eCSE08-10: Optimal parallelisation in CASTEP

Arjen, Tamerus
University of Cambridge
at748@cam.ac.uk

Phil, Hasnip
University of York
phil.hasnip@york.ac.uk

July 31, 2017

Abstract

We describe an improved implementation of OpenMP multithreading in CASTEP, with hybrid MPI/OpenMP execution approaching pure-MPI performance levels at a reduced memory footprint. Additionally, a toolkit was implemented to guide users to improve their utilisation of parallel machines running CASTEP.

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

1 Introduction & Project description

CASTEP [1] is a high-performance implementation of density functional theory for first-principles materials modelling. CASTEP describes the electronic states ("bands") of the material using a plane-wave (Fourier) basis, using parallel fast Fourier transforms (FFTs) to convert between Fourier space ("reciprocal space") and the direct space ("real space").

CASTEP was written to be MPI-parallel from the outset, but in recent years this distributed-memory parallelism has been supplemented with shared-memory multithreading via OpenMP. Combining these two parallel modes enables CASTEP to scale well up to 10,000 cores and beyond (see Fig. 1).

In this report we describe work to improve the existing implementation of multithreading in CASTEP using OpenMP, with the aim of enabling even higher levels of parallelism through hybrid MPI/OpenMP. We achieve this through optimising both internal CASTEP routines, and the use of threaded libraries.

We also report our efforts in developing a guided tool to running CASTEP with an optimal parallelisation strategy.

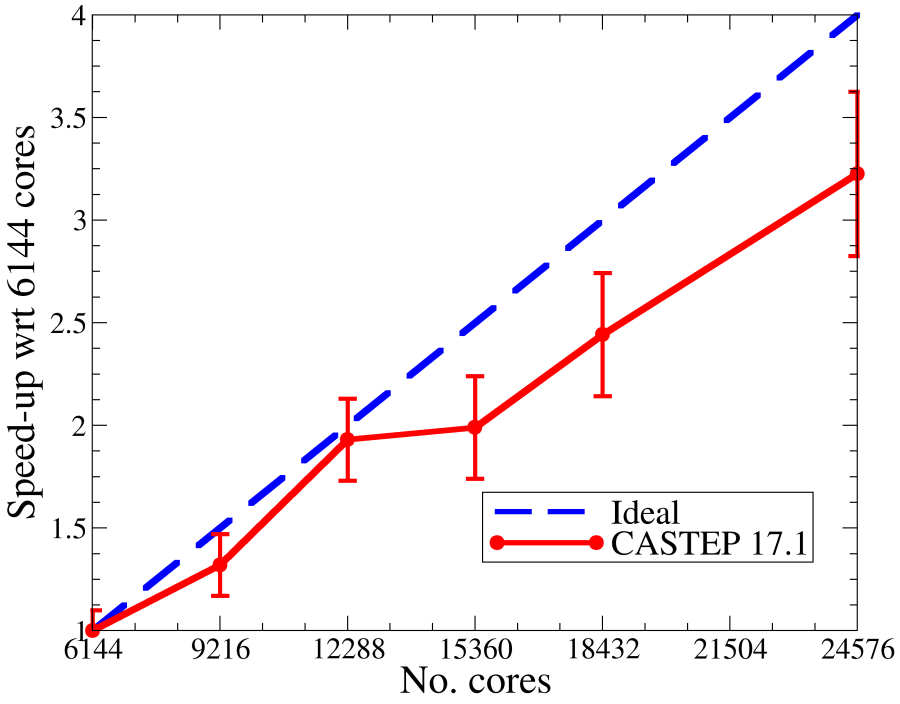

Figure 1: Parallel scaling performance for a simulation of poly-A DNA in vacuum, run on ARCHER. The parallel speed-up is reported as compared to the performance on 6144 cores, which are the fewest cores the simulation will run on using only MPI.

2 Methodology

2.1 Testing environment

Main development and testing was performed on the Darwin and Wilkes clusters hosted at the University of Cambridge. Darwin nodes run dual 8-core Intel Xeon (Model, Sandy Bridge) CPUs; Wilkes is equipped with two six-core Xeon (Model, Ivy Bridge) CPUs per node. Final results were obtained on ARCHER.

Performance testing and optimisation during development was performed with both GNU and Intel compilers and various BLAS, LAPACK and FFT libraries. The compilers used were gfortran 5.3 and Intel fce 16.3.210. MPI functionality was provided by Intel MPI 5.1.3.181 or MVAPICH2 2.1. BLAS/LAPACK and FFT routines were provided by OpenBLAS 0.2.19 and FFTW, 3.3.5, respectively, or by Intel MKL 11.3.3.210.

Two tools were used to profile CASTEP: the internal, built-in profiling tool "trace", and score-p[1].

2.2 Profiling methodology

The main tool used to profile the (OpenMP) performance for CASTEP was its built-in profiling tool. Although its functionality is limited to only reporting the wall clock time spent in subroutines or libraries, the reported timings are reliable for all but the shortest calculations and the impact on performance is minimal (typically 2s).

The timings reported by the internal profiler were used to quickly identify OpenMP performance issues, either due to a lack of thread-parallelism in the code, or due to load imbalance. The simple timings quickly tell us whether and how well routines scale with OpenMP threads.

In order to comprehensively test CASTEP's scaling performance, a number of benchmarks were chosen that represent a range of realistic workloads, in particular covering ground state calculations with general k-point sampling and those using 'gamma-point-only' sampling; when only the gamma-point is used, a large amount of complex-complex arithmetic reduces to real-real, leading to an 8-fold reduction in computational time for large simulations. In base testing, each input case was run on 2 nodes of the test system, using either n MPI processes, 2 OpenMP threads per process with $n/2$ MPI processes, or 4 OpenMP threads and $n/4$ processes.

By running in this way, we could quickly identify areas of the code that did not scale satisfactorily by examining the increase in wall clock time per routine when increasing the level of threading.

Since CASTEP's internal profiler does not support reporting more than per-process wall time, score-p was used to get more detailed information, most importantly potential load imbalance between OpenMP threads.

[1] `http://score-p.org`

3 Optimisations

3.1 Internal routines

Optimisation of the internal CASTEP routines was a mostly mechanical process. By analysing the profiles generated by score-p and the internal profiler, we identified routines that showed a slowdown when running with OpenMP. Tagging routines would be placed around suspect loops and function calls that allowed CASTEP's profiler to identify these new, smaller regions.

These tags were used to identify the loops that were the source of the slowdown. OpenMP directives were then placed in the appropriate places, using parallel regions, reductions and loop collapsing where appropriate.

3.2 Libraries

A significant amount of CASTEP's CPU time is spent in library routines, specifically BLAS and FFT routines. Here we will describe how these libraries have been optimised to make use of OpenMP multithreading.

3.2.1 BLAS

CASTEP 16.1 has limited support for multithreading support in libraries; specifically, there is a manually threaded interface to a subset of the most important BLAS routines (e.g. ZGEMM/DGEMM and ZDOTC/DDOT). Our benchmarks showed that this interface resulted in a more consistent OpenMP performance than relying on the libraries' own OpenMP interfaces, especially in the case of OpenBLAS. The overall better performance of the manual interface led to the decision to keep and expand upon this interface for algebraic routines.

The predominant BLAS routines in terms of time consumption are variations of GEMM and TRMM. A parallel GEMM interface was developed in an earlier eCSE project (eCSE01-17), and only minor optimisations were performed in this work; its scaling is now similar to that using pure MPI parallelism. An interface for TRMM was not implemented in the earlier eCSE work, but has been created over the course of this project and scales similarly well.

Further optimisations include analysing the code for occurrences of direct calls to BLAS calls that have a parallel interface available, and using the interface instead. Furthermore some calls were identified where memory access patterns were non-optimal; these have been modified to run more efficiently.

3.2.2 FFT

In CASTEP 16.1, support for threaded FFTs was not yet implemented. In this eCSE project, OpenMP support was enabled for FFTW3 and MKL, and manually implemented for the GPFA[2] FFT algorithm, which is bundled with CASTEP.

Enabling OpenMP support in FFTW is relatively straightforward, but should only be enabled when desired. To ensure this, we make sure that OpenMP is

initialised to multiple threads when calling the FFTW initialisation routines. We use a guard clause to ensure the OpenMP initialisation routines are only called when OMP_MAX_THREADS is greater than 1, as shown in Listing 1.

Listing 1: Enabling OpenMP support in FFTW3

```
if (openmp_initialised == 0) then
  openmp_initialised = omp_get_max_threads()
  if (openmp_initialised .gt. 1) then
    call dfftw_init_threads(stat)
    call dfftw_plan_with_nthreads(openmp_initialised,stat)
  else
    openmp_initialised = 1
  endif
endif
```

A similar construct was used for MKL. Using the OpenMP domains provided by MKL allows the library to set the number of OpenMP threads only for part of its functionality, in this case the FFT routines. By setting the overall number of threads to 1 (as required for the BLAS routines) and calling MKL_DOMAIN_SET_NUM_THREADS(NTHREADS, MKL_DOMAIN_FFT) to enable OpenMP only for the FFT functions. This solution however was discarded in favour of the less elegant but more effective use of MKL's dynamic parallelism, as described in 3.3.

The GPFA library required a manual OpenMP implementation. Since this is not a strongly performing library by nature, a course-grained multithreading model was deemed sufficient. Parallelisation of 3-dimensional FFTs was achieved by enclosing independent calls to the subroutines FFT_SERIAL_GPFA and FFT_SERIAL_SETGPFA in parallel OpenMP directives, within the main routine FFT_SERIAL_GPF3D (names were left unchanged for compatibility reasons). Additionally, an independent loop in the GPFA routine was parallelised similarly.

3.3 Dynamic parallelism

Intel's MKL has support for dynamic OpenMP multithreading using the environment variable MKL_DYNAMIC. Enabling this feature makes MKL scale the amount of OpenMP threads up or down, based on the number of active threads at the time a routine is called. Using this feature allows the use of CASTEP's manual threading interface to BLAS, whilst also utilising the internal multithreaded implementation for those routines that have not (yet) been implemented manually. Furthermore this is now the preferred method for enabling threaded FFTs (as mentioned in3.2.2).

3.4 Runtime optimisation

To ensure optimal performance, some runtime optimisation is required. Specifically, proper core and thread mapping and binding is paramount to guaran-

tee optimal OpenMP performance. Especially on multi-socket nodes, OpenMP threads belonging to an MPI process should be located on the same physical CPU (where possible). Figure2 shows the optimal mapping for a 4MPI/4OMP setup on a Darwin node. We scatter the MPI processes over the 2 physical CPUs, and bind the OpenMP threads to the cores nearest to the processes.

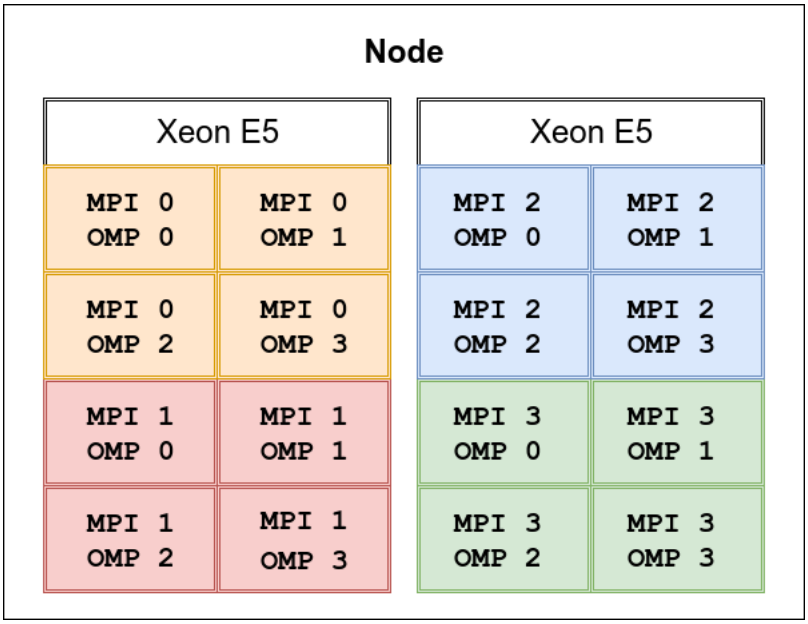

Figure 2: Good CPU binding on Darwin

Using this mapping, the performance of GEMM routines is close or identical to purely MPI-parallelised runs. Without this explicit mapping we observed outliers performing up to 30 percent slower.

4 Performance analysis

The performance of the new OpenMP code is shown in Figure 3, for a representative benchmark (CASTEP's standard al3x3 benchmark). All of the calculations here were hybrid OpenMP-MPI runs, with 192 MPI processes, but different numbers of OpenMP threads per MPI process, ranging from 1 (pure MPI) to 24 (the number of physical cores/node on ARCHER). The single-threaded performance is identical in both cases, as expected, but the performance is improved for all OpenMP threads, with the improvement being greater for the larger thread counts. For CASTEP 16.1.1 the shortest run-time was achieved using 6 threads/process (for a total of $192 \times 6 = 1152$ cores); the OpenMP improvements in this project led to a speed-up of nearly 10% for this calculation, but the improved scaling of the new code means the quickest calculation was actually for

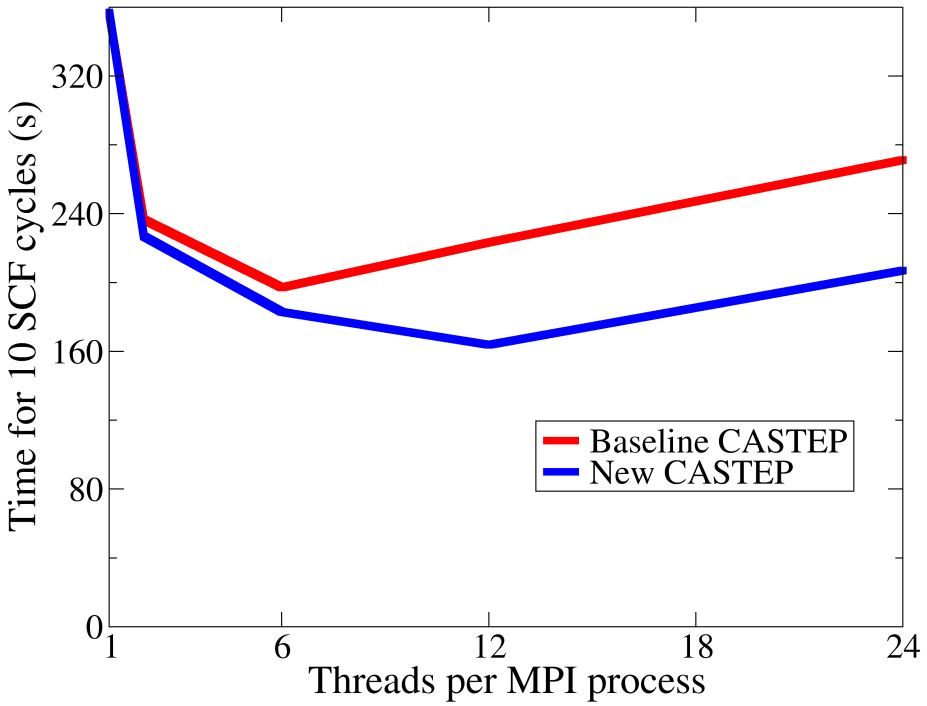

Figure 3: Parallel OpenMP scaling for a simulation of a 3x3 sapphire surface (CASTEP's al3x3 benchmark), run on 192 MPI processes on ARCHER with a range of OpenMP threads per MPI process. The time taken for 1 thread/process is unchanged, but the time for multiple threads/process is improved significantly across the entire range. The shortest run-time is now achieved with 12 threads/process, twice the previous optimal count.

12 threads/process (for a total of $192 \times 6 = 2304$ cores). This 12 thread/process calculation was over 25% faster than the same run using CASTEP 16.1.1.

There are two main issues hindering full performance parity between pure MPI and hybrid MPI/OpenMP. The first is a call to MPI_Allgatherv in the 3D FFT routines; this operation shows a slowdown when the number of processes decreases. The second is the use of indirection in accessing a multi-dimensional array in the FFT calling subroutines `basis_recip_reduced_to_real` and `basis_real_to_recip_reduced`. We suspect that we are suffering from cache thrashing here, but unfortunately, we did not have time to perform in-depth analysis of these issues.

5 Parallel tuning

Apart from balancing the MPI to OpenMP ratio, CASTEP has a number of parameters that affect its parallel performance, depending on the input case. These parameters require manual set-up through CASTEP's simulation input files, and fine-tuning requires in-depth knowledge of CASTEP's parallelism model and the computer hardware. As a result many users may run with non-optimal settings. As a part of this eCSE project, we aimed to develop a toolset that would, through a short 'microbenchmark', automatically find the optimal parameters for a given input case.

5.1 The microbenchmark

Two of the major operations influencing the performance of a CASTEP simulation are `wave_rotation` and `wave_real_to_recip`/`wave_recip_to_real`, containing calls to BLAS (ZGEMM or DGEMM, depending on the simulation parameters) and 3D FFTs, respectively. Since the sizes of the data structures these operations are performed on differ per input case, this has to be taken into account to get a reliable benchmark.

To get the correct sizes, we integrated the benchmark in the initialisation phase of CASTEP. We can use existing routines to extract the required sizes for the ZGEMM/DGEMM and FFT arrays, and use dummy data to perform a number of calls to the routines. We use MPI_Walltime to calculate the time spent both in the `wave_rotate` and FFT routines. This provides us with an indication of overall performance, and can be used to determine whether we should use band or G-vector parallelism.

5.2 Parallel FFTs and shared memory

The parallel 3D FFT requires two MPI_AlltoAllV calls across the G-vector group for each band, at each k-point. As the number of MPI processes N increases, the total number of MPI messages increases as N^2, and the size of each message decreases as $\frac{1}{N^2}$, meaning that the operation becomes latency-dominated. Since each process sends N messages (of the total N^2), the time for the AlltoAllV increases as N; this is exacerbated on modern HPC systems by the large number of cores on each node (24 on ARCHER) and the limited number of NICs available to each node (typically only 1 or 2), leading to severe device-contention and an *increase* in the effective message latency.

Both the small message size and the contention for NICs can be ameliorated if cores on the same node aggregate their messages, and only a small number of cores on each node perform the MPI_AlltoAllV. Aggregating the messages increases the message size, making more effective use of the available interconnect bandwidth, reduces the number of messages and, with fewer processes involved in the inter-node communications, the contention for the NICs is also reduced. The optimal number of processes which should be involved in the inter-node communications depends on the number of available NICs, the latency and

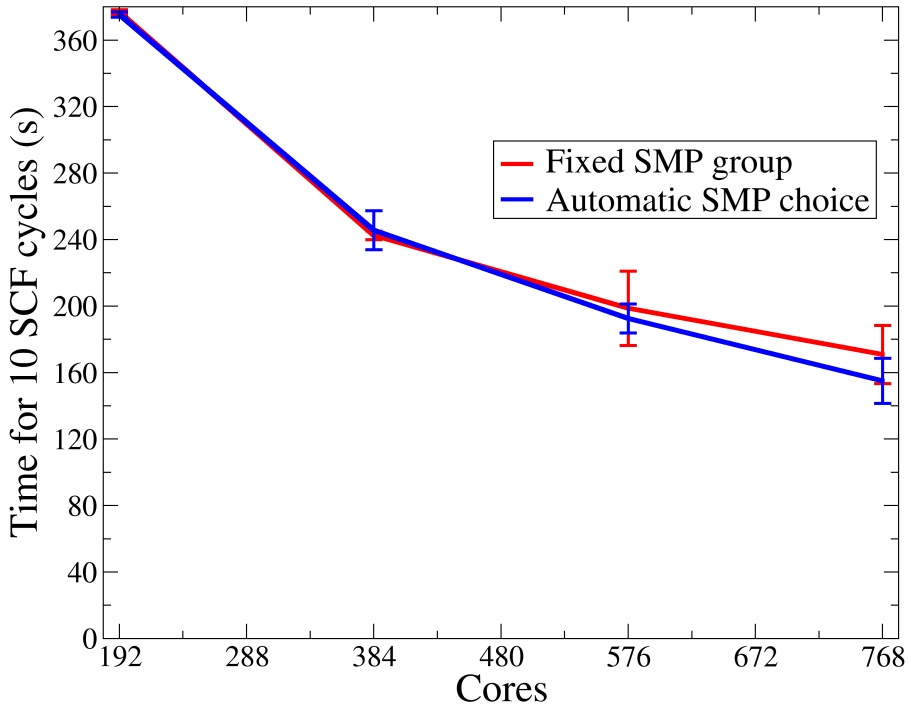

Figure 4: A comparison of the parallel run-times with and without the parallel performance model, for a simulation of a 3x3 sapphire surface (CASTEP's al3x3 benchmark), run on ARCHER using 192-768 cores (6 threads per MPI process). For the fixed SMP group size, the usual recommendation of 6 was used; this is confirmed to be a good choice by the parallel model, but 12 is sometimes slightly better, leading to an improvement in run-times when using the parallel model.

band-width of the interconnect, and the time it takes for message aggregation and (following the inter-node communication) dissemination.

The message aggregation and dissemination could be done entirely within MPI, using an intra-node communicator, but since the cores all have access to the same physical RAM it is more efficient to use an explicit shared-memory interface. CASTEP has the ability to use POSIX or System V shared memory to provide direct memory access between MPI processes, reducing the MPI communication. Like other performance 'fine-tuning factors', enabling this functionality and setting the amount of processes per shared memory region was left to the user. In the modified code, the user may optionally set a maximum number of processes per group; the code will benchmark sensible values for the shared-memory pool size and select the optimal setting. This capability has been merged recently into the current CASTEP codebase, and will be available to all users in next year's release (expected to be CASTEP 18.2).

5.3 Parallelisation strategy

Four different kinds of parallel decomposition are used in CASTEP: farm, k-point, G-vector and band parallelism. Farm and k-point parallelism are independent forms of parallelism, with very little inter-process communications (occasional reductions and synchronisation points). G-vector and band parallelism however do require significant communication between processes, and so increasing the number of processes comes at the cost of increased communications.

While generally G-vector parallelism is preferred over band parallelism, in certain cases the reverse may yield better performance. An initial method to choose the optimum decomposition was implemented by taking the number of MPI processes available after assigning them to farms and/or k-points. We benchmark distributions over different numbers of bands or G-vectors, at varying idle fractions (number of CPU cores left idle); the fastest performance on the microbenchmark is likely the most efficient strategy.

Currently, CASTEP uses heuristics to decide an optimal strategy with a maximum idle fraction decided by the user (default: up to 30% of MPI processes). By taking this decision out of the users' hands and deciding through benchmarking, we rule out a situation where the maximum idle is set too low, forcing CASTEP to choose a less than optimal strategy.

Unfortunately, we ran into stability issues while changing the parallelisation strategy at runtime. Within the time available during this eCSE project we lacked the resources to successfully work around these issues, leading to this aspect of the decomposition functionality not currently being included in CASTEP.

6 Conclusions

The OpenMP multithreading implementation in CASTEP was optimised, resulting in an improved performance approaching that of the native MPI implementation, and in specific cases even exceeding it. Improvements were made both to the threading of internal routines as well as improving the use of parallel BLAS and FFT libraries. Apart from just increasing Hybrid OpenMP performance, by improving the level of threading we also achieve a decrease in memory usage, allowing the computation of larger models on a given machine.

Additionally, a lightweight benchmark was written and integrated with CASTEP, which allows CASTEP to determine an improved parallel decomposition and the size of any POSIX/System V shared memory groups. This framework was designed to be extensible, so that functionality may be added straightforwardly in future projects.

These two classes of performance improvements are independent of each other, and may be combined to provide substantial performance gains over previous CASTEP calculations. These performance gains are achieved whilst simultaneously reducing the need for users to perform detailed benchmarking

themselves, and similarly reducing the technical knowledge users need in order to use CASTEP efficiently on multicore HPC nodes.

References

[1] S. J. Clark, M. D. Segall, C. J. Pickard, P. J. Hasnip, M. J. Probert, K. Refson, and M. C. Payne. First principles methods using castep. *Z. Kristall.*, 220(5-6):567–570, 2005.

[2] Clive Temperton. A generalized prime factor fft algorithm for any n=2ˆp3ˆq5ˆr. *SIAM Journal on Scientific and Statistical Computing*, 13(3):676–686, 1992.