Bands-parallelism in Castep

A dCSE Project

Phil Hasnip

August 14, 2008

Declaration

This short report emphasising on CASTEP performance has been extracted from the original report on Bands-parallelism is Castep, a dCSE project. In case of any kind of ambiguity arising from the contents of this short report, the reader is advised to consult the original full-length report on the subject.

Sarfraz Nadeem

Contents

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 was included as one of the benchmark programs used in the HECToR procurement exercise. 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.

1.1 The dCSE Project

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.

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. 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.

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).

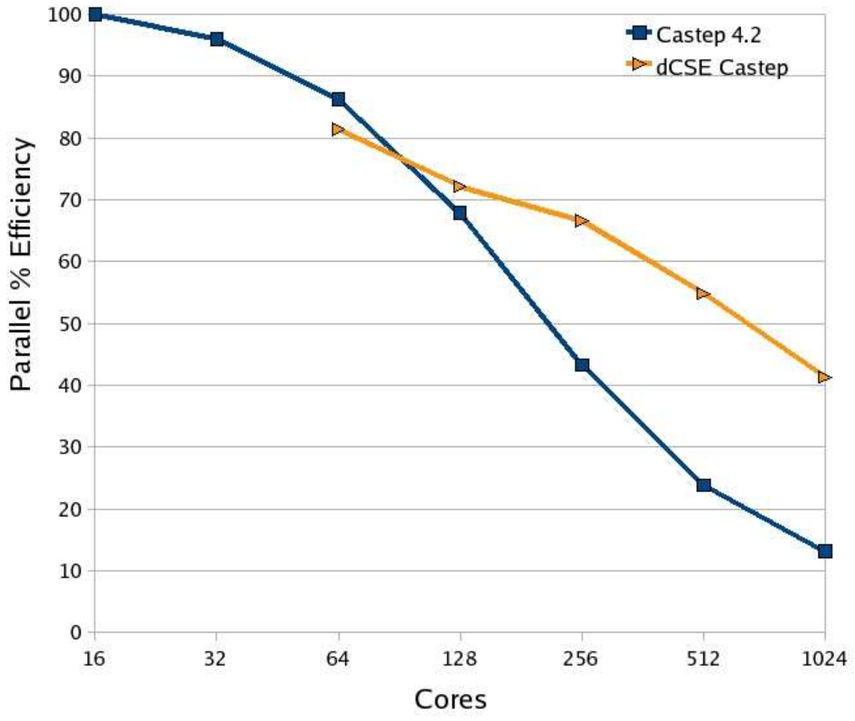


Figure 1.1: Graph showing the performance and scaling improvement achieved by this dCSE project (using 8-way band-parallelism) compared to the ordinary Castep 4.2 code for the standard `al3x3` benchmark.

Chapter 2

Castep Performance on HECToR (Work Package 0)

2.1 General Castep Performance

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.

2.2 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`.

2.2.1 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.

As can be seen from figure 2.1 FFTW 3.1.1 was the fastest FFT library available on HECToR.

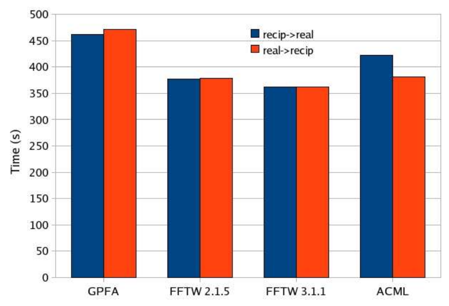


Figure 2.1: Graph showing the relative performance of the four FFT subroutines available to Castep on HECToR for the TiN benchmark. This benchmark transforms wavefunction slices to real space 1634 times and back again.

2.2.2 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 2.2 Cray's LibSci 10.2.1 was by far the fastest BLAS library available on HECToR, at least for ZGEMM.

2.2.3 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.

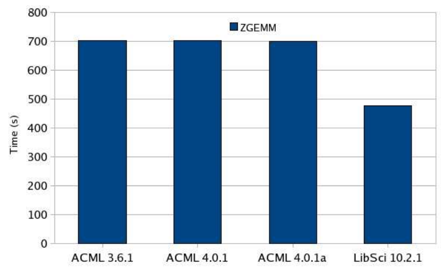


Figure 2.2: Graph showing the relative performance of the ZGEMM provided by the four maths libraries available to Castep on HECToR for the TiN benchmark. This benchmark performs 4980 projector-projector overlaps using ZGEMM. Castep's internal Trace module was used to report the timings.

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

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

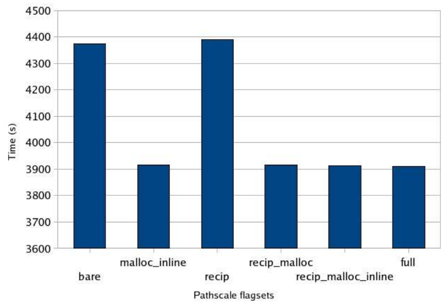
and created six compilation flagsets. The first set, which was used as a base for all the other sets, just used `-O3 -OPT:Ofast`, and we named this the `bare` set. The other five used this, plus:

```
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
```

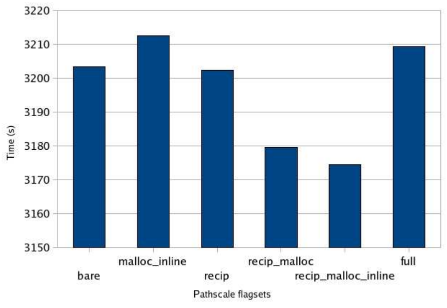
The performance of the various Castep binaries can be seen in figure 2.3. It is clear that the flags we were given by Alan Simpson are indeed the best of this set.

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
```



(a) TiN benchmark (16 PEs)



(b) al3x3 benchmark (32 PEs)

Figure 2.3: Comparison of Castep performance for the Pathscale compiler with various flags, using 39 SCF cycles of the TiN benchmark (2.3(a)) and 11 SCF cycles of the al3x3 benchmark (2.3(b)).

```
      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
```

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 2.4 shows the run times of both the Portland Group and Pathscale compiler as reported by the PBS output for the TiN benchmark.

2.2.4 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 2.5.

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 2.5. 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

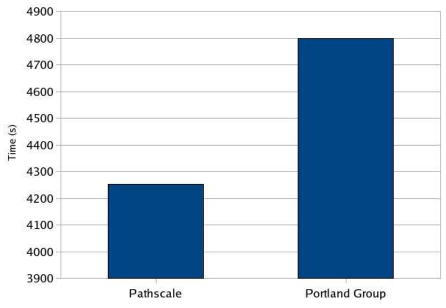
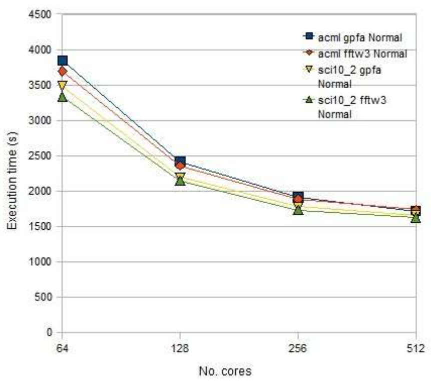
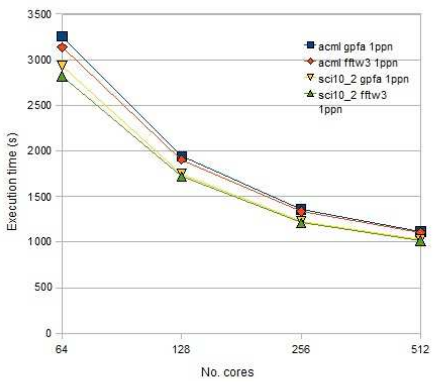


Figure 2.4: Graph showing the relative performance of the Pathscale 3.0 compiler (recip_malloc_inline flags) with the Portland Group 7.1.4 compiler (-fastsse -O3 -Mipa) for the TiN benchmark on 16 PEs. The PBS reported walltime was used to report the timings.



(a) Execution time using 2 cores per node



(b) Execution time using 1 core per node

Figure 2.5: 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 (2.5(a)) and one core per node (2.5(b))

facility for optimising communications within an SMP node[1] the scaling was improved dramatically and rivals that of the one core per node runs.

2.3 Baseline

We decided to choose the Pathscale 3.0 binary, compiled with the recip_malloc_inline flags (see section 2.2.3) and linked against Cray's Libsci 10.2.1 and FFTW3 for our baseline, as this seemed to offer the best performance with the 4.2 Castep codebase.

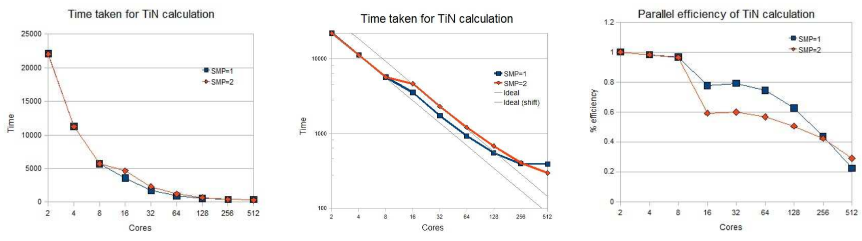
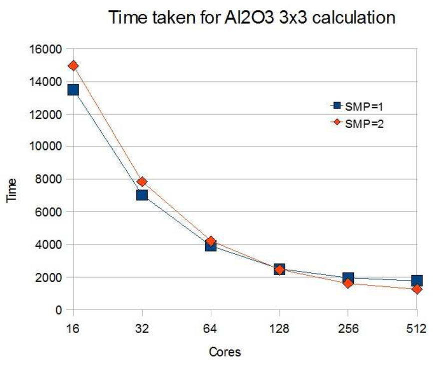
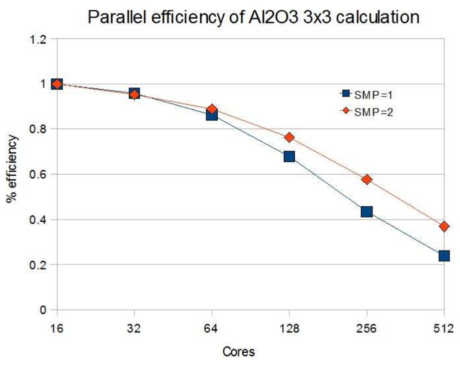


Figure 2.6: Execution time for the 33 atom TiN benchmark. This calculation is performed at 8 k-points.



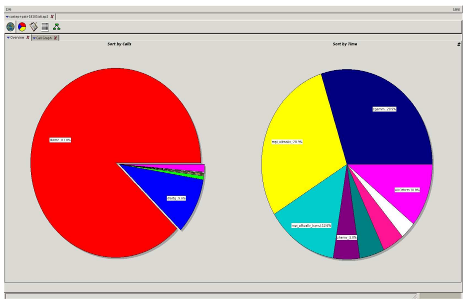
(a) Execution time



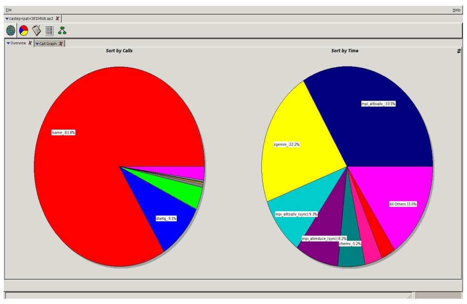
(b) Efficiency with respect to 16 cores

Figure 2.7: Scaling of execution time with cores for the 270 atom Al2O3 3x3 benchmark. This calculation is performed at 2 k-points.

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

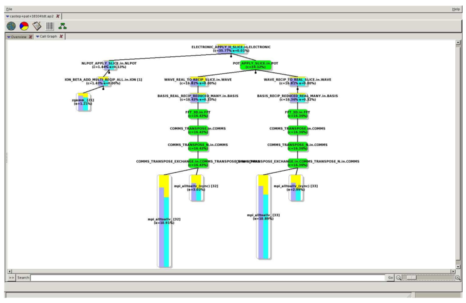


(a) CPU time for Castep on 256 cores

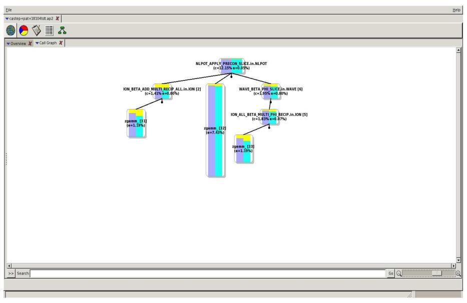


(b) CPU time for Castep on 512 cores

Figure 2.8: Breakdown of CPU time for 256 (2.8(a)) and 512 (2.8(b)) cores using 2 ppn, for Castep Al2O3 3x3 benchmark



(a) CPU time spent applying the Hamiltonian in Castep



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

Figure 2.9: 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

2.4 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.

Chapter 3

Band-Parallelism (Work Package 1)

3.1 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.

3.2 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.

3.3 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 3.1 shows the performance improvement for the al1x1 benchmark using the new band-parallel mode.

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

Table 3.1: 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 3.2: 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.

The performance was analysed using Cray's Performance Analysis Tool (PAT) version 4.2. 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 3.1.

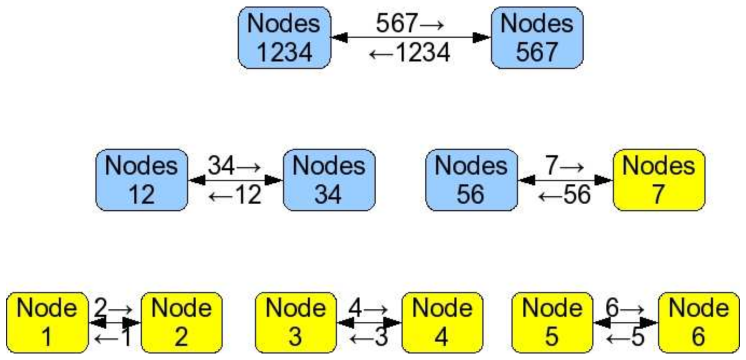
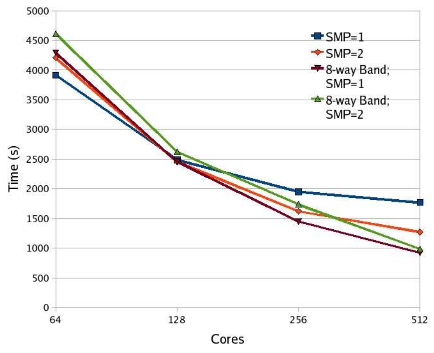


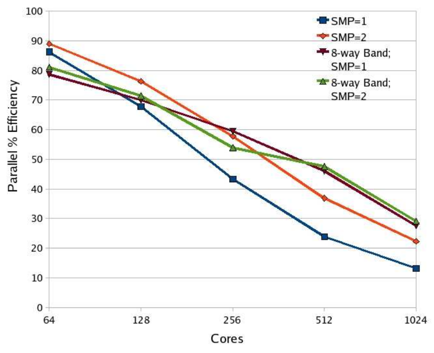
Figure 3.1: The new communication pattern, illustrated for seven nodes in the band group. Nodes with work still to do are coloured blue, and nodes that have finished are coloured yellow. At each of the three communication phases each group of nodes is split to form two child groups. Each node in a child 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 3.2, 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).



(a) Castep performance



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

Figure 3.2: Graphs showing the computational time (3.2(a)) and scaling (3.2(b)) of the band-parallel version of Castep, compared to ordinary Castep 4.2, for 10 SCF cycles of the standard `al3x3` benchmark.

Chapter 4

Distributed Diagonaliser and Inverter (Work Package 2)

4.1 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.

4.2 Programming

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 variant ScaLAPACK to diagonalise it.

4.3 ScaLAPACK Performance

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

	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 4.1: Hermitian matrix diagonalisation times for the ScaLapack subroutine PZHEEV.

An improved parallel matrix diagonalisation subroutine, PZHEEVR (The 'R' is because it uses the *Multiple Relatively Robust Representations (MRRR)* method) , 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 4.1.

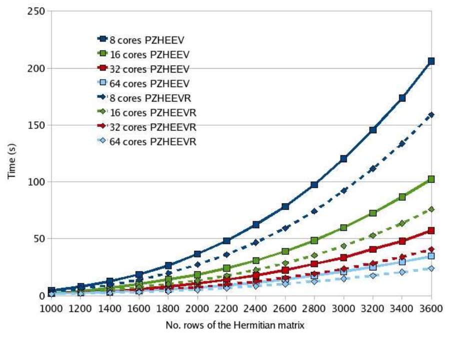


Figure 4.1: A graph showing the scaling of the parallel matrix diagonalisers PZHEEV (solid lines with squares) and PZHEEVR (dashed lines with diamonds) with matrix size, for various numbers of cores (colour-coded)

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 4.2.

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

$$t(N) = a + bN + cN^2 + dN^3 \tag{4.1}$$

to these data for the 8-core runs. The results are shown in table 4.3. This cubic fit reinforces the empirical evidence that the PZHEEVR subroutines have

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

superior performance and scaling with matrix size, since the cubic coefficient for PZHEEVR is around 20% smaller than that of the usual PZHEEV subroutine.

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 4.3: 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.

4.4 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 4.2 shows the improved performance of Castep due to the distribution of the matrix inversion and diagonalisation in this work package.

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 4.3). 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.

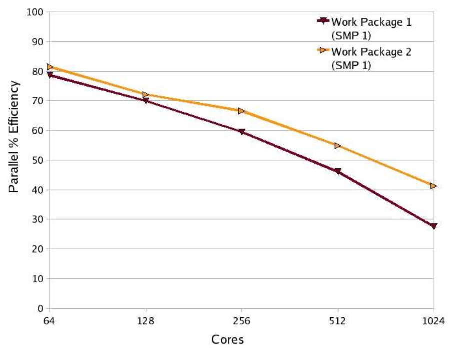
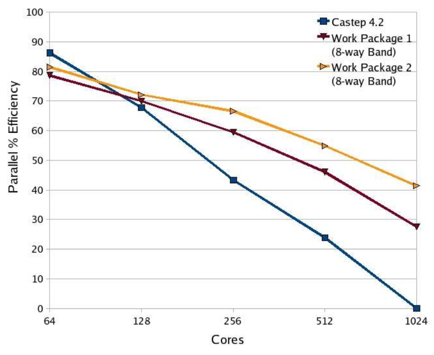
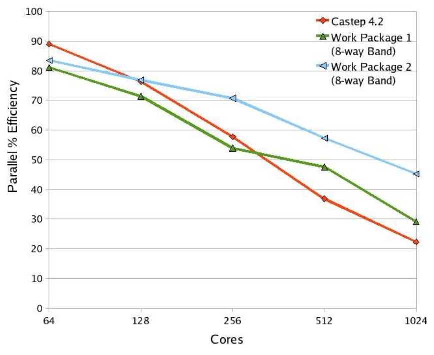


Figure 4.2: Graph showing the performance and scaling improvement achieved by the distributed inversion and diagonalisation work in Work Package 2, compared to the straight band-parallel work from Work Package 1. Each calculation is using 8-way band-parallelism, and running the standard `al3x3` benchmark.



(a) `num_proc_in_smp : 1`



(b) `num_proc_in_smp : 2`

Figure 4.3: 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 5

Independent Band Optimisation (Work Package 3)

5.1 Introduction

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.

5.2 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.

Chapter 6

Final Thoughts

6.1 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 any process including 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.
- **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