Bands-parallelism in Castep
A dCSE Project

Phil Hasnip

September 1, 2008

Contents

1 Introduction **3**
- 1.1 The dCSE Project . . . 4
- 1.2 Summary of Progress . . . 4

2 Castep **7**
- 2.1 Parallelisation . . . 9
- 2.2 Computational Costs . . . 10
  - 2.2.1 Orthogonalisation and Diagonalisation . . . 10
  - 2.2.2 Fourier Transforms . . . 10

3 Castep Performance on HECToR (Work Package 0) **12**
- 3.1 General Castep Performance . . . 12
- 3.2 Benchmarks . . . 13
  - 3.2.1 FFT . . . 14
  - 3.2.2 Maths Libraries (BLAS) . . . 15
  - 3.2.3 Compiler . . . 15
  - 3.2.4 Node Usage . . . 17
- 3.3 Baseline . . . 18
- 3.4 Analysis . . . 18

4 Band-Parallelism (Work Package 1) **26**
- 4.1 Introduction . . . 26
- 4.2 Programming . . . 26
  - 4.2.1 Comms . . . 26
  - 4.2.2 Ion . . . 27
  - 4.2.3 Wave . . . 28

4.2.4 wave_write 32
4.2.5 Density 34
4.2.6 Ewald 34
4.2.7 Electronic 34
4.2.8 Model 34
4.2.9 Secondd 34
4.2.10 Phonon 35
4.3 Testing 35
4.4 Benchmarking and Performance 37
4.5 Scope of Parallelism and Integration with main Castep Codebase 40

5 Distributed Diagonaliser and Inverter (Work Package 2) 42
5.1 Introduction 42
5.2 Programming 42
5.2.1 Development 42
5.2.2 ScaLAPACK Performance 43
5.2.3 Castep Implementation 45
5.2.4 Castep Performance 46
5.2.5 Limitations 47

6 Independent Band Optimisation (Work Package 3) 49
6.1 Introduction 49
6.2 Castep Eigensolver 49
6.3 Proposed Eigensolver 50
6.4 Performance 52

7 Other Developments and Future Work 55
7.1 Γ-Point 55
7.2 Distributing β-Projectors and $\langle\beta|\,\phi\rangle$ 56
7.3 Mpitrace 57

8 Final Thoughts 58
8.1 This Project 58
8.2 HECToR as a Development Machine 59

Chapter 1

Introduction

This Distributed Computational Science and Engineering (dCSE) project is to implement a new parallelisation strategy in the density functional theory program Castep[1], on top of the existing parallelisation if possible, in order to extend the number of nodes that Castep can be run on efficiently. Although benchmarking Castep performance is a part of this dCSE project, it is anticipated that this will allow Castep to run efficiently on O(1000) processing elements (PEs) of the HECToR national supercomputer.

Castep is widely used in materials science, chemistry, physics, engineering and, increasingly, molecular biology research, in both industrial and academic research institutions, and for this reason it was included as one of the benchmark programs used in the HECToR procurement exercise. It is marketed worldwide by Accelrys, but is available free to UK academics under the terms of the United Kingdom Car-Parrinello Consortium (UKCP), and it is under the terms of this agreement that Castep is made available on HECToR. Increasing the efficiency of Castep's parallelisation strategies will not only enable HECToR to be used more productively, it will enable considerably larger simulations to be performed and open up many new avenues of research.

This dCSE project commenced 1st December 2007, and is scheduled to end on the 31st July 2008. The Principal Investigator on the grant was Dr K. Refson (RAL). Dr M.I.J. Probert (York) and Dr M. Plummer (STFC) also provided help and support.

1.1 The dCSE Project

At the heart of a Castep calculation is the iterative solution of a large eigenvalue problem to find the lowest 1% or so of the eigenstates, called 'bands', and this calculation is currently parallelised over the components of the bands. The aim of this project was to implement an additional level of parallelism by distributing the bands themselves. The project was to be comprised of four phases:

1. Basic Band Parallelism
   Split the storage and workload of the dominant parts of a Castep calculation over the 'bands', in addition to the current parallelisation scheme.

2. Distributed Matrix Inversion and Diagonalisation
   At various points of a Castep calculation large matrices need to be inverted or diagonalised, and this is currently done in serial. In this phase we will distribute this workload over as many processors as possible.

3. Band-Independent Optimiser
   The current optimisation of the bands requires frequent, expensive orthonormalisation steps that will even more expensive with the new bands-parallelism. Implementing a different, known optimisation algorithm that does not require such frequent orthonormalisation should improve speed and scaling.

4. Parallelisation and Optimisation of New Band Optimiser
   To work on making the new optimiser as fast and robust as possible, and parallelise the new band optimisation algorithm.

The overall aim was to enable Castep to scale efficiently to at least eight times more nodes on HECToR. Unfortunately only the first three phases of the project were funded, but the aim remained to achieve as great an increase in scaling as possible. An additional phase was introduced at the request of NAG, to investigate Castep performance on HECToR in general to determine the best compiler, compiler flags and libraries to use.

1.2 Summary of Progress

All three phases of the project have been completed successfully, based on Castep 4.2 source code, though there remains some scope for optimisation and several possible extensions. Basic Castep calculations can be parallelised over bands in addition to the usual parallelisation schemes, and the large matrix diagonalisation and inversion operations have also been parallelised. Two band-independent optimisation schemes have been implemented and shown to work under certain conditions, but unfortunately neither is fast or robust enough to be useful to a general Castep user, and neither have been parallelised fully.

Despite the lack of an effective band-independent optimiser, the performance of Castep on HECToR has been improved dramatically by this dCSE project. One example is the standard benchmark `al3x3`, which now scales effectively to almost four times the number of cores compared to the ordinary Castep 4.2 (see figure 1.1).

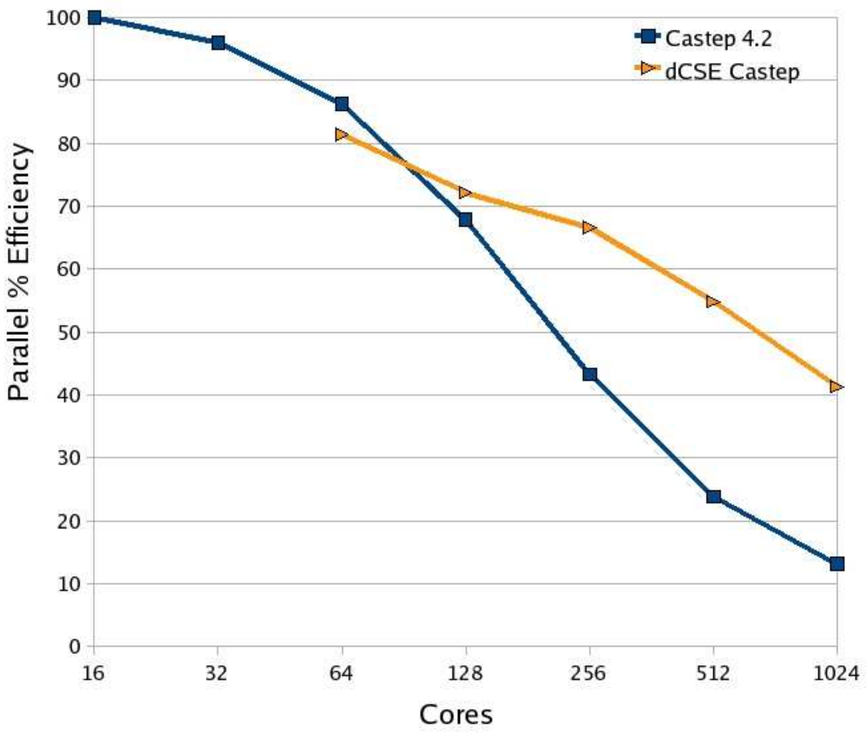


Figure 1.1: Graph showing the performance and scaling improvement achieved by this dCSE project (using 8-way band-parallelism) compared to the ordinary Castep 4.2 code for the standard `al3x3` benchmark.

The changes made in this dCSE project are expected to be merged into the main Castep source code for version 4.4, due for release in the latter quarter of 2008. It is hoped that many of the restrictions of the new scheme will be lifted in time for Castep 4.5.

Chapter 2

Castep

Castep is a plane-wave, pseudopotential density-functional theory program for calculating the groundstate electronic charge density of a periodic system of electrons and nuclei.

The main computational effort is the search for the Kohn-Sham wavefunctions and groundstate electronic charge density for a fixed, 3D-periodic configuration of ions. This requires the solution of a series of one-particle Schrödinger equations:

$$\hat{H}\psi_{bk}(r) = E_{bk}\psi_{bk}(r) \tag{2.1}$$

where $\hat{H}$ is the Hamiltonian, the index b labels different eigenstates, also known as *bands*, and the index k labels different sampling points ('k-points') in the reciprocal-space Brillouin zone of the periodic simulation cell. $\{\psi_{bk}(r)\}$ are the Kohn-Sham wavefunctions, the eigenstates of these Schrödinger equations, and $\{E_{bk}\}$ are the corresponding band-energies.

The equations for different k-points are not completely independent, but interact via the electronic charge density $n(r)$

$$n(r) = \sum_{bk} f_{bk} \int |\psi_{bk}(r)|^2 d^3r \tag{2.2}$$

where f_{bk} is the occupancy of band b at k-point k. For insulating systems all bands below the Fermi energy are fully occupied ($f_{bk} = 1$) and all bands above the Fermi energy are unoccupied ($f_{bk} = 0$). For metals it is common to introduce a small thermal-like distribution function which smears the occupancies in the vicinity of the Fermi level.

For the 3D periodic systems Castep is designed to simulate, the density shares the same periodicity as the simulation system and so it is natural to express it in a Fourier basis. The basis functions are plane-waves whose wave-vectors are reciprocal lattice vectors of the simulation cell, the so-called G-vectors. The bands themselves need not be quite periodic, but they can always be written as the product of a phase factor $e^{i\vec{k}.\vec{r}}$ and a periodic function. It is these k-points that we sample, approximating the integral in equation 2.2 by a summation over discrete k-vectors drawn uniformly from the first Brillouin zone.

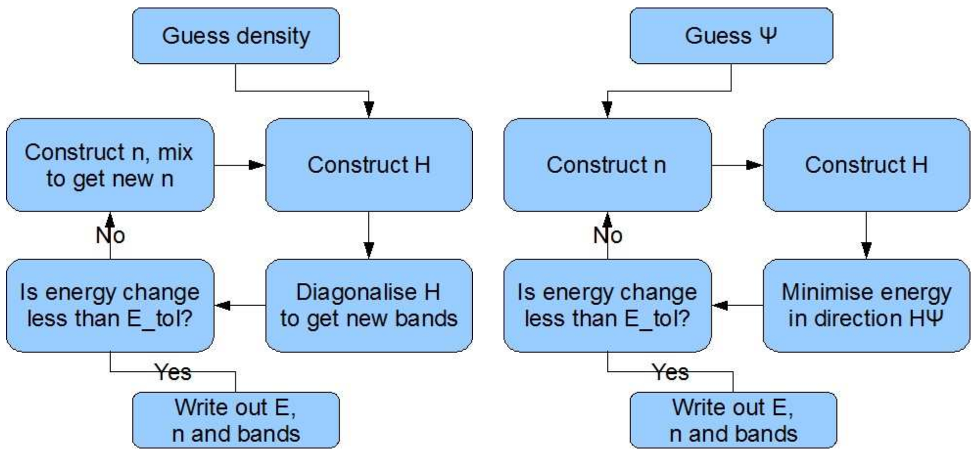
The Hamiltonian consists of some terms which are diagonal, or near-diagonal, in reciprocal space (e.g. the kinetic energy operator) and some terms which are diagonal in real-space (e.g. the ionic potential operator). This necessitates the use of Fourier transforms to transform the data from reciprocal- to real-space and back again.

Diagonalising the Hamiltonian directly is too expensive to be practical, and yields far more eigenstates than the lowest few we require; hence the search for the groundstate is an iterative procedure to compute just the lowest N_b eigenstates, and in Castep this search can proceed in one of two main modes: *density mixing (DM)* or *ensemble density functional theory (EDFT)*.

In the DM methodology the search proceeds by computing approximate eigenstates for a *fixed* trial charge density, computing a new density from these eigenstates, and then employing a density mixer to produce a new estimate for the true groundstate density.

In the EDFT method the density used is always that obtained from the Kohn-Sham wavefunctions. A trial step is performed along the search direction and the density and energy recomputed. Castep then uses the data from this sample point, along with the data at the initial point, to fit a quadratic and estimate the so-called self-consistent minimum step, and hence find a better trial wavefunction. For metallic systems the band-occupancies are determined in a similar manner, updating the density and energy at each trial step of an occupancy cycle and always aiming to take steps that lower the total energy.

Because the EDFT method requires the construction of the density many more times than the DM method, it performs many more Fourier transforms of the wavefunction and is significantly slower per SCF cycle. However by fixing the Hamiltonian in the DM wavefunction updates, the DM algorithm


Guess density
Construct n, mix to get new n
Construct H
No
Is energy change less than E_tol?
Diagonalise H to get new bands
Yes
Write out E, n and bands
Guess Ψ
Construct n
Construct H
No
Is energy change less than E_tol?
Minimise energy in direction HΨ
Yes
Write out E, n and bands

(a) Density mixing (DM) algorithm

(b) Ensemble density functional algorithm (EDFT)

Figure 2.1: Diagram showing the basic Castep self-consistent field (SCF) cycle for the density mixing (2.1(a)) and ensemble DFT (2.1(b)) algorithms

has a tendency to overshoot the true eigenstate and it is only the density mixer itself that drives the algorithm to the correct groundstate. For this reason it is common for DM calculations to take many more SCF cycles to converge than corresponding EDFT calculations, though each DM cycle is considerably faster.

2.1 Parallelisation

Since the eigenstates at different k-points are almost independent of each other, interacting only via the density, it is natural to distribute the data over the k-points. Unfortunately the number of k-points required decreases with increasing system size, and for large systems N_k is O(1). This means that on HPC machines it is rarely possible to distribute the data and workload efficiently using k-point parallelism alone.

A further distribution strategy used by Castep is to distribute the data by the Fourier components, i.e. the 'G-vectors'. Since the number of plane-waves is often large, and grows with simulation cell size, this enables efficient data distribution and load-balancing.

By combining both k- and G-parallelism Castep has demonstrated excellent scaling properties from 1 to O(100) nodes across many different computer architectures.

2.2 Computational Costs

2.2.1 Orthogonalisation and Diagonalisation

In both the DM and EDFT methods the dominant costs for large systems are the orthogonalisation of the trial wavefunctions to each other, and the subspace rotations required to diagonalise the Hamiltonian in the subspace of the current trial bands. Both of these operations scale as $N_G N_b^2 N_k$ for a system with N_G G-vectors, N_b bands and N_k k-points. As has been mentioned already, the number of k-points required decreases with system size, and for very large systems $N_k = 1$ so the asymptotic scaling is cubic.

The orthogonalisation proceeds by constructing the band-overlap matrix, performing a Cholesky decomposition of this matrix to determine an orthogonalising transformation, and then applying this transformation. Both the construction of the band-overlap matrix and the applications of the transform are distributed over G-vectors and k-points, with very little communication required. The Cholesky decomposition is performed on a single node per k-point and the result broadcast, so is not distributed over G-vectors.

The subspace diagonalisation also requires a band-overlap matrix to be constructed, though this time it is between bands drawn from two different wavefunctions (ψ and $\hat{H}\psi$). This matrix is then diagonalised to determine a diagonalising rotation, and the rotation applied to both the input wavefunctions. The first and last stages are distributed naturally over G- and k-points, but the diagonalisation itself is performed on a single node per k-point.

2.2.2 Fourier Transforms

The Fourier Transforms are carried out using conventional Fast Fourier Transform (FFT) libraries, or a built-in FFT written by Clive Temperton according to his Generalised Prime Factor Algorithm (GPFA). Since the data are distributed over G-vectors, Castep transforms the data not as a single 3D FFT but as three 1D FFTs, with a transpose step in between each to reorder the data. Although the cost of the FFTs is only $N_G ln N_G N_b N_k$, this transpose requires all-to-all communications between all processor elements with the same k-points. The data and workload is naturally distributed over k-points, but as the G-vectors are distributed over more and more nodes these all-to-all communications start to dominate.

The length of the FFTs in Castep is typically small, and even for large simulation cells it will usually be O(100). However every band at every k-point requires three 1D FFTs every time the potential is applied or the density constructed so the total number of FFTs required in a simulation can be extremely large. The vast number of messages coupled with their short length means that in Castep the communication stages are usually latency-bound, rather than bandwidth-bound.

In recent years Castep's communication in the transpose stage has been optimised by Martin Plummer (STFC) and Keith Refson (RAL) to allow a two-phase communication pattern. In the first phase PEs within a node exchange data, and then in the second phase the all-to-all communication is performed over the *nodes*, not the individual *PEs*. For small numbers of nodes the overhead of performing two communication phases is prohibitive, but as the all-to-all communications start to dominate the improved scaling of this method more than makes up for the additional cost. This optimisation was of great benefit on HPCx, which has 16 cores per node, but is still expected to be useful on HECToR for large numbers of nodes.

Chapter 3

Castep Performance on HECToR (Work Package 0)

3.1 General Castep Performance

As was noted in the previous chapter, Castep's performance is usually limited by two things: orthogonalisation-like operations, and FFTs. The orthogonalisation (and subspace diagonalisation) are performed using standard BLAS and LAPACK subroutine calls, such as those provided on HECToR by the ACML or Cray's LibSci. Castep has a built-in FFT algorithm for portability, but it is not competitive with tuned FFT libraries such as FFTW and provides interfaces to both FFT versions 2 and 3. ACML also provides FFT subroutines.

Castep is written entirely in Fortran 90, and HECToR has three Fortran 90 compilers available: Portland Group (pgf90), Pathscale (pathf90) and GNU's gfortran. Following the benchmarking carried out during the procurement exercise, it was anticipated that Pathscale's pathf90 compiler would be the compiler of choice and Alan Simpson (EPCC) was kind enough to provide his flags for the Pathscale compiler, based on the ones Cray used in the procurement:

```
-O3 -OPT:Ofast -OPT:recip=ON -OPT:malloc_algorithm=1
-inline -INLINE:preempt=ON
```

Note that this switches on `fast-math`.

Unless otherwise noted, all program development and benchmarking was performed with the Castep 4.2 codebase, as shipped to the United Kingdom Car-Parinello (UKCP) consortium, which was the most recent release of Castep at the commencement of this dCSE project and was the version available on HECToR to end-users.

3.2 Benchmarks

The standard Castep benchmarks have not changed for several years, and many are now too small to be useful for parallel scaling tests. The smallest benchmark, al1x1, is a small slab of aluminium oxide and runs in less that 6 minutes on 8 PEs of HECToR. The larger titanium nitride benchmark, TiN (which also contains a single hydrogen atom), only takes an hour on 16 PEs because the DM algorithm converges slowly – its time per SCF cycle is little more than twice the al1x1 benchmark. For this reason we settled on the al3x3 test system as the main benchmark for the parallel scaling tests, since this was large enough to take a reasonable amount of time per SCF cycle, yet small enough to run over a wide range of nodes.

The al3x3 benchmark is essentially a 3x3 surface cell of the al1x1 system, and has:

- 270 atoms (108 Al, 162 O)
- 88,184 G-vectors
- 778 bands (1296 electrons, non-spin-polarised, plus 130 conduction bands)
- 2 k-points (symmetrised 2×2×1 MP grid)

However the parameter files for this calculation do *not* specify Castep's optimisation level. In general it is advisable to tell Castep how to bias it's optimisation, e.g. `opt_strategy_bias : 3` to optimise for speed (at the expense of using more RAM). Since the default optimisation level is not appropriate for HPC machines such as HECToR, most of our calculations were performed with the addition of `opt_strategy_bias : 3` to the Castep parameter file `al3x3.param`.

3.2.1 FFT

The FFTW version 2 libraries on HECToR are only available for the Portland Group compiler, so the full FFT comparison tests were performed exclusively with pgf90. Cray's LibSci (10.2.1) was used for BLAS and LAPACK. The following compiler flags were used throughout the FFT tests:

```
-fastsse -O3 -Mipa
```

In order to measure the performance of the FFT routines specifically we used Castep's internal Trace module to profile the two subroutines `wave_recip_to_real_slice` and `wave_real_to_recip_slice`. These subroutines take a group of eigenstates, called a wavefunction slice, and Fourier transform them from reciprocal space to real space, or vice versa.

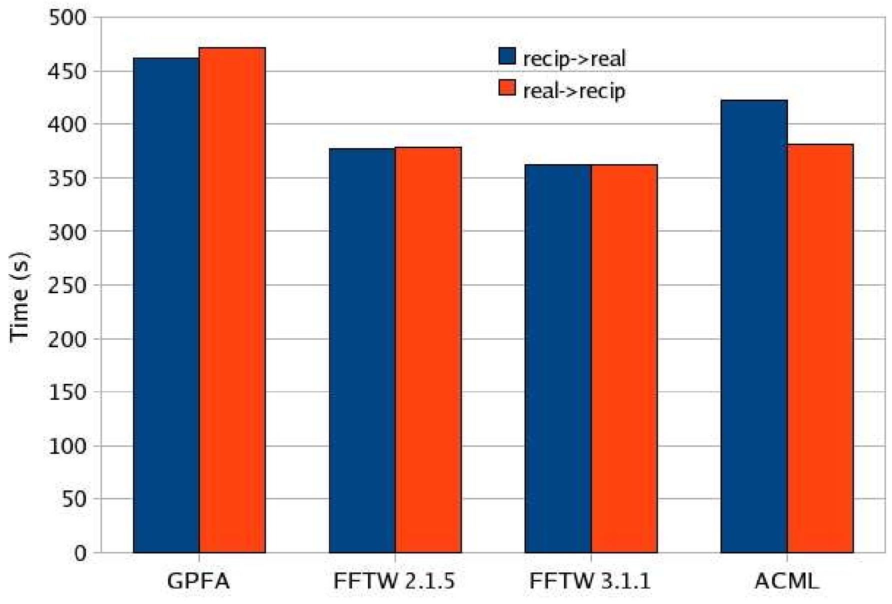


Figure 3.1: Graph showing the relative performance of the four FFT subroutines available to Castep on HECToR for the TiN benchmark. This benchmark transforms wavefunction slices to real space 1634 times and back again.

As can be seen from figure 3.1 FFTW 3.1.1 was the fastest FFT library available on HECToR.

3.2.2 Maths Libraries (BLAS)

Much of the time spent in Castep is in the double-precision complex matrix-matrix multiplication subroutine ZGEMM. The orthogonalisation and subspace rotation operations both use ZGEMM to apply unitary transformations to the wavefunctions, and it is also used extensively when computing and applying the so-called non-local projectors. Although the unitary transformations dominate the asymptotic cost of large calculations, the requirement that benchmarks run in a reasonable amount of time means that they are rarely in this rotation-dominated regime. The orthogonalisation and diagonalisation subroutines also include a reasonable amount of extra work, including a memory copy and updating of meta-data, which can distort the timings for small systems. For these reasons we chose to concentrate on the timings for the non-local projector overlaps as a measure of ZGEMM performance, in particular the subroutine ion_beta_add_multi_recip_all which is almost exclusively a ZGEMM operation.

For the BLAS tests, the Pathscale compiler (version 3.0) was used throughout with the compiler options:

```
-O3 -OPT:Ofast -OPT:recip=ON -OPT:malloc_algorithm=1 -inline
-INLINE:preempt=ON
```

As can be seen from figure 3.2 Cray's LibSci 10.2.1 was by far the fastest BLAS library available on HECToR, at least for ZGEMM.

3.2.3 Compiler

Much of the computational effort in a Castep calculation takes place in the FFT or maths libraries, but there are still significant parts of the code for which no standard library exists. It is the performance of these parts of code that changes depending on the compiler used and the various flags associated with it.

Unfortunately GNU's gfortran compiler (4.2.4) is not capable of correctly compiling the Castep 4.2 codebase, so our investigations on HECToR were restricted to the Portland Group and Pathscale compilers. It should be noted that versions 4.3.0 and later can compile Castep, but are not yet available on HECToR.

For the Pathscale compiler (3.0) we used the flags provided by Alan Simpson (EPCC) as a base for our investigations,

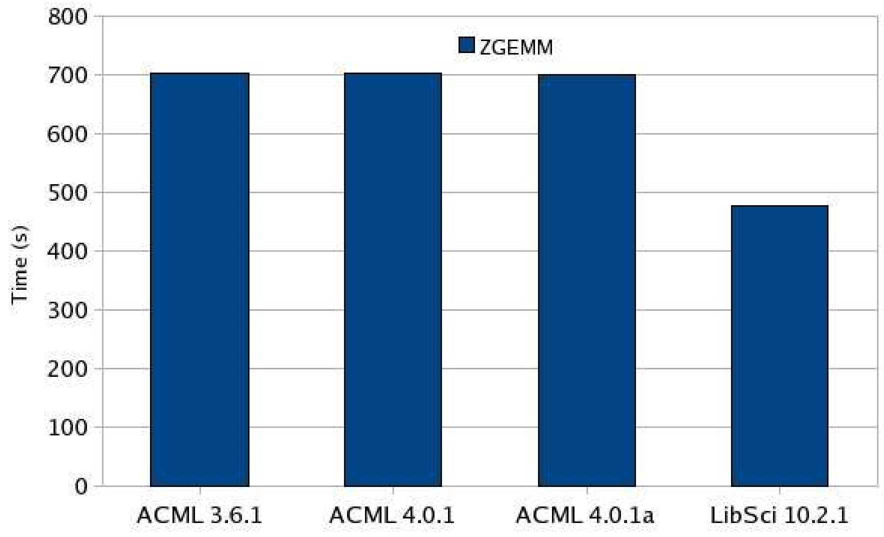


Figure 3.2: Graph showing the relative performance of the ZGEMM provided by the four maths libraries available to Castep on HECToR for the TiN benchmark. This benchmark performs 4980 projector-projector overlaps using ZGEMM. Castep's internal Trace module was used to report the timings.

```
-O3 -OPT:Ofast -OPT:recip=ON -OPT:malloc_algorithm=1
-inline -INLINE:preempt=ON
```

and created six compilation flagsets. The first set, which was used as a base for all the other sets, just used `-O3 -OPT:Ofast`, and we named this the `bare` set. The other five used this, plus:

```
malloc_inline        -OPT:malloc_algorithm=1 -inline -INLINE:preempt=ON
recip                -OPT:recip=ON
recip_malloc         -OPT:recip=ON -OPT:malloc_algorithm=1
recip_malloc_inline  -OPT:recip=ON -OPT:malloc_algorithm=1 -inline
                     -INLINE:preempt=ON
full                 -OPT:recip=ON -OPT:malloc_algorithm=1 -inline
                     -INLINE:preempt=ON -march=auto -m64 -msse3 -LNO:simd=2
```

The performance of the various Castep binaries can be seen in figure 3.3. It is clear that the flags we were given by Alan Simpson are indeed the best of this set.

For the Portland Group compiler we used the base flags from the standard Castep pgf90 build as a starting point, `-fastsse -O3`. Unfortunately there seemed to be a problem with the timing routine used in Castep when compiled with `pgf90`, as the timings often gave numbers that were far too small and did not tally with the actual walltime. Indeed the Castep output showed that the SCF times were 'wrapping round' during a run, as in this sample output from an al3x3 benchmark:

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial  -5.94087234E+004  5.75816046E+001                        71.40  <-- SCF
      1  -7.38921628E+004  4.31787037E+000   5.36423678E+001     399.29  <-- SCF
      2  -7.78877742E+004  1.96972918E+000   1.47985607E+001     689.06  <-- SCF
      3  -7.79878794E+004  1.79936064E+000   3.70760070E-001     954.04  <-- SCF
      4  -7.78423468E+004  1.96558259E+000  -5.39009549E-001    1250.05  <-- SCF
      5  -7.77212605E+004  1.34967844E+000  -4.48467894E-001    1544.50  <-- SCF
      6  -7.77152926E+004  1.12424610E+000  -2.21032775E-002    1863.09  <-- SCF
      7  -7.77129468E+004  1.05359411E+000  -8.68814103E-003      14.53  <-- SCF
      8  -7.77104895E+004  1.02771272E+000  -9.10094481E-003     288.19  <-- SCF
      9  -7.77084348E+004  9.96278161E-001  -7.60993336E-003     582.43  <-- SCF
     10  -7.77059813E+004  1.11167947E+000  -9.08729795E-003     872.09  <-- SCF
     11  -7.77052050E+004  1.16249354E+000  -2.87513162E-003    1162.86  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

This behaviour has not been seen on other machines to our knowledge, was not reproduced on HECToR with the Pathscale compiler, where the Castep timings agreed with the walltime reported in the PBS output file to within a second. Unfortunately this behaviour meant that we were forced to rely on the PBS output file for the total walltime for each run, which includes set-up and finalisation time that we would have liked to omit.

We experimented with various flags to invoke interprocedural optimisation `-Mipa`, `-Mipa=fast` but the Castep timings remained constant to within one second. Figure 3.4 shows the run times of both the Portland Group and Pathscale compiler as reported by the PBS output for the TiN benchmark.

3.2.4 Node Usage

Each node on HECToR has two cores, or PEs, so we ran a series of Castep calculations to see how the performance and scaling of a Castep calculation depends on the number of PEs used per node. We also used these calculations to double-check the results of our investigation into different libraries. The results are shown in figure 3.5.

The best performance was achieved with the Goto BLAS in Cray's libsci version 10.2.0 coupled with the FFTW3 library, as can be seen in figure 3.5.

The best performance *per core* was achieved using only one core per node, though the performance improvement over using both cores was not sufficient to justify the expense (since jobs are charged *per node* not *per core*). Using Castep's facility for optimising communications within an SMP node[1] the scaling was improved dramatically and rivals that of the one core per node runs.

It may be possible to run one MPI thread per core for the main Castep calculation and use the other core for threaded BLAS, but this would be a major project in itself and we do not propose to undertake it in this work.

3.3 Baseline

We decided to choose the Pathscale 3.0 binary, compiled with the recip_malloc_inline flags (see section 3.2.3) and linked against Cray's Libsci 10.2.1[2] and FFTW3 for our baseline, as this seemed to offer the best performance with the 4.2 Castep codebase.

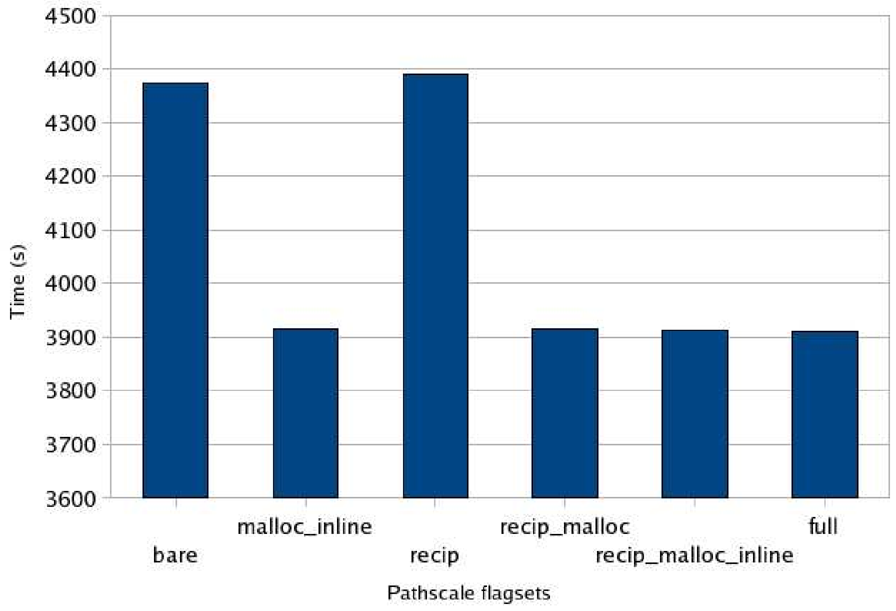
3.4 Analysis

Both the Cray PAT and built-in Castep trace showed that a considerable amount of the time in ZGEMM, as well as the non-library time, was spent in nlpot_apply_precon. The non-library time was attributable to a packing routine which takes the unpacked array beta_phi, which contains the projections of the wavefunction bands onto the nonlocal pseudopotential projectors, and packs them into a temporary array. Unfortunately this operation was poorly written, and the innermost loop was over the slowest index.

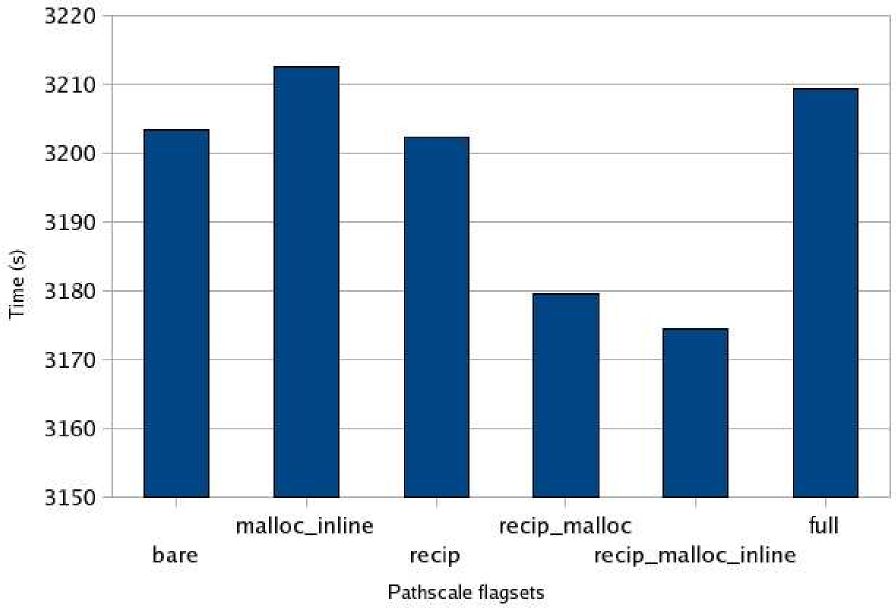
The ZGEMM time in nlpot_apply_precon could also be reduced because the first matrix in the multiplication was in fact Hermitian, so the call could be replaced by ZHEMM to do approximately half the work.

[1]Using the `num_proc_in_smp` and `num_proc_in_smp_fine` parameters

[2]In fact this library was only made available part-way through this project; the benchmark results were updated later to reflect this.



(a) TiN benchmark (16 PEs)



(b) al3x3 benchmark (32 PEs)

Figure 3.3: Comparison of Castep performance for the Pathscale compiler with various flags, using 39 SCF cycles of the TiN benchmark (3.3(a)) and 11 SCF cycles of the al3x3 benchmark (3.3(b)).

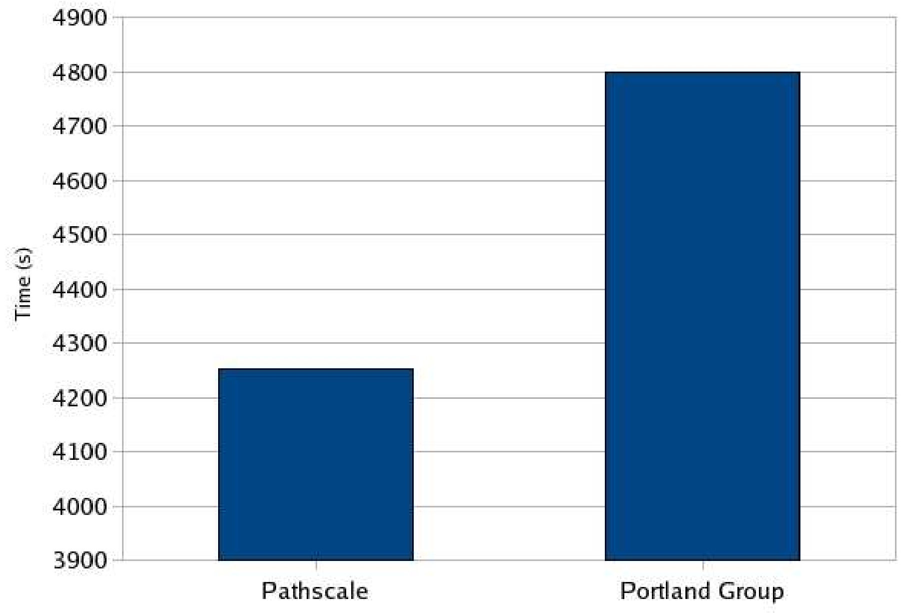
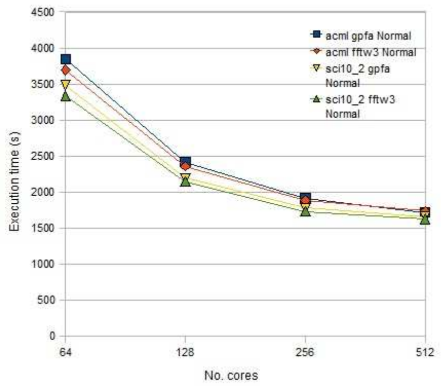
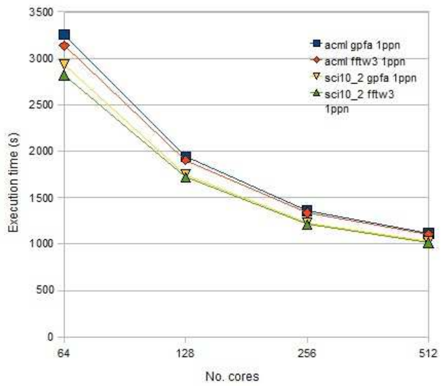


Figure 3.4: Graph showing the relative performance of the Pathscale 3.0 compiler (recip_malloc_inline flags) with the Portland Group 7.1.4 compiler (-fastsse -O3 -Mipa) for the TiN benchmark on 16 PEs. The PBS reported walltime was used to report the timings.



(a) Execution time using 2 cores per node



(b) Execution time using 1 core per node

Figure 3.5: Comparison of Castep performance for the ACML and LibSci (Goto) BLAS libraries, and the generic GPFA and FFTW3 FFT libraries, run using two cores per node (3.5(a)) and one core per node (3.5(b))

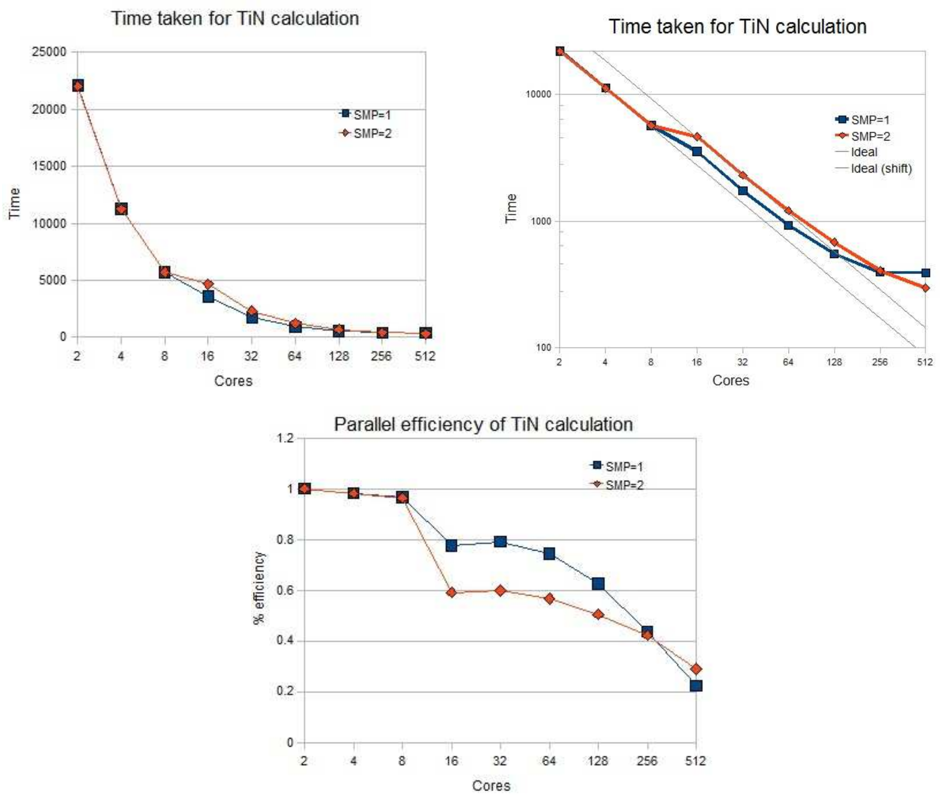
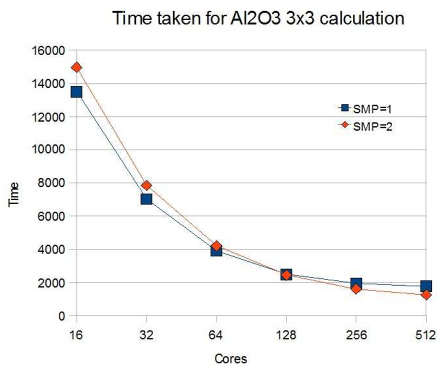
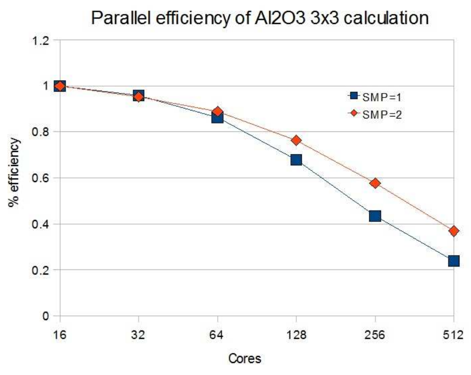


Figure 3.6: Execution time for the 33 atom TiN benchmark. This calculation is performed at 8 k-points.

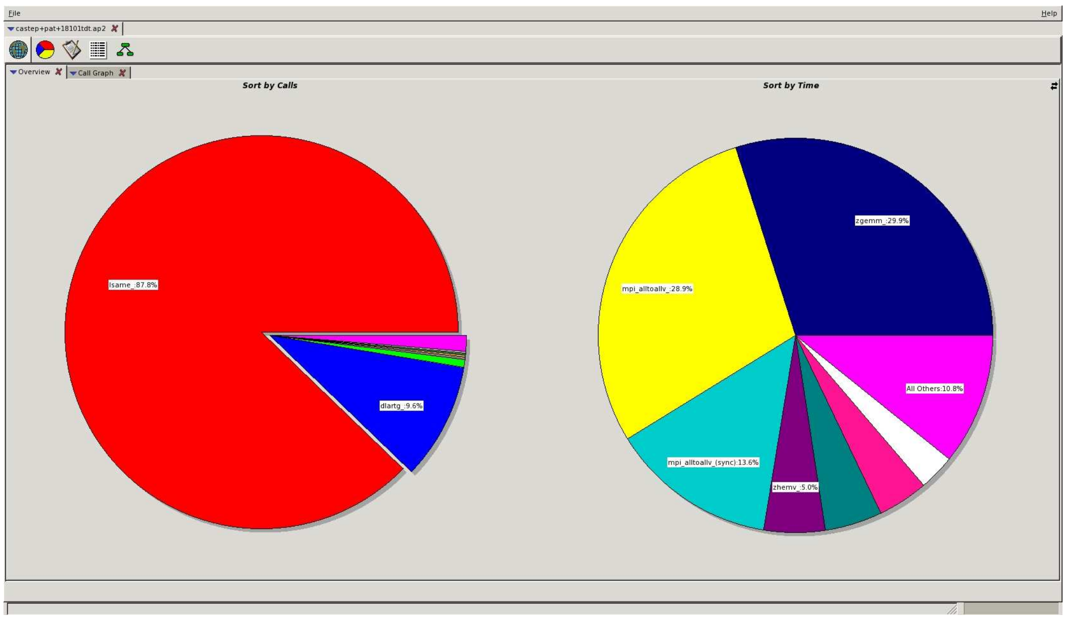


(a) Execution time

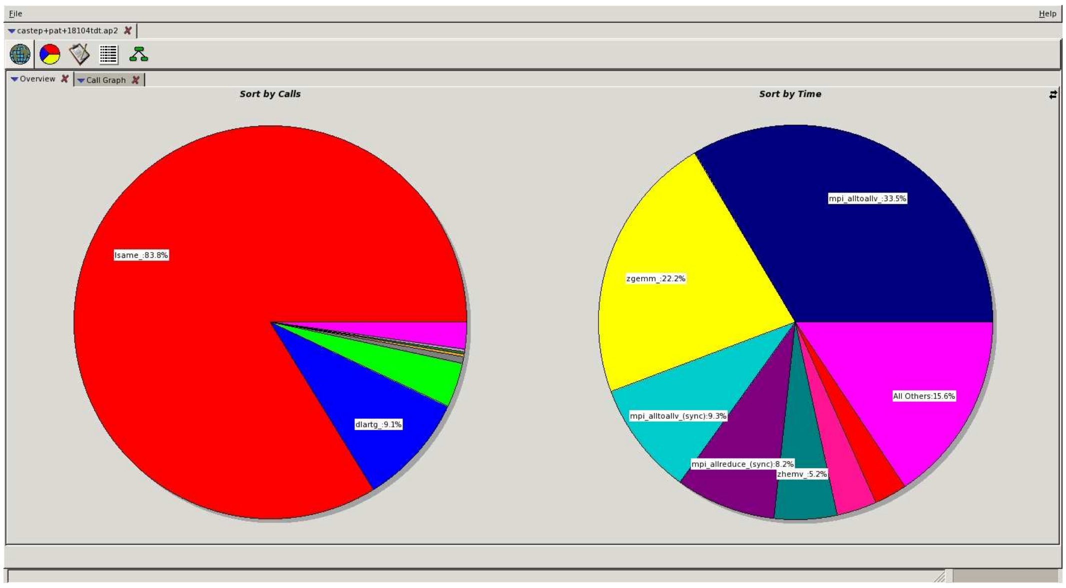


(b) Efficiency with respect to 16 cores

Figure 3.7: Scaling of execution time with cores for the 270 atom Al2O3 3x3 benchmark. This calculation is performed at 2 k-points.

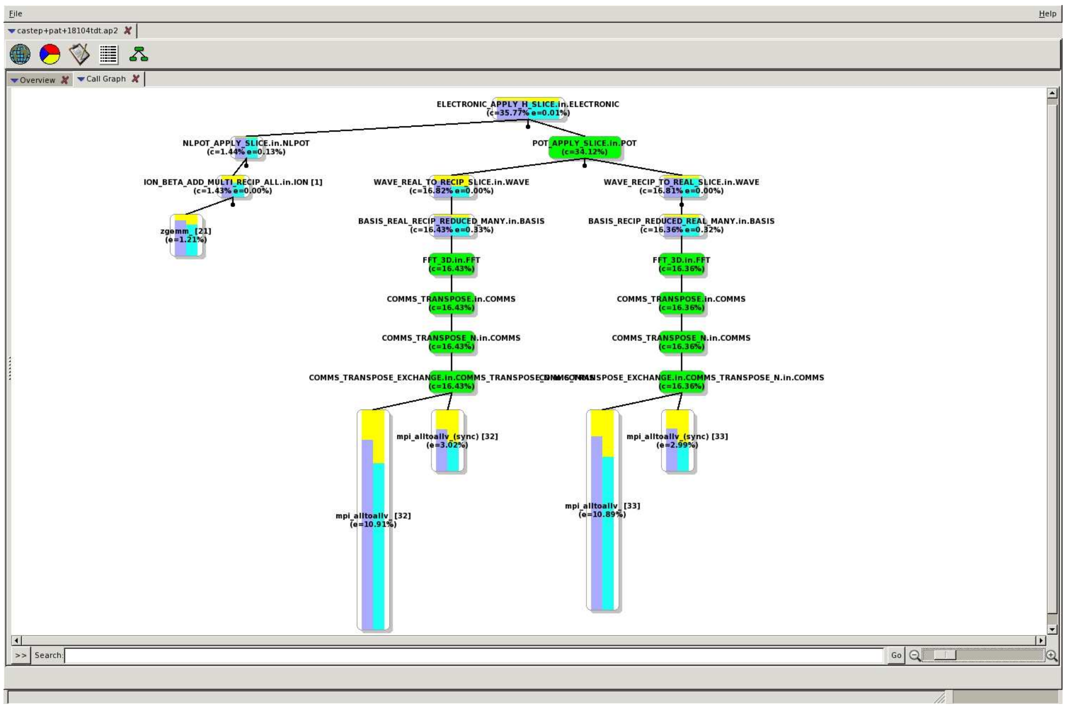


(a) CPU time for Castep on 256 cores

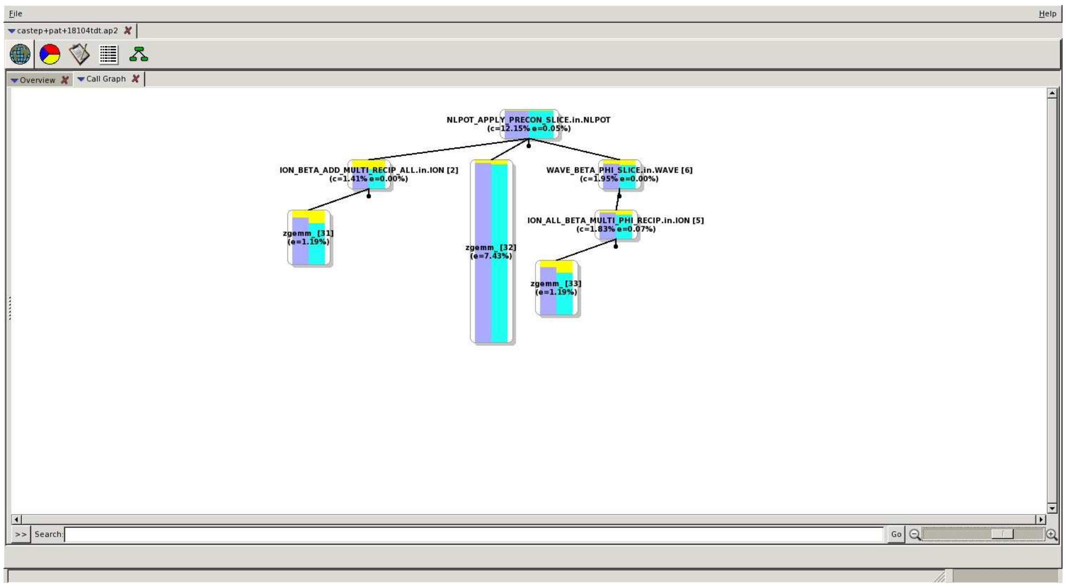


(b) CPU time for Castep on 512 cores

Figure 3.8: Breakdown of CPU time for 256 (3.8(a)) and 512 (3.8(b)) cores using 2 ppn, for Castep Al2O3 3x3 benchmark


File
Help
castep+pat+18104tdt.ap2
Overview
Call Graph
ELECTRONIC_APPLY_H_SLICE.in.ELECTRONIC (c=35.77% e=0.01%)
NLPOT_APPLY_SLICE.in.NLPOT (c=1.44% e=0.13%)
POT_APPLY_SLICE.in.POT (c=34.12%)
ION_BETA_ADD_MULTI_RECIP_ALL.in.ION [1] (c=1.43% e=0.00%)
WAVE_REAL_TO_RECIP_SLICE.in.WAVE (c=16.82% e=0.00%)
WAVE_RECIP_TO_REAL_SLICE.in.WAVE (c=16.81% e=0.00%)
zgemm_ [21] (e=1.21%)
BASIS_REAL_RECIP_REDUCED_MANY.in.BASIS (c=16.43% e=0.33%)
BASIS_RECIP_REDUCED_REAL_MANY.in.BASIS (c=16.36% e=0.32%)
FFT_3D.in.FFT (c=16.43%)
FFT_3D.in.FFT (c=16.36%)
COMMS_TRANSPOSE.in.COMMS (c=16.43%)
COMMS_TRANSPOSE.in.COMMS (c=16.36%)
COMMS_TRANSPOSE_N.in.COMMS (c=16.43%)
COMMS_TRANSPOSE_N.in.COMMS (c=16.36%)
COMMS_TRANSPOSE_EXCHANGE.in.COMMS_TRANSPOSE_N.in.COMMS (c=16.43%)
COMMS_TRANSPOSE_EXCHANGE.in.COMMS_TRANSPOSE_N.in.COMMS (c=16.36%)
mpi_alltoallv_(sync) [32] (e=3.02%)
mpi_alltoallv_(sync) [33] (e=2.99%)
mpi_alltoallv_ [32] (e=10.91%)
mpi_alltoallv_ [33] (e=10.89%)
Search:
Go

(a) CPU time spent applying the Hamiltonian in Castep


File
Help
castep+pat+18104tdt.ap2
Overview
Call Graph
NLPOT_APPLY_PRECON_SLICE.in.NLPOT (c=12.15% e=0.05%)
ION_BETA_ADD_MULTI_RECIP_ALL.in.ION [2] (c=1.41% e=0.00%)
WAVE_BETA_PHI_SLICE.in.WAVE [6] (c=1.95% e=0.00%)
ION_ALL_BETA_MULTI_PHI_RECIP.in.ION [5] (c=1.83% e=0.07%)
zgemm_ [31] (e=1.19%)
zgemm_ [32] (e=7.43%)
zgemm_ [33] (e=1.19%)
Search:
Go

(b) CPU time spent preconditioning the search direction in Castep

Figure 3.9: The CPU time spent in the two dominant user-level subroutines and their children, for a 512-core (2 ppn) Castep calculation of the Al2O3 3x3 benchmark

Chapter 4

Band-Parallelism (Work Package 1)

4.1 Introduction

The first stage of the project is to implement the basic band-parallelism. This involves distributing the wavefunction coefficients over the N_b bands, in addition to the N_G G-vectors and N_k k-points, as well as distributing other data such as the eigenvalues, band occupancies, and band-overlap matrices such as the subspace Hamiltonian. Along with the data distribution, as much of the workload as possible should also be distributed over the bands.

4.2 Programming

The main programming effort lies in the basic band-parallelism, and in particular the construction of band-overlap matrices which will now need to be distributed. Most of this effort will be concentrated in the `wave` module, which handles the vast majority of operations on wavefunctions and bands, but inevitably there are some changes required to other modules.

4.2.1 Comms

The `comms` module handles all of the communications in Castep, and comes in two flavours at the moment: serial and MPI. The work in comms boils down to:

- Create a new communicator for the new 'band-groups'.
- Create new comms_reduce and comms_copy routines to support reduction/copying of data over the band-groups.

In particular the subroutines comms_parallel_strategy and comms_reassign_strategy now need to know how many bands to parallelise over, and so has gained an additional mandatory argument 'nbands'. This has necessitated trivial changes to model.F90 and castep.f90.

Work completed, tested and working.

4.2.2 Ion

In the density_augment subroutines, the augmentation charge and spin densities now to be reduced over the band-group.

During later testing, it was discovered that a large proportion of both the memory and computational time of Castep calculations were spent in the subroutine `ion_beta_beta_recip`. The reason for both the memory and time cost of this operation is that this subroutine has to construct a modified overlap matrix between the so-called non-local projectors, often referred to as the β-projectors. These projectors are arrays of plane-wave coefficients, just like a wavefunction, but because they are independent of the bands they are only distributed by plane-wave and k-point. The precise operation this subroutine performs is the construction of the matrix B:

$$B_{ij} = \sum_{p=1}^{N_p} \beta^*_{pi} K_{pp} \beta_{pj} \tag{4.1}$$

where K_{pp} is a diagonal positive-definite matrix used for preconditioning. The problem with memory is that in order to exploit the optimised BLAS most efficiently, this is converted to

$$B_{ij} = \sum_{p=1}^{N_p} \gamma^*_{pi} \gamma_{pj} \tag{4.2}$$

where $\gamma_{pi} = \sqrt{K_{pp}}\beta_{pi}$. This allows the use of the BLAS subroutine `ZHERK`, but at the cost of an extra copy of the non-local projectors.

Our solution to this problem was to distribute the γ-projectors over the band-group. In the first phase the local γ-projectors are constructed, and a call to ZHERK computes the purely local contribution to the B matrix. The second phase requires a computation of the local projectors with the projectors on the other nodes via a call to ZGEMM. Rather than get the relevant data via comms, we instead redefined the γ-projectors at this point so that they now contained the full effect of the diagonal matrix K, i.e. $\gamma_{pi} = K_{pp}\beta_{pi}$. This allows us to compute the second contribution to the B by simply computing the overlap between the local node's γ-projectors and the non-distributed β-projectors.

Work completed, tested and working.

Optimisation

Clearly a better solution to the problem of the projector-overlaps in the Ion module would be to distribute the β-projectors as well as the γ-projectors. However the β-projectors are used extensively throughout the Ion module, including computing overlaps between wavefunctions and the projectors which can be done without communication in the current code. Distributing the β-projectors would thus require extensive modifications and an extra communication overhead, whereas the changes described above were localised to the problematic subroutine.

4.2.3 Wave

The wavefunction module needs extensive modification. It is this module which handles the actual distribution of data.

- Data distribution and associated meta-data.
- Band-overlap and dot-product matrices
- (S-)orthogonalisation

Band-overlaps and dot-product

The (S-)orthonormalisation and subspace diagonalisation operations both require dot products between all pairs of bands. In the case of the (S-)orthonormalisation the overlaps are between all pairs of bands of a wavefunction, and so the result is Hermitian. For the subspace diagonalisation the dot-product is between each band of a wavefunction with each band of a different wavefunction – in fact since the second wavefunction is the result of a Hermitian operator applied to the first wavefunction the result should also be Hermitian, though at the moment we do not exploit this[1].

We distribute this by passing our data n places to the right, and receiving data from n places to the left, and dotting the received data with our local data. Then we increment n by 1 and repeat the process until $n \geq N_b/2$.

Rotations

From time to time the wavefunctions need to be rotated by a band $\times$ band matrix M, e.g.

$$\psi'_{pb} = \sum_k \psi_{pk} M_{kb} \tag{4.3}$$

Clearly this transformation, in general, mixes data between all of the nodes. Rather than storing an entire wavefunction we transform just the parts needed for one node at a time.

```
! construct the data for node ‘node’
do node=1,bnd_nodes
  transform local data to give contribution to ‘node’
  reduce over bnd-group
end do
```

This is simplest if we encapsulate all such operations into a single subroutine, overloaded for the various wavefunction types. In fact such a routine already exists, `wave_rotate`.

There are also occasions when we wish to transform a wavefunction by a matrix that is not square, so that the resulting number of bands is not the same as the initial number of bands. This occurs when, for example, we orthogonalise a slice to an existing wavefunction; we copy the wavefunction to a temporary slice, transform the data by multiplying by the overlap matrix, which changes the number of bands to the same as the slice, and then subtract it from slice. This is not a ‘rotation’, and goes against the specification for `wave_rotate`. For this reason we coded a new, more general subroutine `wave_transform` which allows *any* linear transformation of the bands.

The orthogonalisation subroutines can now be coded compactly as:

[1]In fact the bands-parallel scheme makes it *easier* to exploit this property, should we wish to do so

```
                                node 1   node 2   node 3   node4
  o-> wave_rotate_slice        289.97s  296.95s  305.35s  310.88s
  o-> wave_nullify_slice         0.00s    0.00s    0.00s    0.00s
  o-> comms_reduce_bnd_integer  19.25s   13.22s    3.73s    0.04s
  o-> wave_allocate_slice        0.01s    0.03s    0.02s    0.01s
  o-> wave_initialise_slice      6.73s    6.67s    7.49s    7.76s
  o-> comms_recv_integer         0.06s   45.13s   30.15s   10.71s
  o-> comms_send_integer        45.14s    0.09s    0.10s    0.07s
  o-> comms_recv_complex        61.95s   15.96s   32.17s   29.83s
  o-> comms_send_complex        15.64s   72.11s   72.45s  100.96s
  o-> wave_copy_slice_slice      5.46s    6.30s   11.62s    7.58s
  o-> wave_deallocate_slice      0.02s    0.00s    0.02s    0.00s
```

Table 4.1: Table showing the timings for the wave_rotate_slice operation on each of the nodes in a 4-node calculation

```
! orthogonalise slice to all bands of wvfn
call wave_dot_all(wvfn,slice,overlap)

call wave_copy(wvfn,tmp_slice)
call wave_transform(tmp_slice,overlap)

! Subtract overlapping components
call wave_add(tmp_slice,slice,c1=-cmplx_1)
```

Communication pattern

The scheme outlined above works well for calculations over 2 or 3 nodes, but as the number of nodes increases the time taken by the communication calls also increases. If we look at the profiles from Castep's internal trace for `wave_rotate_slice` on a 4-node calculation (table 4.1) we can see that the higher-numbered nodes are spending more time in the subroutine than the lower-numbered ones.

The reason for this disparity is that by constructing all of the data for a given node before moving onto the next node, we 'serialise' the communication calls–each communication phase is one-to-many or many-to-one, and because Castep uses standard MPI_send and _recv calls every node has to wait for all of the preceding nodes to finish their communications before it can begin. This process proved to be a severe bottleneck for calculations over large numbers of nodes.

Removing this bottleneck is straightforward–we simply copy the communication pattern from the dot-all subroutines. The communication pattern is now a cyclic one, where every node constructs the data for a node n hops prior to it. For each n there are two sets of communications, each of two phases. The first communication set consists of each node sending its rotation matrix n places to the left, and receiving a rotation matrix from n places to the right (there is also some exchange of meta-data). Each node then applies the rotation matrix it received to its local data. In the second communication phase each node passes the result of its rotation n places to the right, and receives the contribution to its own transformed data from n places to the left.

All of the communication phases are now point-to-point, and many such communications can take place simultaneously.

Optimisation Note

The above changes assume a general transformation or rotation is required. As the calculation proceeds, the approximate eigenstates should converge to the true eigenstates and so many of the transformations will approach the identity. It may be possible to exploit this anticipated sparsity to optimise the communication pattern.

wave_(S)dot_lower

This is like wave_(S)dot_all but only takes the dot-product for bands lower than the band with index nb. This argument refers to the *global* band index, and so must be translated into a local value on each node.

There is a problem with the existing mechanism, because it call the dot_all_many_one version of the dot_all routines, and passes it the slice recip_coeffs(:,1:nb-1) and recip_coeffs(:,nb); now, however, band nb may or may not be on the node and so one or both of these may be null.

In addition wave_(S)dot_lower_slice_slice on different nodes may have different numbers of bands in the second slice and a different band to orthogonalise below as well.

We created dedicated subroutines to perform these operations, rather than attempting to reuse existing subroutines which were not designed for these circumstances.

Type issues

We assume wavefunctions are always distributed, and bands are never distributed, but groups of bands, called ‘slices’, are sometimes used as local store (e.g. slice, slice_direction etc. in electronic) and sometimes global store (e.g. S_wvfn_slice in electronic). Thus we add a new logical bands_distributed to the wavefunction_slice type, set to ‘.false.’ by default.

Work completed, tested and working.

4.2.4 wave_write

Castep writes its wavefunctions in a distribution-independent manner. The basic scheme, updated for band-parallelism, is as follows:

- All of the I/O is done by the root-node
- The root node is a kp-master, bnd-master and gv-master
- The root node loops over all nodes that are both bnd- *and* gv-masters, and receives their data
- Nodes that are both bnd- and gv-masters assemble the gv-distributed data from across their gv-group (i.e. the group of bnd-masters) and send the data to the root node, getting it where necessary from the appropriate node in their bnd-group
- Nodes that are bnd-masters send the appropriate band data to their gv-master, if necessary getting it from the appropriate node in their band-group.
- General nodes send their local data to their band-master

Work completed, tested and working.

Band distribution

The initial implementation of the band-parallelism gave each node a contiguous block of bands. Whilst this allowed greater optimisation on each node, the load balancing was problematic. The two greatest problems for load-balancing were:

- Construction of the charge density
  The construction of the valence charge density can be a time-consuming operation, but only occupied bands contribute. If bands are distributed in contiguous blocks, most nodes will contain only occupied bands, but a small number of nodes will have a much larger proportion of conduction bands–indeed some nodes might have no occupied bands

- EDFT unoccupied bands update
  The EDFT algorithm requires the unoccupied bands to be optimised separately from the occupied bands. If the bands are assigned to nodes in contiguous blocks, only a small number of nodes will have all of the conduction bands and the rest will be idle in this phase.

There is a further issue with a distribution by contiguous blocks of bands that is not related to load-balancing: traditional optimisation methods are not band-local. This means that the nodes cannot optimise their own local set of bands independently of the other nodes–in particular, higher bands (i.e. more energetic bands) should not be optimised until the lower bands have converged.

For all of these reasons, the decision was made to switch the band-distribution to a round-robin scheme, whereby each of n nodes gets every nth band. This improves the load-balancing greatly, and also allows the existing optimisation algorithm to be used with few changes. This distribution is not 'hard-wired', and it is trivial to change to a different scheme.

It should be noted that the proposed optimisation scheme in Work Package 3 (see chapter 6) of this project *is* band-local, and the detail of the band-distribution may be revisited in that stage of the project.

Known issues remaining

The wave_reassign and wave_read subroutines have *not* yet been updated for the new parallelisation, neither has the wave_apply_symmetry subroutine.

4.2.5 Density

In the density_calculate subroutines, the density needs now to be reduced over the band-group.

Work completed, tested and working.

4.2.6 Ewald

Already parallelised over all active nodes, so just needed to add an extra reduction over the band-group.

Work completed, tested and working.

4.2.7 Electronic

The following needed to be addressed:

- Eigenvalue sums are now reduced over the band-group as well as the k-point group
- The Fermi level search and entropy correction are parallelised over the band-group as well as the k-point group
- The eigenvalue finder was parallelised over the band-group

Also needed to change electronic_analyse_occupancies to map the local band index to a global one, and then correctly determine the *global* highest occupied and lowest unoccupied states.

Completed, tested and working.

4.2.8 Model

Trivial changes needed to handle change in comms_parallel_strategy argument list, which now needs to know the number of bands as well as k-points.

4.2.9 Secondd

Trivial changes needed to handle change in comms_reassign_strategy argument list, which now needs to know the number of bands as well as k-points.

4.2.10 Phonon

Trivial changes needed to handle change in comms_reassign_strategy argument list, which now needs to know the number of bands as well as k-points.

4.3 Testing

The changes to many of the Castep modules were trivial, and needed little testing. The basic splitting of the MPI communicator to create a band-group was tested with a modified version of Castep's `comms_test` program.

The changes to the `wave` and `electronic` modules were rather more extensive, and difficult to test in isolation. In order to test and debug these modules, a short Castep calculation was run in serial to create a wavefunction and density in a Castep `.check` file, and then the calculation was restarted from that checkpoint file in serial and band-parallel modes. Because both jobs started from the same known point the calculations could be compared in detail, right down to individual wavefunction coefficients where necessary. This, coupled with Castep's in-built trace functionality, enabled bugs to be found and fixed quickly.

The ability to compare data in detail, even mid-calculation, proved necessary, since several bugs were found and fixed that did not affect the final, converged result but either hindered convergence, or could give rise to incorrect answers in unusual circumstances.

For a small 8-atom silicon test calculation performed using the density mixing (DM) method, the serial calculation produces

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial   5.10476316E+002  4.62264099E+001                        15.74  <-- SCF
      1  -7.76802126E+002  2.64224391E+000   1.60909805E+002      25.28  <-- SCF
      2  -8.50574887E+002  2.02770490E-001   9.22159500E+000      33.17  <-- SCF
      3  -8.54801574E+002  3.79693598E-001   5.28335886E-001      44.40  <-- SCF
      4  -8.52981743E+002  7.44988320E-001  -2.27478843E-001      52.26  <-- SCF
      5  -8.52884167E+002  9.08590414E-001  -1.21969434E-002      60.13  <-- SCF
      6  -8.52886334E+002  8.98636611E-001   2.70796284E-004      68.36  <-- SCF
      7  -8.52887081E+002  9.06719344E-001   9.34638588E-005      76.22  <-- SCF
      8  -8.52887250E+002  9.10591664E-001   2.11356795E-005      84.07  <-- SCF
      9  -8.52887250E+002  9.11100143E-001  -3.08962712E-008      88.90  <-- SCF
     10  -8.52887250E+002  9.11105407E-001  -4.65249702E-008      93.76  <-- SCF
     11  -8.52887250E+002  9.11110563E-001  -1.77844141E-008      98.98  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

and a two-core band-parallel calculation produces

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial   5.10476316E+002  4.62264099E+001                        16.67  <-- SCF
      1  -7.76802126E+002  2.64224391E+000   1.60909805E+002      28.13  <-- SCF
      2  -8.50574887E+002  2.02770490E-001   9.22159500E+000      38.62  <-- SCF
      3  -8.54801574E+002  3.79693598E-001   5.28335886E-001      52.16  <-- SCF
      4  -8.52981743E+002  7.44988320E-001  -2.27478843E-001      60.98  <-- SCF
      5  -8.52884167E+002  9.08590414E-001  -1.21969434E-002      70.63  <-- SCF
      6  -8.52886334E+002  8.98636611E-001   2.70796284E-004      79.96  <-- SCF
      7  -8.52887081E+002  9.06719344E-001   9.34638588E-005      88.78  <-- SCF
      8  -8.52887250E+002  9.10591664E-001   2.11356795E-005      97.65  <-- SCF
      9  -8.52887250E+002  9.11100143E-001  -3.08962712E-008     102.84  <-- SCF
     10  -8.52887250E+002  9.11105407E-001  -4.65248977E-008     108.05  <-- SCF
     11  -8.52887250E+002  9.11110563E-001  -1.77844503E-008     113.57  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

Note that the results as reported are identical for the first 9 SCF cycles, and only differ by $O(10^{-14})$eV/atom in the last two cycles, which is the same order as ϵ for double-precision arithmetic and so may be attributed to different rounding errors for the serial and band-parallel calculations.

This calculation takes longer when run band-parallel compared to the serial calculation, but this is not a cause for alarm – the test system is very small, containing only 16 valence bands, so it is not surprising that the communication overhead outweighs the gains.

The same calculation run using the 'all-bands' self-consistent code path yields

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy                           Energy gain       Timer   <-- SCF
                                               per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial   6.83465549E+002                                         10.35  <-- SCF
      1  -7.97053977E+002                    1.85064941E+002      20.90  <-- SCF
      2  -8.48247959E+002                    6.39924773E+000      31.57  <-- SCF
      3  -8.50914193E+002                    3.33279207E-001      42.16  <-- SCF
      4  -8.51618587E+002                    8.80493249E-002      54.31  <-- SCF
      5  -8.52080365E+002                    5.77221874E-002      64.82  <-- SCF
      6  -8.52436527E+002                    4.45203123E-002      75.48  <-- SCF
      7  -8.52663071E+002                    2.83179709E-002      85.99  <-- SCF
      8  -8.52769350E+002                    1.32848145E-002      96.64  <-- SCF
      9  -8.52812636E+002                    5.41075552E-003     107.15  <-- SCF
     10  -8.52829576E+002                    2.11747553E-003     117.69  <-- SCF
     11  -8.52836183E+002                    8.25924796E-004     128.48  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

in serial, and

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy                           Energy gain       Timer   <-- SCF
                                               per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial   6.83465549E+002                                          7.04  <-- SCF
      1  -7.97053977E+002                  1.85064941E+002        15.95  <-- SCF
      2  -8.48247959E+002                  6.39924773E+000        24.90  <-- SCF
      3  -8.50914193E+002                  3.33279207E-001        33.79  <-- SCF
      4  -8.51618587E+002                  8.80493249E-002        43.01  <-- SCF
      5  -8.52080365E+002                  5.77221874E-002        51.94  <-- SCF
      6  -8.52436527E+002                  4.45203123E-002        61.04  <-- SCF
      7  -8.52663071E+002                  2.83179709E-002        69.93  <-- SCF
      8  -8.52769350E+002                  1.32848145E-002        79.12  <-- SCF
      9  -8.52812636E+002                  5.41075552E-003        87.98  <-- SCF
     10  -8.52829576E+002                  2.11747553E-003        96.76  <-- SCF
     11  -8.52836183E+002                  8.25924796E-004       107.40  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

in two-core band-parallel. Note that this time there is a small speed improvement for the band-parallel run – this is because the 'all-bands' path does more FFTs per SCF cycle than the DM path, and the FFTs distribute trivially among the band-group.

With the basic band-parallelism tested and complete, Castep has been demonstrated to work in band-parallel mode for the EDFT and DM algorithms.

The only known problem outstanding is with the EDFT mode. In the EDFT algorithm the empty bands are optimised non-self-consistently after the full bands have been updated, but at the moment this does not use the same algorithm as the DM code path and so is not band-parallel.

4.4 Benchmarking and Performance

The first reasonable simulation we performed with the new band-parallelism was the al1x1 benchmark.

The test simulations were restricted to small test cases (8-atom silicon, and the al1x1 benchmark) and numbers of cores ($\leq$8), where the results could be compared in detail to known results and serial calculations, but once testing was complete we were able to move to larger systems. Table 4.2 shows the performance improvement for the al1x1 benchmark using the new band-parallel mode.

The performance was analysed using Cray's Performance Analysis Tool (PAT) version 4.2 (earlier versions had bugs which prevented them being

cores	DM efficiency
2	65%
4	50%
8	35%

Table 4.2: Parallel scaling of the al1x1 benchmark in band-parallel mode.

cores	Time (s)	band-parallel efficiency
8	5085.04	(k-point parallel)
16	3506.66	72%
32	2469.84	51%

Table 4.3: Execution time and parallel efficiency for the 33-atom TiN benchmark (8 k-points). Times are for 40 SCF cycles using the DM algorithm. The 8-core calculation is running purely k-point parallel, the others are running with mixed band and k-point parallelism.

used with the Pathscale compiler and/or Castep properly). It was also necessary to create symbolic links to the Castep source files in Source/Utility in Castep's obj/linux_x86_64_pathscale/Utility and similarly for Fundamental and Functional.

```
pat_build -D trace-max=2048 -u -g mpi,blas,lapack,math castep
```

We profiled a Castep calculation on the al1x1 benchmark parallelised over 16 nodes (4-way band-parallel, 4-way gv-parallel). The subroutine with the most overhead from the band-parallelism was `wave_rotate`, the Trace output of which was:

```
|   o-> wave_orthonormalise_over_slice        1290                           |
|   o-> electronic_find_eigenslice            2580                           |
|   o-> wave_rotate_slice                     3870        3870     40.06s    |
|   o-> wave_nullify_slice                    3870        3870      0.01s    |
|   o-> wave_allocate_slice                   3870        3870      0.01s    |
|   o-> wave_initialise_slice                 3870        3870      1.75s    |
|   o-> comms_reduce_bnd_logical              3870        3870      0.28s    |
|   o-> comms_reduce_bnd_integer              3870        3870      0.10s    |
|   o-> comms_send_complex                   23220       23220      6.48s    |
|   o-> comms_recv_complex                   23220       23220      7.79s    |
|   o-> wave_copy_slice_slice                 3870        3870      1.09s    |
|   o-> wave_deallocate_slice                 3870        3870      0.00s    |
```

This is to be expected, since these wavefunction rotations scale cubically with system size, and also incur a communication cost when run band-parallel. Some time was spent optimising this subroutine, and in the end we settled on a refactoring of the communications whereby each node does $log_2 nodes$ communication phases, the first phase involving an exchange of half the transformed data, and each subsequent phase exchanging half the data of the previous one. This scheme is illustrated in figure 4.1.

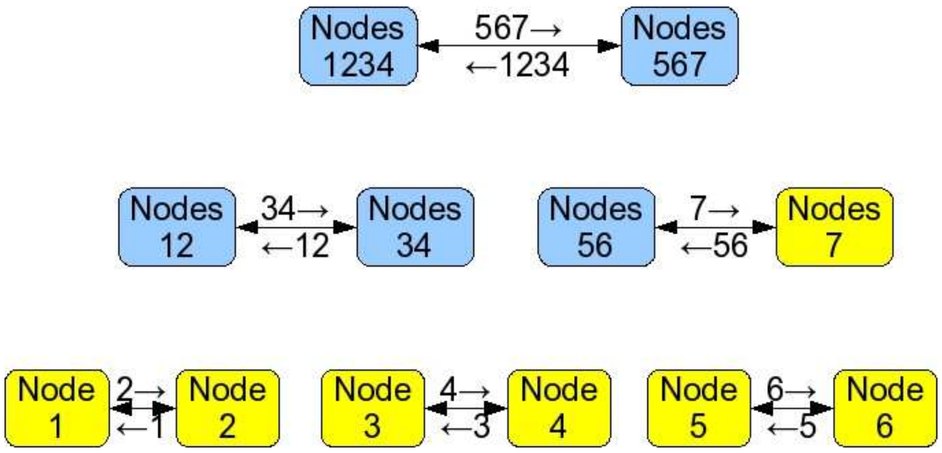


Figure 4.1: The new communication pattern, illustrated for seven nodes in the band group. Nodes with work still to do are coloured blue, and nodes that have finished are coloured yellow. At each of the three communication phases each group of nodes is split to form two child groups. Each node in a child group transforms its local data to produce its contribution to the nodes in the other child group, the 'sibling group'; it then sends this data to one of the nodes in that group, and receives the sibling node's contribution to all of the nodes in the child group.

This communication pattern improved the speed of the `wave_rotate` subroutine considerably, but at the cost of increased storage. Indeed the first phase involves the exchange of half of the newly transformed data, so the send and receive buffers constitute an entire non-distributed wavefunction. As the band-parallelisation is only efficient over relatively small numbers of nodes (typically ≤ 16) this has not proved too much of a hindrance thus far, but it would be wise to restrict this in future, perhaps to a low multiple of a single node's storage, at the cost of slightly more communication phases. Such a change could, of course, be made contingent on the value of the `opt_strategy_bias` parameter.

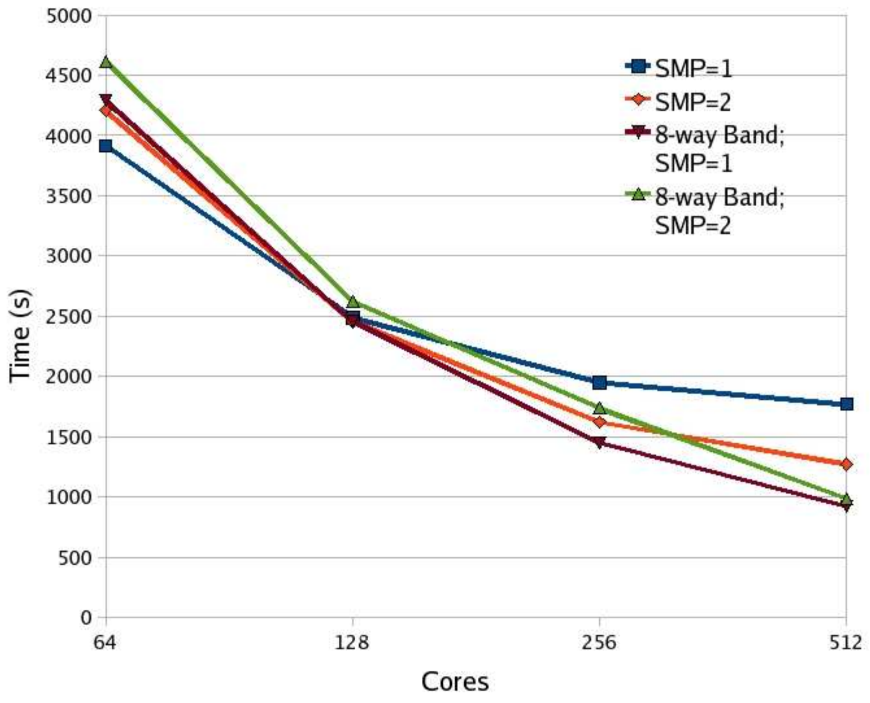
Once `wave_rotate` had been optimised, Castep's performance was measured on the `al3x3` benchmark. As can be seen from figure 4.2, the basic band-parallelism implemented in this stage of the project improved Castep's scaling considerably. Using linear interpolation of the data points we estimated that the maximum number of PEs that can be used with 50% or greater efficiency has been increased from about 221 to about 436 (without using the SMP optimisations).

4.5 Scope of Parallelism and Integration with main Castep Codebase

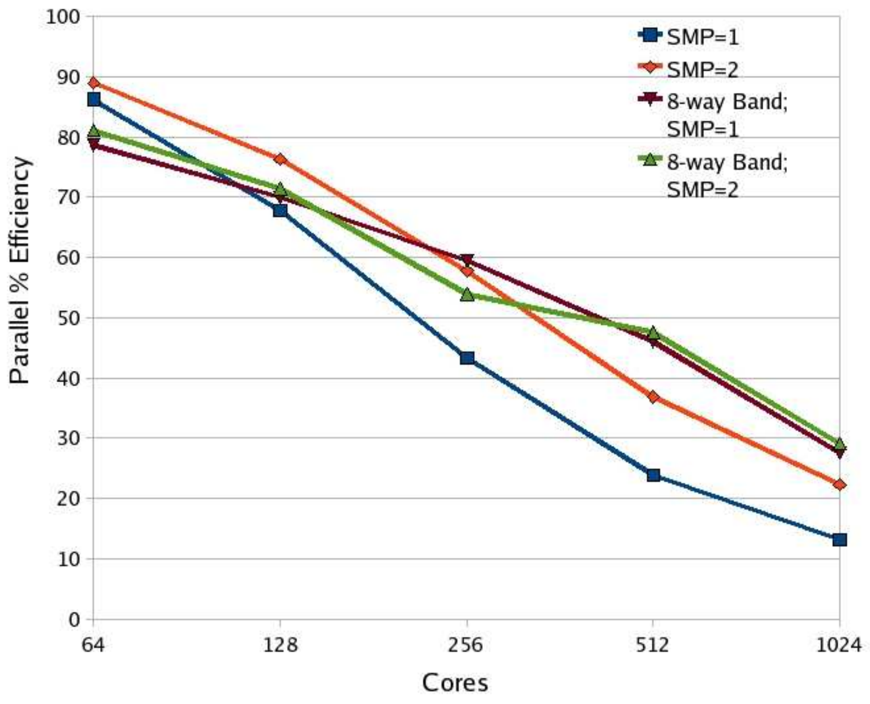
The current implementation of band-parallelism has been restricted to the groundstate calculation. Geometry optimisation and molecular dynamics calculations have not been tested, but it should be trivial to get them to work if they do not already. However EDFT with conduction bands, real space representations of β-projectors, and band-structure calculations all currently use an older, obsolete subroutine to obtain their eigenstates and this has not been parallelised over bands. The reason for this omission is simply that the correct course of action is to change these code paths to use the newer subroutine instead, which is already parallelised over bands.

The band-parallel version of Castep writes a standard Castep checkpoint file, so users wishing to perform operations that are not supported in band-parallel mode can still benefit from the band-parallel code by using it for their groundstate calculation, and then continuing their calculation with a normal Castep run.

An alpha-release of the band-parallel code was made to selected collaborators in June 2008. The aim is to integrate the band-parallelism and the main Castep code base for Castep release 4.4. This release is due in the last quarter of 2008, and will open up the band-parallelism to the whole Castep community. In addition to the groundstate calculations, geometry optimisations and molecular dynamics will be supported, as will EDFT. It is hoped that band-structure calculations and the use of real space β-projectors will also be supported, but depending on time this may need to wait until version 4.5.



(a) Castep performance



(b) Castep parallel efficiency relative to 16 PE Castep 4.2

Figure 4.2: Graphs showing the computational time (4.2(a)) and scaling (4.2(b)) of the band-parallel version of Castep, compared to ordinary Castep 4.2, for 10 SCF cycles of the standard `al3x3` benchmark.

Chapter 5

Distributed Diagonaliser and Inverter (Work Package 2)

5.1 Introduction

The first stage of the project included the distribution over the band-group of the calculation of the band-overlap matrices required for both orthonormalisation of the wavefunction and the subspace Hamiltonian. These matrices still need to be diagonalised (or inverted) and Castep 4.2 does this in serial. These operations scale as N_b^3 so as larger and larger systems are studied they will start to dominate. For this reason the second stage of the project involves distributing these operations over the nodes.

5.2 Programming

5.2.1 Development

Because the aim of this stage of the project was to implement a parallel matrix diagonaliser and inverter, which is independent of most of Castep, we created a testbed program to streamline the development.

The subspace diagonalisation subroutine in Castep uses the LAPACK subroutine `ZHEEV`, so we concentrated on that operation first. The testbed program creates a random Hermitian matrix, and uses the parallel LAPACK

	time for various matrix sizes					
cores	1200	1600	2000	2400	2800	3200
1	19.5s	46.5s	91.6s	162.7s		
2	28.3s	65.9s	134.6s			
4	15.8s	38.2s	54.7s	90.1s		
8	7.9s	19.0s	37.6s	63.9s	81.6s	
16	4.3s	10.5s	20.3s	32.5s	76.2s	
32	2.7s	6.0s	11.6s	19.2s	43.1s	

Table 5.1: Hermitian matrix diagonalisation times for the ScaLapack subroutine PZHEEV.

variant ScaLAPACK to diagonalise it.

5.2.2 ScaLAPACK Performance

The performance of the distributed diagonaliser (PZHEEV) was compared to that of the LAPACK routine `ZHEEV` for a range of matrix sizes.

An improved parallel matrix diagonalisation subroutine, PZHEEVR[1], was made available to us by Christof Vömel (Zurich) and Edward Smyth (NAG). This subroutine consistently out-performed PZHEEV, as can be seen from figure 5.1.

The ScaLAPACK subroutines are based on a block-cyclic distribution, which allows the data to be distributed in a general way rather than just by row or column. The timings for different data-distributions for the PZHEEVR subroutine are given in table 5.2.

The computational time t for diagonalisation of a $N \times N$ matrix scales as $O(N^3)$, so we fitted a cubic of the form

$$t(N) = a + bN + cN^2 + dN^3 \tag{5.1}$$

to these data for the 8-core runs. The results are shown in table 5.3. This cubic fit reinforces the empirical evidence that the PZHEEVR subroutines have superior performance and scaling with matrix size, since the cubic coefficient for PZHEEVR is around 20% smaller than that of the usual PZHEEV subroutine.

[1]The 'R' is because it uses the *Multiple Relatively Robust Representations (MRRR)* method

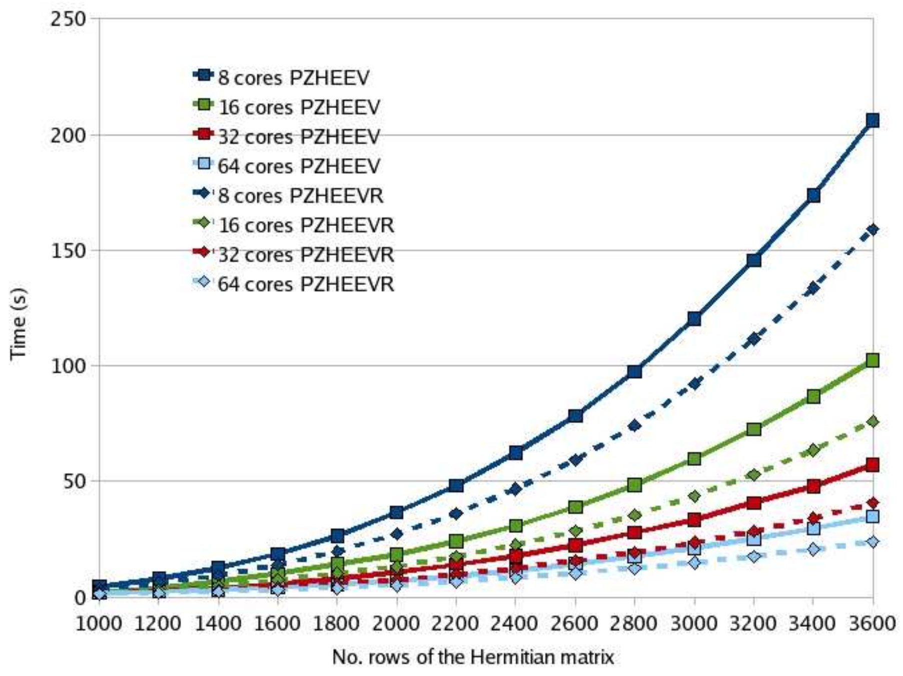


Figure 5.1: A graph showing the scaling of the parallel matrix diagonalisers PZHEEV (solid lines with squares) and PZHEEVR (dashed lines with diamonds) with matrix size, for various numbers of cores (colour-coded)

Cores used for distribution of		
Rows	Columns	Time
1	64	6.48s
2	32	6.45s
4	16	5.80s
8	8	5.92s

Table 5.2: PZHEEVR matrix diagonalisation times for a 2200x2200 Hermitian matrix distributed in various ways over 64 cores of HECToR.

Coefficient	PZHEEV	PZHEEVR
a	-1.43547	-0.492901
b	0.00137909	0.00107718
c	9.0013e-08	-7.22616e-07
d	4.31679e-09	3.53573e-09

Table 5.3: The best-fit cubic polynomials for the PZHEEV and PZHEEVR matrix diagonalisation times for Hermitian matrices from 1000×1000 to 3600×3600 distributed over 8 cores of HECToR.

5.2.3 Castep Implementation

In order to encapsulate the changes to the diagonalisation and inversion routines, we moved the operations from Castep's `wave` module to its general algorithm module `algor`. The existing subroutine `algor_invert` was overloaded to work with complex matrices as well as double-precision real ones, and also gained optional arguments to specify the symmetry of the matrix (e.g. Hermitian).

A new subroutine `algor_diagonalise` was created to handle matrix diagonalisation. This subroutine takes a complex matrix and returns its eigenvectors and eigenvalues. Optional arguments specify the symmetry of the matrix (e.g. Hermitian).

The initialisation of the Basic Linear Algebra Communication Subroutines (BLACS) that are used by ScaLAPACK for its communication is done in the `comms` module, inside `comms_parallel_strategy`. Because BLACS libraries are not commonplace, all of the BLACS and ScaLAPACK calls are wrapped inside conditional compilation sections – only if Castep is compiled with -DBLACS will the parallel matrix diagonalisers and inverters be used.

As has already been mentioned (e.g. sections 3.2.2 and 4.2.2), in testing it was found that in fact the largest matrix Castep diagonalises in the al3x3 benchmark is not a band-overlap matrix at all, but a projector-projector overlap matrix. Rewriting this to take advantage of the distributed inverter gave the single greatest performance benefit in this phase of the project.

5.2.4 Castep Performance

With the new distributed inversion and diagonalisation subroutines the performance and scaling of Castep was improved noticeably. As expected, this improvement was more significant when using larger number of cores. Figure 5.2 shows the improved performance of Castep due to the distribution of the matrix inversion and diagonalisation in this work package.

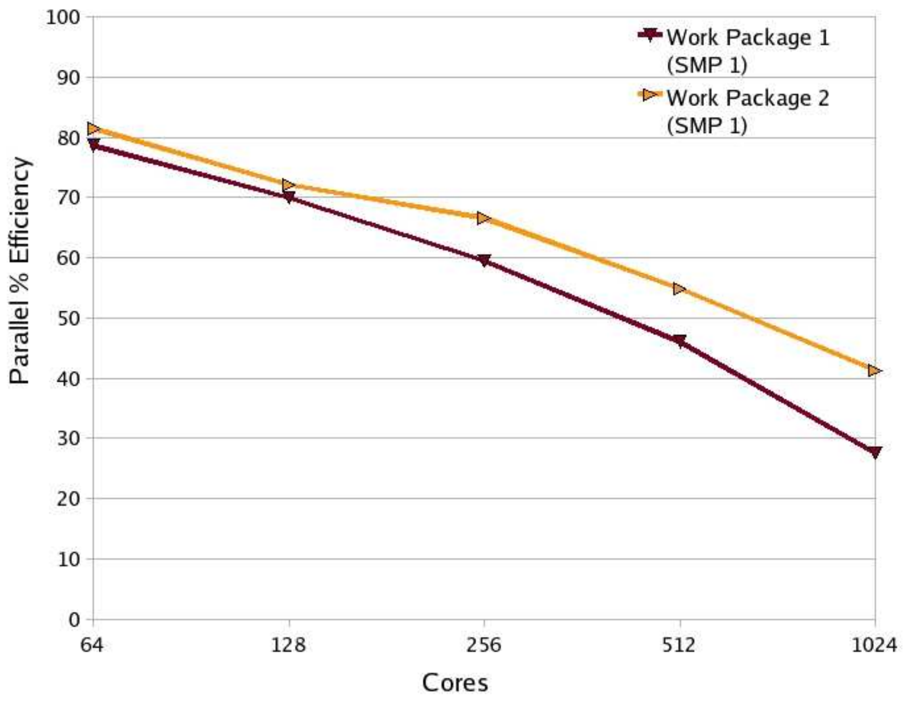


Figure 5.2: Graph showing the performance and scaling improvement achieved by the distributed inversion and diagonalisation work in Work Package 2, compared to the straight band-parallel work from Work Package 1. Each calculation is using 8-way band-parallelism, and running the standard `al3x3` benchmark.

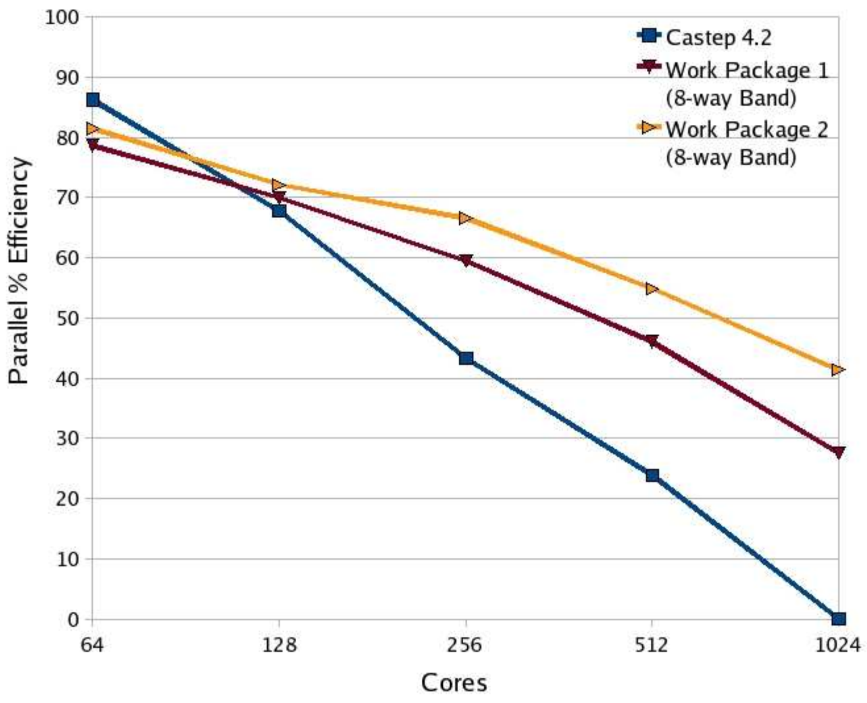
The distributed diagonalisation, on top of the basic band-parallelism, enables Castep calculations to scale effectively to between two and four times more cores compared to Castep 4.2 (see figure 5.3). The standard `al3x3` benchmark can now be run on 1024 cores with almost 50% efficiency, which equates to over three cores per atom, and it is expected that larger calculations will scale better. A large demonstration calculation is being performed that should illustrate the new Castep performance even better.

5.2.5 Limitations

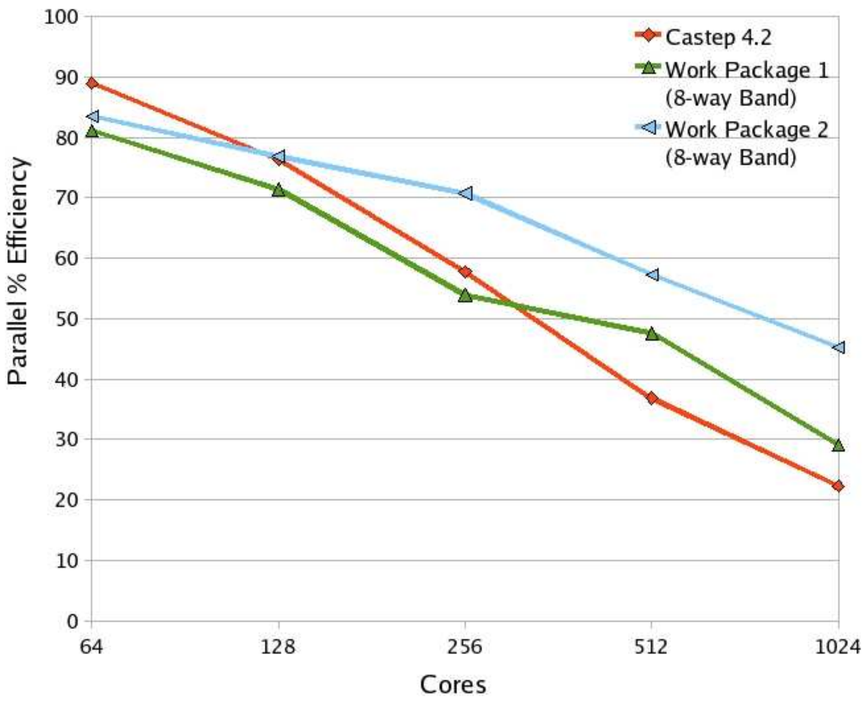
Although the *interfaces* to the `algor_invert` and `algor_diagonalise` subroutines allow general matrices and those of specific symmetries, the actual implementation has been restricted to Hermitian matrices. Hermitian matrices are by far the most common large matrices in Castep so this restriction has not been problematic so far, but it is straightforward to implement the operations for other matrices and this should be done at the earliest opportunity.

The current BLACS processor grid assumes that all nodes in the band- and gv-groups are available to participate in any ScaLAPACK operation. Although this is true for the matrices we have discussed so far, it would be better to define several BLACS grids and add optional arguments to `algor_invert` and `algor_diagonalise` to define the distribution. This would also aid optimisation, as at the moment the matrix that is passed into the `algor` subroutines must be the global, non-distributed matrix which often requires the calling routine to do an otherwise unnecessary reduction over the band- and/or gvector-groups.

Finally, at the time of writing there are two limitations inherent in the BLACS and ScaLAPACK libraries. Firstly, ScaLAPACK lacks a Hermitian matrix inverter, and so we have to use a general complex matrix inverter (`PZGETRF` and `PZGETRI`); for this reason it may be better to use the serial version of the subroutine when parallelising over only two PEs. The second limitation is that many of the diagonalisation and inversion routines require a so-called ‘square’ distribution of the matrices, i.e. the row and column block sizes are the same. This is unfortunate as the natural distribution in Castep is to distribute one over the band-group and the other over the gvector-group and these will not usually have the same number of nodes in them.



(a) `num_proc_in_smp : 1`



(b) `num_proc_in_smp : 2`

Figure 5.3: Comparison of Castep scaling for Work Packages 1 and 2 and the original Castep 4.2, for 10 SCF cycles of the al3x3 benchmark. Parallel efficiencies were measured relative to the 16 core calculation with Castep 4.2.

Chapter 6

Independent Band Optimisation (Work Package 3)

6.1 Introduction

As we have seen, the bottleneck in large Castep calculations is the explicit S-orthonormalisation of the eigenstates. This orthonormalisation involves the calculation and inversion of the band-overlap matrix, operations which scale as $N_p N_b^2$ and N_b^3 respectively, where N_p is the number of plane-wave basis states and N_b is the number of bands (eigenstates). Furthermore, when operating in band-parallel mode the former operation is also a communication bottleneck, as the individual eigenstates reside on different processing elements.

Clearly it is desirable to implement an optimisation scheme which will allow the approximate bands to be optimised without the need for an explicit S-orthonormalisation.

6.2 Castep Eigensolver

When using ultrasoft pseudopotentials, the DFT Schrödinger-like equation becomes a generalised eigenvalue problem,

$$\hat{H}\psi_i = \epsilon_i \hat{S}\psi_i, \tag{6.1}$$

where $\hat{H}$ and $\hat{S}$ are $N_p \times N_p$ matrices in a plane-wave basis set of N_p plane-waves, and we seek the lowest N_b eigenstates (where $N_b << N_p$).

Direct construction and diagonalisation of the Hamiltonian $\hat{H}$ is impractical, but there are a variety of methods available to solve for the lowest eigenstate iteratively. Typically Castep employs a Krylov subspace method:

1. The current (S-orthonormal) set of trial eigenstates is split into blocks, each block spanning a small subspace and update sequentially

2. For each ψ_i in the current block, $\hat{K}\left(\hat{H}\psi_i - \epsilon_i \hat{S}\psi_i\right)$ is computed (where $\hat{K}$ is a preconditioning matrix[2]) and added to the block subspace.

3. The augmented subspace is S-orthogonalised to the existing approximate eigenstates

4. The augmented subspace Hamiltonian is constructed and diagonalised, i.e. we solve

$$H\psi_i' = \epsilon_i' S\psi_i', \tag{6.2}$$

   where

$$\begin{aligned} H_{ij} &= \langle \psi_i | \hat{H} | \psi_j \rangle \\ S_{ij} &= \langle \psi_i | \hat{S} | \psi_j \rangle \end{aligned} \tag{6.3}$$

5. The new lowest eigenstates of the subspace are accepted as improved approximations to the true eigenstates, and the convergence criteria are checked. If they are not satisfied the algorithm repeats from step 2 with further augmentations of the subspace.

It is step 3 of this algorithm that we wish to avoid, but without it the eigenstates of each block would always tend to converge to those of the lowest block.

6.3 Proposed Eigensolver

Consider the following:

$$\begin{aligned}
\hat{H}\psi_i &= \epsilon_i \hat{S}\psi_i \\
\Rightarrow \left(\hat{H} - \lambda\hat{S}\right)\psi_i &= \left(\epsilon_i - \lambda\right)\hat{S}\psi_i \\
\Rightarrow \left(\hat{H} - \lambda\hat{S}\right)^{\dagger}\left(\hat{H} - \lambda\hat{S}\right)\psi_i &= \left(\epsilon_i - \lambda\right)^2\hat{S}\psi_i \qquad (6.4)
\end{aligned}$$

Equation 6.4 is simply a modified eigenvalue equation that shares the same eigenstates as the original Schrödinger-like equation but with modified eigenvalues. However the lowest eigenvalue of this modified equation is not the lowest energy eigenvalue ϵ_0, but the eigenstate whose eigenvalue is closest to λ. Furthermore, the eigenstate with eigenvalue λ is now at a local minimum, regardless of whether it has the globally lowest eigenvalue or not.

Consider now the usual Castep algorithm outlined in section 6.2, for the special case where each block contains only a single eigenstate. If we omit the orthonormalisation step then every approximate eigenstate will tend to converge to the lowest eigenstate, but if we first replace the augmented subspace matrix

$$H_{ij} = \langle \psi_i | \, \hat{H} \, | \psi_j \rangle \qquad (6.5)$$

with the modified matrix,

$$\begin{aligned}
M_{ij} &= \langle \psi_i | \left(\hat{H} - \lambda\hat{S}\right)^{\dagger}\left(\hat{H} - \lambda\hat{S}\right) | \psi_j \rangle \\
&= \left\langle \left(\hat{H} - \lambda\hat{S}\right)\psi_i \middle| \left(\hat{H} - \lambda\hat{S}\right)\psi_j \right\rangle
\end{aligned}$$

then by varying the value of λ we can select which of the modified eigenvalues is lowest.

If we choose $\lambda = \epsilon_i$ then this algorithm will automatically choose the best approximate eigenstate closest to the initial approximation. In this case the states in the inner product are the band residuals R,

$$R_i = \hat{H}\psi_i - \epsilon_i\hat{S}\psi_i \qquad (6.6)$$

and the algorithm is very similar to the *Residual Minimisation Method by Direct Inversion in an Iterative Subspace* algorithm (RMM-DIIS)[3].

The only difference between our scheme and the usual RMM-DIIS method is the choice of state to add to our block's subspace. In the usual Castep scheme we add a state ϕ given by

$$\phi = \hat{K}\left(\hat{H}\psi_i - \epsilon_i\hat{S}\psi_i\right). \qquad (6.7)$$

In the RMM-DIIS scheme we search along this direction for the minimal residual norm, i.e. find a new trial eigenstate along the direction ϕ, and add that trial eigenstate to the subspace instead of ϕ. The schemes are so similar that we implemented both in Castep.

After all of the eigenstates have been updated, a global orthonormalisation is still necessary before a new density can be constructed. However this single orthonormalisation per SCF cycles is considerably fewer than the usual Castep scheme where three or four orthonormalisations are per SCF cycle are common.

Both schemes converge to the closest eigenstate to the current trial eigenstate, so it is important that we have a reasonable set of trial eigenstates before using these schemes. We opted to use the usual Castep optimisation for four SCF cycles before switching to one of the new schemes.

6.4 Performance

Unfortunately neither the RMM-DIIS optimiser nor our variation proved to be either robust or quick; the reduction in orthonormalisations reduced the SCF cycle time considerably, but vastly more SCF cycles were needed for convergence. The RMM-DIIS scheme in particular suffered from severe numerical instabilities near convergence, since the residual matrix becomes more and more singular as the trial eigenstates approach the true eigenstates.

In order to ensure only the direct changes to the optimiser were observed, we ran Castep for a *fixed* density. Typical convergence for a simple magnesium oxide test case using the usual Castep algorithm is:

```
------------------------------------------------------------------------ <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
------------------------------------------------------------------------ <-- SCF
Initial  -4.95078326E+003  5.20975146E+001                         2.99  <-- SCF
      1  -5.59753549E+003  7.89244217E+000   8.08440297E+001       3.90  <-- SCF
      2  -5.66226988E+003  7.15740116E+000   8.09179761E+000       4.68  <-- SCF
      3  -5.66301246E+003  7.16625993E+000   9.28225593E-002       5.76  <-- SCF
      4  -5.66306881E+003  7.16423308E+000   7.04427727E-003       7.01  <-- SCF
      5  -5.66306893E+003  7.16423173E+000   1.49140438E-005       8.41  <-- SCF
      6  -5.66306893E+003  7.16423137E+000   4.02077714E-007       9.87  <-- SCF
      7  -5.66306893E+003  7.16423137E+000   5.27220802E-008      11.01  <-- SCF
      8  -5.66306893E+003  7.16423137E+000   1.76063159E-009      11.94  <-- SCF
      9  -5.66306893E+003  7.16423137E+000   3.90757352E-010      12.61  <-- SCF
     10  -5.66306893E+003  7.16423137E+000   1.33410476E-011      13.10  <-- SCF
     11  -5.66306893E+003  7.16423137E+000   5.99380402E-012      13.53  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

Switching to the RMM-DIIS gave

```
----------------------------------------------------------------------- <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
----------------------------------------------------------------------- <-- SCF
Initial  -4.95078326E+003  5.20975146E+001                         2.85  <-- SCF
     1  -5.59753549E+003  7.89244217E+000   8.08440297E+001        3.69  <-- SCF
     2  -5.66226988E+003  7.15740116E+000   8.09179761E+000        4.41  <-- SCF
     3  -5.66301246E+003  7.16625993E+000   9.28225593E-002        5.39  <-- SCF
     4  -5.66306881E+003  7.16423308E+000   7.04427727E-003        6.55  <-- SCF
     5  -5.66306892E+003  7.16423445E+000   1.34939453E-005        7.66  <-- SCF
     6  -5.66306891E+003  7.16423645E+000  -1.18066010E-006        8.76  <-- SCF
     7  -5.66306893E+003  7.16423715E+000   2.90858466E-006       10.06  <-- SCF
     8  -5.66306893E+003  7.16424036E+000   5.70679748E-008       11.11  <-- SCF
     9  -5.66306893E+003  7.16424784E+000  -9.96659399E-008       12.21  <-- SCF
    10  -5.66306890E+003  7.16426887E+000  -3.48059116E-006       12.98  <-- SCF
    11  -5.66306893E+003  7.16499317E+000   3.20356934E-006       13.78  <-- SCF
    12  -5.66306893E+003  7.16435180E+000  -1.03932948E-007       14.46  <-- SCF
    13  -5.66306893E+003  7.16439686E+000  -1.62527990E-007       15.22  <-- SCF
    14  -5.66306892E+003  7.16467568E+000  -2.95401151E-007       15.88  <-- SCF
    15  -5.66306891E+003  7.16448445E+000  -1.60189845E-006       16.58  <-- SCF
    16  -5.66306891E+003  7.16566473E+000   5.03725090E-008       17.30  <-- SCF
    17  -5.66305809E+003  7.16892722E+000  -1.35318489E-003       17.94  <-- SCF
    18  -5.66289950E+003  7.17878051E+000  -1.98232364E-002       18.59  <-- SCF
    19  -5.66295014E+003  7.20703280E+000   6.33023123E-003       19.23  <-- SCF
    20  -5.65353849E+003  7.27226034E+000  -1.17645706E+000       19.82  <-- SCF
----------------------------------------------------------------------- <-- SCF
```

Even with this small test case there was a slight improvement in the SCF cycle time, but the numerical instabilities caused the solution to diverge eventually. Our modified algorithm proved slightly more stable for this test case, but slower and also showed signs of diverging:

```
----------------------------------------------------------------------- <-- SCF
SCF loop      Energy           Fermi           Energy gain       Timer   <-- SCF
                               energy          per atom          (sec)   <-- SCF
----------------------------------------------------------------------- <-- SCF
Initial  -4.95078326E+003  5.20975146E+001                         3.25  <-- SCF
     1  -5.59753549E+003  7.89244217E+000   8.08440297E+001        5.65  <-- SCF
     2  -5.66226988E+003  7.15740116E+000   8.09179761E+000        6.44  <-- SCF
     3  -5.66301246E+003  7.16625993E+000   9.28225593E-002        7.51  <-- SCF
     4  -5.66306881E+003  7.16423308E+000   7.04427727E-003        8.79  <-- SCF
     5  -5.66306892E+003  7.16423445E+000   1.34668025E-005       10.08  <-- SCF
     6  -5.66306891E+003  7.16423645E+000  -1.24884522E-006       11.33  <-- SCF
     7  -5.66306825E+003  7.16423517E+000  -8.20412820E-005       12.79  <-- SCF
     8  -5.66306852E+003  7.16424032E+000   3.31933376E-005       13.98  <-- SCF
     9  -5.66306886E+003  7.16424765E+000   4.29159705E-005       15.26  <-- SCF
    10  -5.66306888E+003  7.16426863E+000   2.20859478E-006       16.17  <-- SCF
    11  -5.66306892E+003  7.16496512E+000   5.82997827E-006       17.23  <-- SCF
    12  -5.66306892E+003  7.16434686E+000  -2.59482603E-007       17.99  <-- SCF
    13  -5.66306892E+003  7.16439357E+000  -5.01087043E-007       18.80  <-- SCF
    14  -5.66306891E+003  7.16466770E+000  -1.12797108E-006       19.56  <-- SCF
    15  -5.66306890E+003  7.16447879E+000  -1.59632306E-006       20.42  <-- SCF
```

```
     16  -5.66306886E+003  7.16561534E+000  -4.04882177E-006      21.16  <-- SCF
     17  -5.66306881E+003  7.16881083E+000  -6.32000384E-006      21.89  <-- SCF
     18  -5.66306867E+003  7.17899059E+000  -1.85164167E-005      22.64  <-- SCF
     19  -5.66306845E+003  7.20738379E+000  -2.73264238E-005      23.33  <-- SCF
     20  -5.66306769E+003  7.29973182E+000  -9.40002692E-005      24.05  <-- SCF
------------------------------------------------------------------------ <-- SCF
```

These results were fairly typical of the performance of these optimisers–it was relatively straightforward to get them close to the groundstate, but difficult to get the accuracy we require. Imposing orthonormality on the updates enabled both methods to converge quickly and robustly, indicating that this poor performance was not a bug, but inherent in the algorithms. We investigated restricted orthonormalisation, whereby only certain directions are projected out, but although this improved matters neither algorithm converged reliably.

There are several improvements that could be made to try to improve the new optimisers, and of course it would be trivial to alternate the new and old optimisers in some way, but unfortunately we did not have time to investigate further. For the moment we shall have to content ourselves with the tried-and-trusted Castep optimiser.

Chapter 7

Other Developments and Future Work

7.1 Γ-Point

During this dCSE project NAG has generously allowed the author to participate in two Castep Developer Group (CDG) workshops, one in St Andrews in January 2008, and one in York in July 2008. These workshops gathered together all of the Castep Developers to work on an optimisation known as Γ-point.

As has already been noted in section 2.1, the number of k-points required for a Castep calculation decreases as the system size increases, so that for most HPC Castep calculations only O(1) k-points are required. If the simulation system is large enough, it can be described well by sampling only at the so-called Γ-point, k=(0,0,0). This is important because at the Γ-point the eigenstates can be chosen to be explicitly real, rather than complex. Not only does this halve the storage requirements for the wavefunctions, it also doubles the speed of many of the operations on the wavefunctions, including the FFTs.

Over the course of the two CDG workshops this optimisation has been implemented in the Castep 4.4 codebase, and has already been tested and shown to be working on HECToR. Switching on the Γ-point optimisations, a Castep groundstate calculation on a 1230 atom polypeptide, running on 512 cores of HECToR, went from a computational time of 27,523s to 14,261s, a speed-up of over 1.9.

In principle the orthogonalisation and subspace rotations can be quadrupled in speed, since all the complex-complex operations become real-real. However in reciprocal-space the wavefunctions are still complex, and it is only their dot-products that are real–this is not straightforward to exploit using standard BLAS calls and has not yet been implemented fully in Castep.

7.2 Distributing β-Projectors and $\langle \beta | \phi \rangle$

As noted in section 4.2.2, the β-projectors are not currently distributed over the band-group. This is a problem for large systems as the number of projectors is proportional to the number of atoms. For the al3x3 benchmark there are 88,184 plane-waves and 2,160 β-projectors, so their total storage comes to almost 3GB per k-point–there is no way a purely band- or mixed k-point/band-parallel calculation can run on HECToR under these circumstances simply because of insufficient RAM. Distributing the associated workload over the band-group should also improve the run-time, provided the extra communications can be kept to a minimum.

In addition to the β-projectors themselves, the inner products of the projectors with the wavefunction bands are also non-distributed. These so-called $\langle \beta | \phi \rangle$ data are computed and stored in the wavefunction, wavefunction_slice and band data types for efficiency. For ease of use the array is stored in unpacked form, but this is undesirable for large systems since the addition of a single new species of ion can increase the size of the array dramatically.

Because the $\langle \beta | \phi \rangle$ array has data for each band, it is naturally distributed over the band-group, as implemented in this dCSE project. However the projector dimension is not distributed, and it is not distributed over the gvector-group; thus the array could be distributed by projector index over the gvector-group to further reduce the memory overhead and improve scaling.

Both of these proposed distributions would save memory and improve runtime. However care must be taken in the implementation, as projector data would be distributed differently depending on whether it was the projectors themselves (which are currently distributed over the gvector-group and would be then distributed over the band-group too), or the $\langle \beta | \phi \rangle$ arrays (which are *already* distributed over the bands, and would now be distributed over gvector-group as well).

7.3 Mpitrace

The development, debugging and profiling of Castep was made substantially easier by the presence of a built-in Trace module. This module logs almost all of the subroutine and function entries and exits, and keeps track of their parents and children. In addition to this there are optional facilities to time the calls, and also to log each entry and exit–all on a per-PE basis.

By switching on the subroutine logging, it is possible to see which routine each PE is in on any machine at any point of a Castep calculation. This can aid the programmer in tracking down possible parallel node-desynchronisation, especially if Castep hangs, but the root cause is often a long way back up the call-tree and can prove very difficult to locate. Furthermore, the Trace module needs to be lightweight enough (in terms of time and memory) that it can always be used without undue penalty and so it lacks the facility to store or log variables etc.

A possible extension has been proposed by Dr K. Refson, which is essentially MPI-Trace. This would contain its own, independent communication layer, communicators etc. to enable it to verify the state of a parallel Castep calculation. Calls to this module could check whether the nodes are synchronised at this point across the gvector-, band- or k-point groups, compare key variables, or even check MPI collectives, or trap mismatched point-to-point communications.

Chapter 8

Final Thoughts

8.1 This Project

The project as a whole has gone smoothly, and broadly to plan. The main problem with the project was a lack of time. When the project was proposed, the work was split into four phases with the first phase estimated to take two months, the next four, the third two and the fourth phase four months for a total of a year. Because the fourth phase was not funded, the project as a whole was shortened to eight months. Unfortunately it has been anticipated that much of the general optimisation work would be carried out during this fourth phase, since only then would the full performance implications be clear. The omission of this phase from the project meant that the optimisation of the band-scheme had to be squeezed into the other eight months, with the result that the workload was rather more intense than expected and the final result not quite as polished as had been hoped. It also meant that the third phase of the project, the band-independent optimisation algorithm, was never likely to be generally useful.

The problems with time outlined above were exacerbated by the inclusion of a 'work package 0', without a corresponding extension to the overall timeline. Whilst a general analysis of Castep performance was necessary, the detailed option-by-option study requested by NAG was unexpected and it would have been better to set aside an extra month for this work rather than try to fit it into the existing timeframe.

One final frustration with the general schedule of the project was the holiday allowance. Due to various negotiations between NAG and the University of York regarding the precise financial arrangements, the holiday allowance was set at seven weeks, which was of course extremely generous. Unfortunately at nearly 20% of the project time it was completely impractical to actually use most of this allowance and still expect to get the work done.

Despite these misgivings, the project has been very successful. The original goal of the project was to improve the scaling of Castep with cores by 'a factor of 8 or more to over 2000'. Despite the lack of the fourth phase's parallel band-independent optimiser, the scaling of Castep has been improved by about a factor of four, and with the extra work on Γ-point optimisations the target factor of eight or more in performance awaits only the integration of the band-parallel developments with the latest Castep source.

The considerable computational resources of HECToR, coupled with the developments in Castep detailed in this dCSE project report, should enable accurate DFT calculations on a far greater scale than has been possible to this point.

8.2 HECToR as a Development Machine

HECToR is clearly an excellent machine for running Castep, even without the efficiency gains made in this project. However there are some features that have made using it as a *development* machine rather difficult. Chief amongst these are:

- Buffered I/O
  It is obviously important for performance to buffer I/O, but HECToR does not flush these buffers when the system call `flush` is invoked, or even on job termination. This made tracking bugs down extremely difficult using Castep's built-in Trace logging, or even adding `write` statements.

- Out-of-Memory not logged for user
  When one of HECToR's compute nodes runs out of memory, the Linux OOM module kills a randomly selected process. This may be the Castep job, in which case the job terminates. However the PBS output shows only 'exit code 137', indicating that the job was killed, but not why. The OOM module may also kill any Totalview process, rendering the debugger useless in such cases.

- No dedicated benchmarking time
  In the process of developing and testing the modified Castep, calculations had to be performed on a large number of nodes. Many of these calculations were short–the al3x3 benchmark, for example, takes less than 15mins on 2000 PEs or more–and it would have been very useful to have some time set aside for benchmarking. Perhaps this time could be made available after scheduled downtime.

Bibliography

[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"*, Zeit. für Kryst. **220(5-6)** (2004) 567–570

[2] P.J. Hasnip and C.J. Pickard, *"Electronic energy minimisation with ultrasoft pseudopotentials"*. Comp. Phys. Comm. **174** (2006) 24–29

[3] P. Pulay, *"Convergence acceleration of iterative sequences. The case of SCF iteration"*. Chem. Phys. Lett. **73** (1980) 393–398