CASTEP Quad Core Benchmarking and Optimisation

Asimina Maniopoulou, Chris Armstrong
CSE Team
NAG Ltd., support@hector.ac.uk

June 2009

1 Introduction

The following investigations have been made on the HECToR TDS (Test and Development System) which is configured as the main system will be in phase 2a (aside from some differences in the IO system), consisting of quad core AMD Opteron processors. The system has 64 nodes, giving 256 processing cores in total.

2 Benchmarking CASTEP 4.4 as it stands

CASTEP 4.4 was compiled with the Pathscale 3.2 compiler on HECToR (dual core nodes) and the HECToR TDS (quad core nodes). The flags used for the compilation for both cases were chosen to be the ones prescribed in the CASTEP dCSE Project report [1]. The flags read:

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

In addition a new module "xtpe-quadcore" was loaded during compilation, which appends a quadcore-specific optimisation flag (for Pathscale -march=barcelona) and links to a quadcore-optimised version of the Cray libsci library, if used.

On the quadcore system it was observed that the executable exited with a segmentation fault after preforming all of the SCF calculations. The routine that was failing was basis_sum_recip_grid in the Fundamental/basis.F90 file. The problem was found to be related to the array "grid" in that routine. We could overcome this issue by compiling the aforementioned file at lower optimization level. As the executable was found to seg. fault even at level -O2, the optimization flags used were:

```
-O1 -OPT:Ofast -OPT:recip=ON -OPT:malloc_algorithm=1 -inline -INLINE:preempt=ON
```

It should be noted that this did not degrade the performance of the code at all. Alternatively maybe one could use compiler optimization directives in basis.F90, so that lower optimization is applied only to the problematic function and not the whole file.

Tables 1-3 below give the performance results for three benchmarks run on dual- and quad-core systems. (Details about the benchmarks can be found in [1].) The times recorded in the tables are the last SCF cycle in secs reported in the corresponding ⟨*benchmark*⟩.castep output file. In bold we point out the cases where the performance on the quadcore is inferior to that of than the dual-core system.

nprocs	dualcore	quadcore	speedup
8	398.31	327.16	1.22
16	231.60	218.77	1.06
32	174.02	218.91	0.79
64	194.57	301.10	0.65

Table 1: Performance of the al1x1 benchmark (last SCF cycle: 7). Time is given in secs.

2.0.1 Benchmark al1x1

For 32 and 64 cores we note that performance is worse on the quad-core system, although the multicore option in the castep.param file yields better performance (See table 8 in Section 5.2). The improvement is very small for this benchmark and certainly does not compensate for the 1.5 times increase in AU cost. Our observation is that since al1x1 is a small case the inferior performance for > 16 procs is caused by communication dominating more than for smaller job sizes, and with four cores there is more contention on the path to the interconnect in each node (see Section 5 for an optimisation related to this problem).

2.0.2 Benchmark al3x3

nprocs	dualcore	quadcore	speedup
32	5948	4759.03	1.25
64	3121.02	2500.48	1.25
128	1915.14	1696.30	1.13
256	1464.74	1526.11	0.96

Table 2: Performance of the al3x3 benchmark(last SCF cycle: 11). Time is given in secs.

In table 2 the measurements on 32 and 64 cores show a speedup of about **1.25** times, which is quite close to the 1.5 expected. On 256 cores we note that scaling degrades when compared with the old dual core system (0.96 slowdown), which agrees with the observation made for al1x1.

2.0.3 Benchmark TiN-mp

nprocs	dualcore	quadcore	speedup
64	785.09 (42)	654.33 (42)	1.20
128	572.12 (39)	422.02 (39)	1.35
256	370.71 (42)	404.35(42)	0.92

Table 3: Performance of the TiN-mp benchmark (Number of last SCF cycle depends on the number of processors and is the number in brackets next to the SCF timing). Time is given in secs.

The speedup here (table 3) is **1.2** on 64 cores and **1.35** on 128. Scaling deterioration is evident on 256 cores.

In a summary large benchmarks show a speedup between 1.2 and 1.35 times. Evidently though, CASTEP scales much worse on the quad core system than the dual core system on more than 128 processors for large jobs (al3x3 and TiN-mp benchmarks), while for smaller jobs scaling issues become evident even on 32 cores (al1x1 benchmark). This scaling deterioration is partly addressed in Section 5.

3 CASTEP Tuning on the quadcore system

3.1 Compiler Optimization

A number of different compilation flags were tried on top of the existing ones, but no further optimization was achieved. Some of these flags were

```
-ipa, -LNO:blocking=ON, -LNO:prefetch=3, -LNO:prefetch_ahead=3, -LNO:prefetch_ahead=4,
-LNO:full_unroll_outer=ON, -LNO:full_unroll_size=10000,  -LNO:fu=12,
-OPT:unroll_times_max=16,  -LNO:fission=2
```

It should be noted that when we tried to enforce more aggressive vectorization using the flag **-LNO:simd2** the compilation failed with a segmentation fault. One could use this flag only when the optimization level was reduced to -O2, but the combination lead to inferior performance.

3.2 Optimization via Environment Variables

We experimented with various MPICH environment variables, including huge page file support[1]. Out of these we found particularly beneficial for CASTEP the MPICH_RANK_REORDER_METHOD. When the scaling deteriorates (i.e. on 256 processors for the benchmarks al3x3 and TiN-mp) it is also beneficial to change the ALLTOALLV algorithm used by setting MPICH_ALLTOALLVW_RECVWIN =1 (the details of what this does are not clear, but it has the effect of using a different algorithm for MPI_Alltoallv).

3.2.1 MPICH_RANK_REORDER_METHOD

The default MPI rank placement scheme is SMP-style placement. Experimenting with different schemes yielded that the folded-rank placement can provide a speedup of 1.15 times over the default placement for large jobs. When the folded rank placement is enforced ranks are placed on the next node in the list and once every node has been used, the rank placement folds from the last node, going back to the first. Further investigations should be made, but the following results show that castep users should set this environment variables in their job scripts always.

nprocs	default-smp	folded	speedup
8	327.16	325.87	1.00
16	218.77	219.44	1.00
32	218.91	214.78	1.02
64	301.10	294.05	1.02

Table 4: Performance of the al1x1 benchmark (last SCF cycle: 7). Time is given in secs.

For the small benchmark al1x1, there seems to be little benefit of enforcing the folded-rank placement.

3.2.2 MPICH_ALLTOALLVW_RECVWIN

In the following table we compare the default performance on quadcore to the performance obtained when both MPICH_RANK_REORDER=2 and MPICH_ALLTOALLVW_RECVWIN =1. It should be noted

[1]From the Cray XT Programming Environment User's Guide, "Cray XT systems support 2 MB huge pages and 4 KB base pages for CNL applications. Previous versions of CNL supported only base pages. For applications that use a large amount of virtual memory, 4 KB pages can put a heavy load on the virtual memory subsystem. Huge pages can provide a significant performance increase for such applications. The 4 KB base pages remain the default."

nprocs	default-smp	folded	speedup
32	4759.03	4539.56	1.05
64	2500.48	2402	1.04
128	1696.30	1620.38	1.05
256	1526.11	1405.14	1.09

Table 5: Performance of the al3x3 benchmark (last SCF cycle: 11) with folded rank order. Time is given in secs

nprocs	default-smp	folded	speedup
64	654.33 (42)	606.37	1.08
128	422.02 (39)	376.66	1.12
256	404.35	350.48	1.15

Table 6: Performance of the TiN-mp benchmark with folded rank order. (Number of last SCF cycle depends on the number of processors). Time is given in secs.

that the effect of MPICH_ALLTOALLVW_RECVWIN=1 is greater with the default-smp rank order, but the optimal performance is obtained when both MPICH environment variables are set.

benchmark	nprocs	default dualcore	default quadcore	folded	folded + RECVWIN
al3x3	256	1464.74	1526.11	1405.14	1344.12
TiN-mp	256	370.71	404.35	350.48	344.03

Table 7: Optimal Performance with MPICH environment variables set. Timings are given for the last SCF cycle.

We suggest further experiments with MPICH_ALLTOALLVW_RECVWIN and the corresponding SEND environment variable for different number of processors.

Table 7 shows that with these environment variables set the performance of CASTEP for al3x3 and TiN-mp on the quadcore machine is now superior to the dualcore machine, although it should be noted that the dual core machine has not been tested with the environment variables mentioned in this section set.

4 Profiling

We profiled the code on the quadcore system (using TiN-mp) to examine mainly vectorization and cache utilization (which are extremely important on the quadcore system).

4.1 User functions

The most time consuming user functions when the code ran on 64 cores were found to be:

```
|| Time%  | Time        | D1 Cache hit, miss ratio | Function
||  56.0% | 419.238296  | 94 %                     | main
||   9.7% |  72.656015  | 94 %                     | POT_NONGAMMA_APPLY_SLICE.in.POT
||   6.3% |  46.856810  | 96.6%                    | COMMS_TRANSPOSE_N.in.COMMS
||   3.7% |  27.553444  | 78.9%                    | WAVE_INITIALISE_SLICE.in.WAVE
||   1.4% |  10.700597  | 88.5%                    | ION_Q_RECIP_INTERPOLATION.in.ION
||   1.1% |   8.429017  | 89.6%                    | ION_APPLY_AND_ADD_YLM.in.ION
```

This shows a fairly low cache hit:miss ratio. Also, vectorization was found to be an issue for the functions ION_Q_RECIP_INTERPOLATION.in.ION and ION_APPLY_AND_ADD_YLM.in.ION. Using the Pathscale flags

```
-FLIST:=on   -LNO:simd_verbose
```

during compilation, one finds that non-contiguous access to arrays is a very common occurrence and this prohibits both cache utilization and vectorization. It was found that the compiler failed to vectorized in many locations where mixed type operations (double complex and double precision) are used. We find this a surprising outcome and suggest further tests should be performed to verify that this is the case.

4.2 MPI functions

4.2.1 64 cores

When TiN-mp is run on 64 cores here are the most time consuming MPI functions. Before each MPI collective a barrier is set and the time needed for the synchronization is recorded as MPI_function_name_(sync). The time needed to complete the collective is given separately.

```
Time % |       Time    | Imb. Time |   Imb. |  Calls    |Function
|| 16.1% | 119.638932  |        -- |     -- |  285344.2 |MPI
||-----------------------------------------------------------------
||  13.9% | 103.655824 |  6.980603 |   6.4% |  207187.5 |mpi_alltoallv_
||   1.1% |   8.160020 |  1.956344 |  19.6% |     622.3 |mpi_recv_
||=================================================================
|   5.2% |  39.001770  |        -- |     -- |  284091.5 |MPI_SYNC
||-----------------------------------------------------------------
||   2.2% |  16.375367 |  4.858887 |  23.2% |  207187.5 |mpi_alltoallv_(sync)
||   2.2% |  16.000521 |  4.925778 |  23.9% |   58403.2 |mpi_allreduce_(sync)
```

4.2.2 256 cores

```
 100.0% | 1161.156466 |        -- |     -- | 1788338.6 |Total
|-------------------------------------------------------------------
|  58.8% |  682.655539 |        -- |     -- |  513017.7 |MPI
||------------------------------------------------------------------
||  54.9% |  636.979240 | 89.855737 |  12.4% |  392235.5 |mpi_alltoallv_
||   2.0% |   23.695335 |  5.885210 |  20.0% |   87424.8 |mpi_allreduce_
||   1.4% |   16.091737 |  1.547607 |   8.8% |    1317.8 |mpi_recv_
||==================================================================
||==================================================================
|  18.8% |  217.732094 |        -- |     -- |  510374.0 |MPI_SYNC
||------------------------------------------------------------------
||   9.3% |  107.482739 | 82.866435 |  43.7% |   87424.8 |mpi_allreduce_(sync)
||   7.8% |   90.137755 | 68.867932 |  43.5% |  392235.5 |mpi_alltoallv_(sync)
||   1.0% |   11.947095 | 10.972329 |  48.1% |       7.0 |mpi_barrier_(sync)
```

Comparing the MPI communication time for the same benchmark run on 64 and 256 cores, we can see that MPI communication prevails over the execution time. Special attention is paid to mpi_alltoallv, as used for the 3D FFT operations, in sections 5 and 6. The location and purpose of these MPI calls should be determined and understood. Load balancing issues come up as now almost 19 % of the execution time is spent on synchronization.

While some effort needs to be dedicated to improving vectorization and utilizing the cache more efficiently, the 1.2 to 1.3 factor speedup on the quadcore is quite good. Scaling is the issue that should be addressed in a dCSE project that aims to improve performance of CASTEP on the quadcore.

The next two sections investigate the effect of two suggested optimisations.

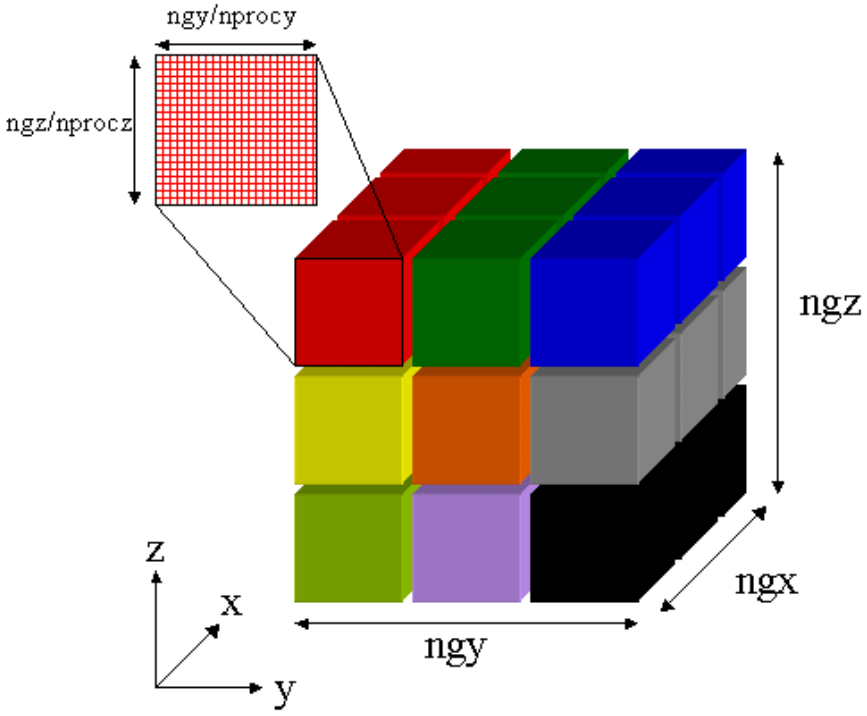

Figure 1: Domain decomposition for 3D parallel FFT. Step 1: perform FFTs in x-direction.

5 Introducing a shared memory buffer for MPI_Alltoallv communication

5.1 Background

The strategy CASTEP uses to perform 3D FFTs (Fast Fourier Transform) in parallel is to have each process perform separate 1D transforms in the x, y and z directions on a subset of the grid using a "pencil" distribution. We will illustrate this procedure using a simple case consisting of 9 processors.

Figure 1 illustrates the domain decomposition when each process is performing a number of transforms in the x-direction. The data on each process is represented by a different colour. There are ngx, ngy, ngz points in the x, y and z directions respectively. Thus, each process will perform $\frac{ngy}{nprocy} \cdot \frac{ngz}{nprocz}$ transforms in the x-direction in Figure 1. The next step is to rotate (or "transpose") the data so that each process can perform transforms in the y-direction. The resulting data distribution is given in Figure 2. This process is then repeated, transposing data so that each process is able to perform transforms in the z-direction (Figure 3).

5.2 Implementation

The most expensive aspect to this operation on HECToR is transposing data between calculating the 1D FFTs. This is implemented using MPI_Alltoallv. CASTEP has two options for performing the transposes. The default mode of operation is to have every process participate in the MPI_Alltoallv call. The other option is for multicore systems whereby a single process per node makes the call to MPI_Alltoallv. This is achieved by using MPI_Gather within a node to collect data to be sent into a single send buffer. The nominated participating processes then take part in the MPI_Alltoallv call, thus minimising contention

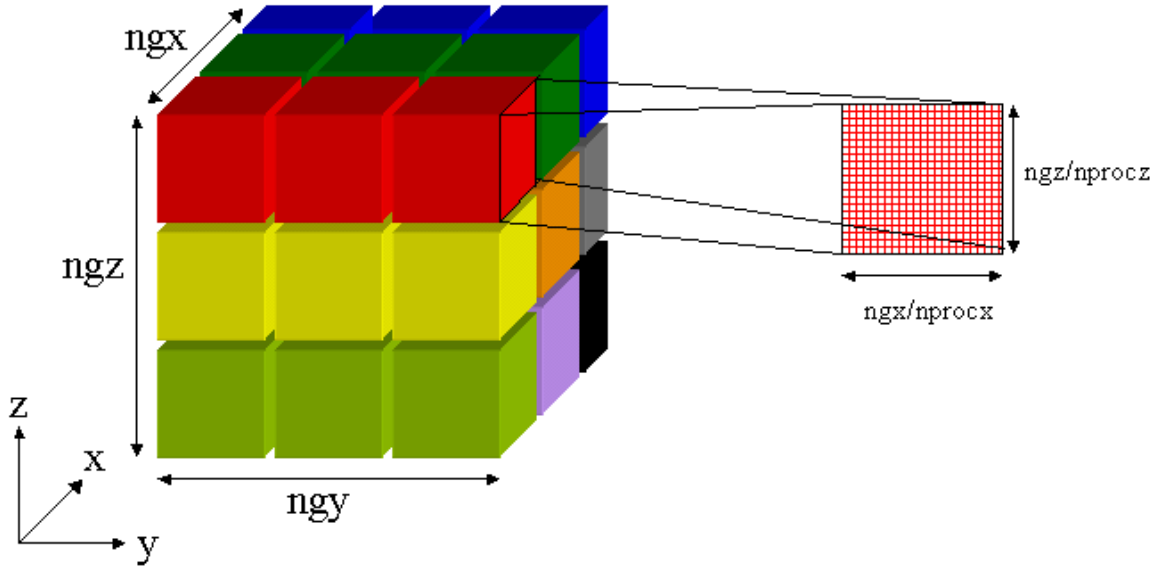

Figure 2: Domain decomposition for 3D parallel FFT. Step 2: transpose data as shown in Figure 1 and perform FFTs in y-direction.

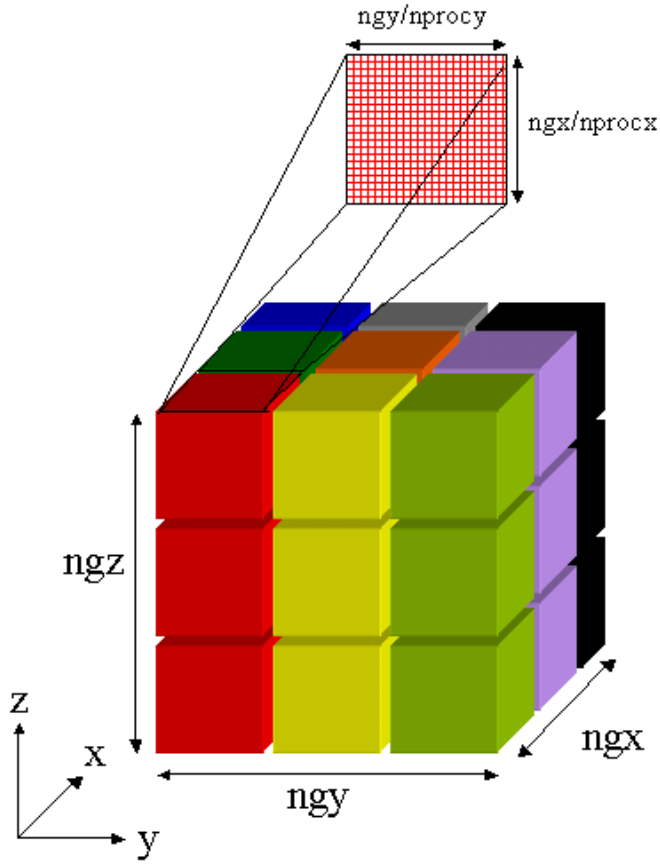

Figure 3: Domain decomposition for 3D parallel FFT. Step 3: transpose data as shown in Figure 2 and perform FFTs in z-direction.

on the interconnect. Data received by the participating processes is then scattered to the other processes in the node.

The main cost of using the multicore option is associated with gathering data within a node, then marshalling it into the correct order for the MPI_Alltoallv call, and unmarshalling then scattering after the MPI_Alltoallv call. For example, running the al1x1 benchmark on 16 processors in multicore mode gives an overall run time of 292.1s versus 222.9s in default mode. In default mode, 68.6s were spent in the routine that performs the transpose (comms_transpose_exchange) compared with 115.8s in multicore mode. Gathering and marshalling took 17.5s, MPI_Alltoallv took 57.5s, and unmarshalling and scattering took 58.9s (timings averages across processors).

Introducing a shared memory (System V shm) buffer provides the opportunity to merge the costly gather/marshal and unmarshal/scatter tasks into one loop either side of the MPI_Alltoallv call since every process within a node can directly pack or unpack its data. Doing so reduces the gather/marshal time to 8.7s and the unmarshal/scatter time to 7.2s. The overall run time when using the shared memory buffer reduces to 207.6s.

Tables 8-10 below give the performance results for three benchmarks. The default mode of operation tends to be faster when fewer processes are used because the overhead of packing and unpacking the send/recv buffers is greater. The shm optimisation pays off well for long jobs using more processes. For example, we see a 1.44 times speedup in the performance of al3x3 on 128 processes. The most spectacular result is the 1.78 times speedup for al1x1 running on 32 processes. It is noted that the introduction of the shm buffer has improved the scaling of the code in all cases.

nprocs	default	multicore	shm	default:shm
8	334.9	545.3	387.9	0.86
16	222.9	292.1	207.6	1.07
32	223.5	157.3	125.5	1.78

Table 8: Performance of the al1x1 benchmark (SCF cycle 7)

nprocs	default	multicore	shm	default:shm
64	2260.6	3102.7	2665.3	0.85
128	1788.1	1783.9	1571.5	1.14
256	1563.9	1191.3	1086.6	1.44

Table 9: Performance of the al3x3 benchmark (SCF cycle 11)

nprocs	default	multicore	shm	default:shm
32	1324.53	1931.1	1404.9	0.94
64	655.0	993.5	748.9	0.87
128	431.3	550.1	415.6	1.04

Table 10: Performance of the TiN-mp benchmark (SCF cycle 39)

5.3 Suggestions for further (possible dCSE) work

The current shm shared memory implementation makes use of calls to C in order to allocate and attach the shm memory segments. In order to incorporate this work into the CASTEP code base the C routines should be interfaced to Fortran in a standard way, for example using the Fortran 2003 standard.

The current implementation assumes an SMP-style rank placement, as is the default on HECToR. However, Section 3.2.1 has shown that CASTEP can benefit from a folded placement strategy. The C code used to interface to shm does contain a function to determine placement, based on reading a Cray system file. A further challenge is to find a portable way of determining rank placement at runtime in order to investigate the effect of different strategies on the shm optimisations.

In order to enable the multicore option described above the option num_proc_in_smp was set to 4 in the .param files (i.e. group all 4 cores within a node into a single SMP group), but experience on HPCx has shown that there are situations in which it is beneficial to set this value lower than the maximum number of cores available in a node in order to optimise the size of the MPI_Alltoallv message. It has been suggested that an empirical algorithm be investigated for optimising message size in this way, mirroring what had been used on HPCx.

6 Investigating the effect of bands blocking FFTs

The 3D FFT operation described in Section 5.1 is performed consecutively on an independent sequence of geometrically identical 3D domains, each of which is called a band. Since the transpose in a particular direction is the same for all bands, there is the opportunity to pack together multiple bands into a single MPI_Alltoallv call in order to hide latency.

This investigation has concentrated on implementing this optimisation for a single instance where such "bands blocking" is available: in subroutine basis_real_recip_reduced_many. This subroutine contains a loop of the form:

```
do nb=1,nblock
 call basis_fft_wrapper(ngx,ngy,ngz,grid(1,nb),-1,'W')
```

which performs a 3D FFT over nblock bands. Each band is stored in a column of the grid array.

This loop has been changed to loop over bandblock bands as follows:

```
do nb=1,nblock/bandblock
  call basis_fft_wrapper_bb(ngx,ngy,ngz,grid(1,(nb-1)*bandblock+1),&
                            -1,'W',max_grid_points)
```

(along with a loop to tidy up the remaining bands).

The new FFT wrapper routine (implemented in the FFTW3 wrapper only) performs bandblock dfftw_execute_dft calls before transposing in a single call the data for bandblock bands. The new transpose operation packs bandblock bands into the shm send buffer, performs MPI_Alltoallv and unpacks bandblock bands from the recv buffer. Packing and unpacking the send and recv buffers is done with an adapted indexing scheme to ensure the correct placement of data.

Figures 4-6 show the performance of this bands blocking for the three benchmarks with timings taken from CASTEP's own tracing. In all cases performance degrades with the number of bands blocked. Figure 7 shows why this performance degradation occurs: while the cost of the MPI_Alltoallv call decreases up to 12 bands, the cost of packing and unpacking the shared memory buffers increases with number of bands (timings here taken from MPI_Wtime on a separate run). This shows that the idea behind bands blocking – packing more data into send buffers to mitigate latency – is sound, but the new memory access pattern used to pack and unpack the buffers tends to dominate. In order to make effective use of bands blocking it is essential to optimise the way in which data is copied to and from the shared memory buffers.

Table 11 shows gives the performance figures for 5 bands divided by the same figures for 1 band (i.e. a value < 1 indicates superior performance for 5 bands; a value > 1 indicates inferior performance).

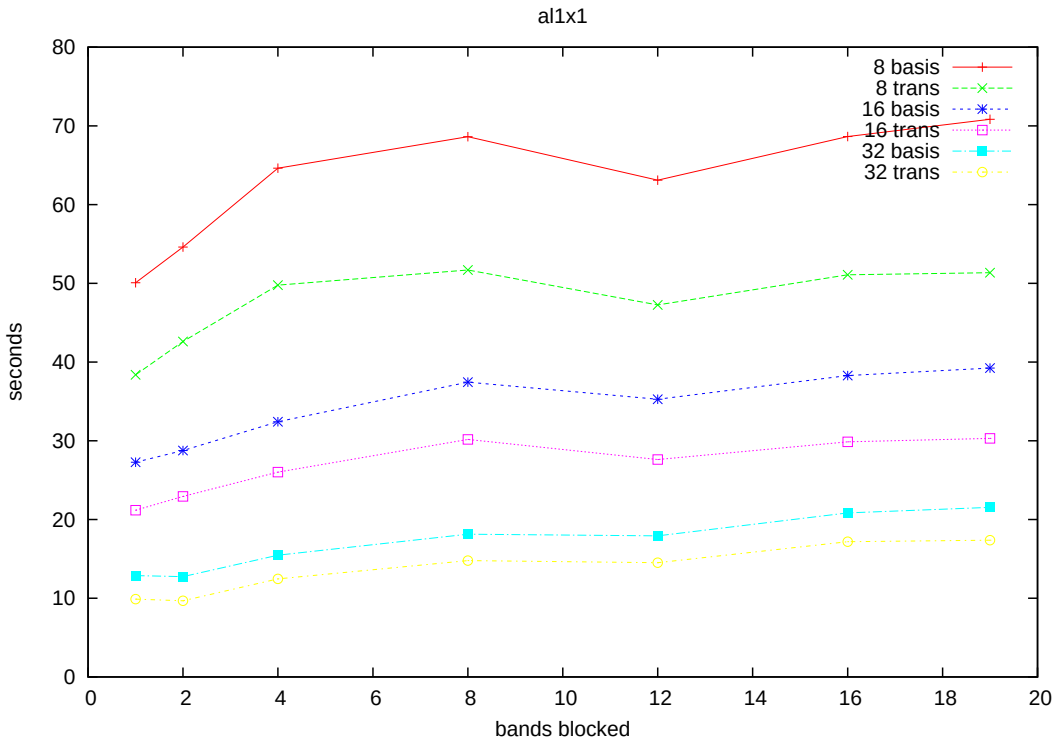

Figure 4: Performance of al1x1 for 8, 16, 32 processors. "basis" indicates time traced from basis_real_recip_reduced_many; "trans" indicates time traced in comms_transpose.

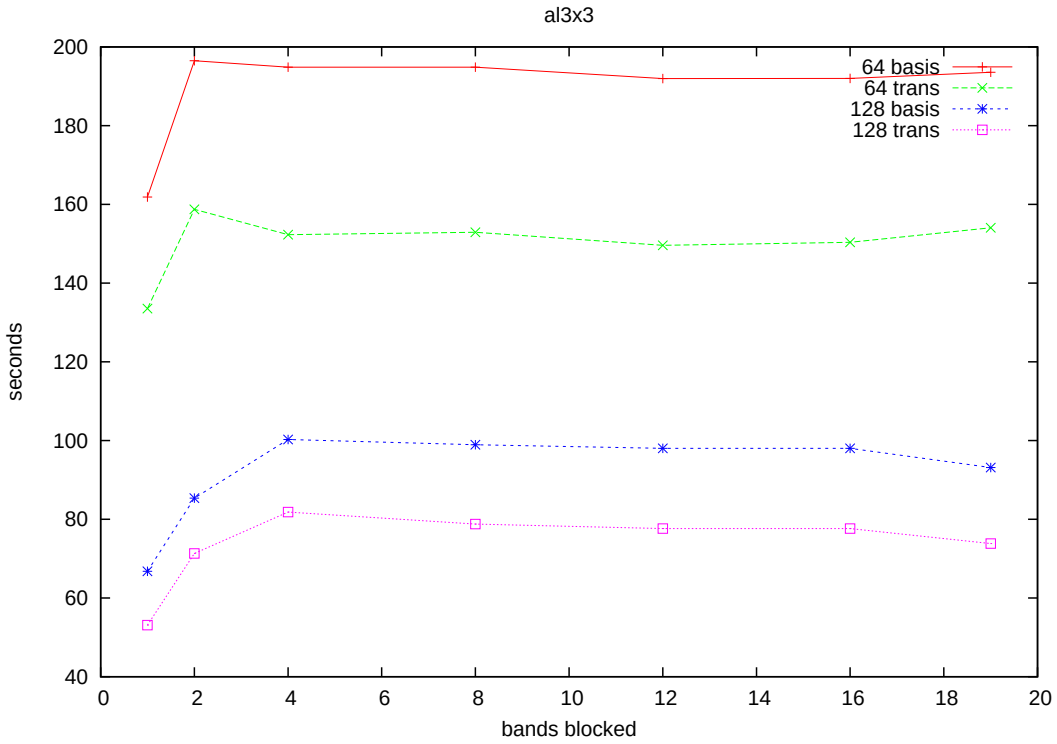

Figure 5: Performance of al3x3 for 64, 128 processors. "basis" indicates time traced from basis_real_recip_reduced_many; "trans" indicates time traced in comms_transpose.

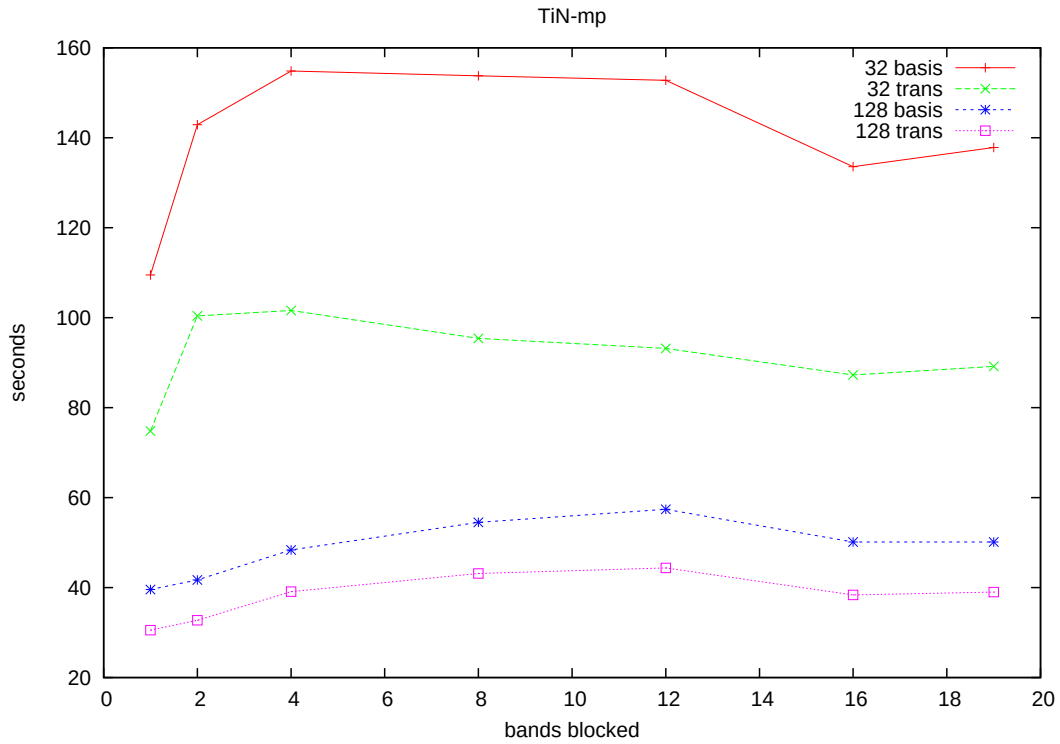

Figure 6: Performance of TiN-mp for 32, 128 processors. "basis" indicates time traced from basis_real_recip_reduced_many; "trans" indicates time traced in comms_transpose.

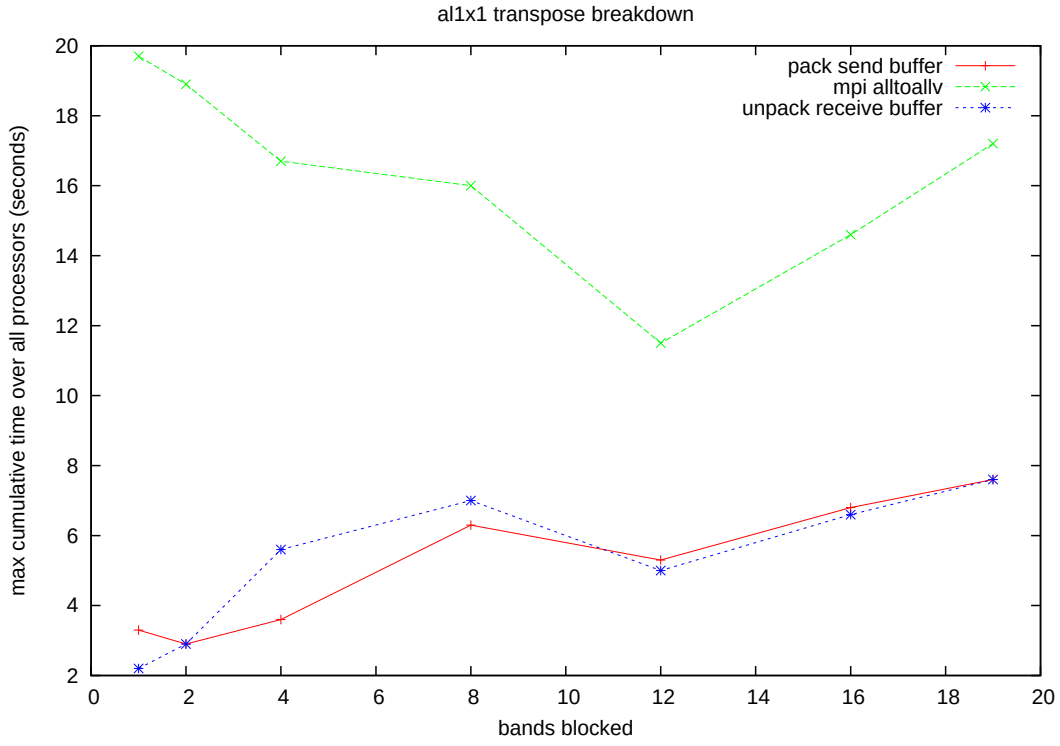

Figure 7: Profile of time spent inside comms_transpose for al1x1, 16 processors.

Table 12 gives the same performance ratios when huge page files are enabled, still comparing against base pages.

Table 12 indicates that using huge pages has the effect of decreasing the number of TLB misses to be comparable with using 1 band with base pages. However, the timing results show no significant performance gains, and the poor L2 and L3 cache figures remain. That enabling huge pages has no effect on performance is reinforced by running with 1 band and huge pages: the ratios of TLB misses for the pack and unpack loops compared to 1 band with base pages are reduced to 0.09 and 0.10, respectively, but the timing ratios are both very close to 1.

This indicates that TLB misses are not the limiting performance factor, and that the aim should be to minimise L2 and L3 cache misses (it is interesting to note that the number of L1 cache misses actually decreases for 5 bands).

	Time	L1 misses	L2 misses	L3 misses	TLB misses
pack	1.07	0.78	5.83	3.65	1.63
unpack	1.26	0.77	3.17	7.67	2.22

Table 11: Performance slowdown factors for 5 bands compared with 1 band

	Time	L1 misses	L2 misses	L3 misses	TLB misses
pack	1.00	0.78	5.73	3.72	1.08
unpack	1.24	0.77	3.15	7.59	1.02

Table 12: Performance slowdown factors for 5 bands with huge page files compared with 1 band with base page files

6.1 Suggestions for further (possible dCSE) work

Change the way in which data is copied to and from shared memory buffers. The present approach relies on existing indexing arrays, which appear to step over memory and make poor use of locality. Some analysis of the current memory access pattern is required and an investigation into more effective packing/unpacking.

Figure 7 shows that bands blocking can improve the performance of MPI_Alltoallv and, given improved memory copies, applying the idea to the rest of the code (i.e. not just basis_real_recip_reduced_many) will multiply the improvement.

Acknowledgments

Thank you to David Tanqueray for supplying the code for managing System V shared memory and to Keith Refson for his input on CASTEP.

References

[1] Bands parallelism in CASTEP http://www.hector.ac.uk/cse/distributedcse/reports/castep/