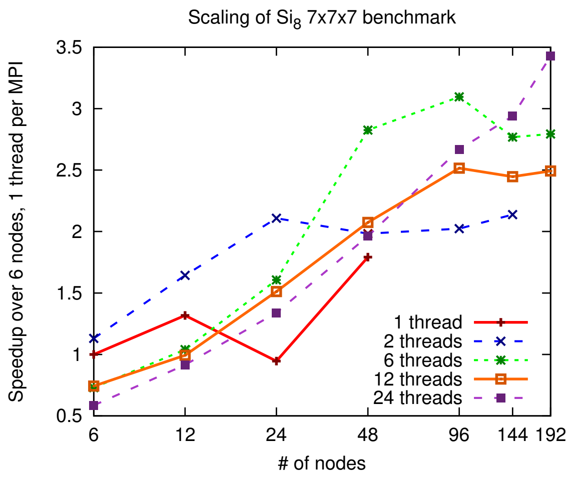


Figure 5: Performance of the Si_8 7x7x7 supercell benchmark on different numbers of nodes in CASTEP 9.0. The times are also compared to that of CASTEP 8.0. Here, each System V SMP group contains 12 cores.

Figure 5 shows how CASTEP 9.0 scales with core count, on varying numbers of nodes. It shows that while lower thread numbers are faster for small core counts, higher thread numbers scale much better and are eventually much faster. For example, 2 threads per MPI process is the fastest on 6, 12 and 24 nodes, but does not scale beyond that. 24 threads however is the slowest on 12 and 24 nodes, but is able to scale right up to 144 and 192 nodes, where it is the fastest.

This is not surprising as it is the communications that are the bottleneck in the scaling limit. Since higher thread numbers means fewer MPI processes, this communication time would be expected to reduce.

It is also interesting to note that calculations with 1 thread per MPI process ran out of memory after 48 nodes, and 2 threads per MPI process ran out at 192 nodes.

Future work

WP1 While all the major bottlenecks for large calculations have been threaded where possible, the profile for smaller systems can be significantly different. While beyond the scope of this project, threading could be implemented in such routines, allowing non-MPI CASTEP binaries to still make use of multi-core machines.

In addition, new technologies such as the Xeon Phi and GPU cards make heavy use of many-threaded architectures. Extensions into OpenMP 4 or OpenACC will allow for such accelerators to be used more effectively. Alternatively, MPI 3 provides a shared memory interface which, while still relatively new, could be of benefit and may be worth investigating.

WP2 Now that the diagonaliser is in place in CASTEP, various optimisations could be made to it, for example with improved blocking or asynchronous communications. The Littlefield paper[5] does suggest a way of doing a 2D distribution of the work, enabling the calculation to scale to higher numbers of cores. This was implemented using threading, but performance for medium sized matrices was poor, so this was left out. However it may well still possible, either with a different approach to the threading or by distributing in 2D over MPI.

Beyond the diagonaliser, matrix inversion is another task which is currently not MPI parallel. While not a bottleneck at the moment, this scales cubically with the system size and will eventuall dominate the run-time for a large enough system.

Summary

This project set out to reduce memory per node of large calculations, without affecting the runtime of the calculation. In addition to this, it also aimed to improve the parallel scaling of such calculations by distributing the work of the matrix diagonalisation.

By implementing OpenMP threading and making selected use of threaded BLAS and LAPACK libraries, the number of MPI processes per node has been reduced without any significant slowdown. Indeed, on large core counts, using threads often turns out to be faster than not. By doing this, the amount of memory per node has been reduced dramatically.

By implementing a parallel matrix diagonaliser, calculations on large systems has sped up significantly, with systems such as Si_8 7x7x7 (6500 bands) seeing a reduction in the time spent in the diagonaliser from over 1200 seconds to less than 200 seconds on 48 nodes.

Acknowledgements

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

References

[1] S.J. Clark et al., *First principles methods using CASTEP*; Zeitschrift für Kristallographie - Crystalline Materials, (2005) 220: 5-6 p567.

[2] *CASTEP web page*; http://www.castep.org Accessed 31 August, 2015.

[3] A. Turner, *Parallel Software usage on UK National HPC Facilities 2009 - 2015*; ARCHER White paper, 2015.

[4] *ARCHER Usage Data*; http://www.archer.ac.uk/documentation/white-papers/app-usage/ARCHER_report.txt, Accessed 28 August, 2015.

[5] R. Littlefield, K. Maschhoff, *Investigating the performance of parallel eigensolvers for large processor counts*; Theoretica Chemica Acta (1993) 84: p457-473.

[6] G. Golub, C. Van Loan, *Matrix computations 4th Edition*; John Hopkins University Press, 2013.