Improving the scalability of CP2K on multi-core systems
A dCSE Project

I. Bethune
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

September 22, 2010

Abstract

Six months of HECToR dCSE funding was given to implement mixed-mode OpenMP parallelism in CP2K, building on the results of an earlier successful dCSE project. Improved scalability of up to 8 times as many cores was demonstrated for a small benchmark, and a larger, inhomogeneous benchmark was shown to scale up to 9000+ cores. An increase in peak performance of up to 60% was also realised on HECToR Phase 2b. In addition, the performance of the code was studied on three generations of Cray systems - XT4, XT5 and XT6 - and under four different compilers.

Contents

1 Introduction

This report documents the work done under a HECToR Distributed Computational Science and Engineering (dCSE) grant to implement OpenMP parallelism in CP2K. This follows on from an earlier dCSE project which was completed in July 2009. A report on the first project is available online[1], and in the interest of brevity, much of the generic background regarding the code will not be repeated in this document.

6 months of funding were awarded and the work was carried out during the period September 2009 - September 2010 by Iain Bethune, an Applications Consultant at EPCC, the University of Edinburgh.

The project was made possible thanks to the ongoing support from Dr. Ben Slater (University College London, CP2K User), and Dr. Joost Vandevondele *et al* (University of Zurich, CP2K Developers).

1.1 CP2K

CP2K[2] is a freely available atomistic and molecular simulation code, able to study of a wide range of molecular and bulk materials with methods including classical potentials, density functional theory (DFT) and Hartree-Fock methods. It contains an implementation of the Quickstep linear scaling algorithm for DFT, and interested readers are referred to the paper of VandeVondele *et al*[3] or the prior dCSE report[1] for further details of the algorithm and its parallel implementation in CP2K.

1.2 HECToR

HECToR[4] (High End Computinging Terascale Resource) is the UK National Supercomputing Service, and consists of Cray XT and X2 hardware. During this project, the Phase 2a (XT4) and Phase 2b (XT6) components of the system were used.

The XT4 consists of 5664 compute nodes, each containing a 2.3 GHz quad-core 'Barcelona' AMD Opteron processor and 8 GB of main memory, giving 2 GB per core. The nodes are connected to the Cray High Speed Network by a SeaStar2+ router chip. This gives a peak performance of 208 TF, and the system is ranked 26th in the June 2010 Top 500 list[5]. The system has since been halved in size to make way for the newly-installed XT6 system (Phase 2b).

The XT6 consists of 1856 compute nodes, each containing two 2.1 GHz 12-core 'Magny-cours' AMD Opteron processor and 32 GB of main memory, giving 1.33 GB per core. Similarly to the XT4, the nodes are connected to the network via a SeaStar2+ router, although this will be replaced in late 2010 by the upgraded Gemini interconnect. Currently the peak performance is 360 TF, and so the XT6 ranks 16th in the June 2010 Top 500 list.

1.3 Rosa

'Monte Rosa'[6] is a Cray XT5 system at CSCS in Switzerland, consisting of 1844 XT5 compute nodes, each with two 2.4 GHz hexa-core 'Istanbul' AMD Opteron processors and 16GB of main memory, giving 1.33 GB per core. This system provided the ability to test CP2K on a 12-way shared memory node before HECToR Phase 2b became

available mid-way through the project. We are grateful to Prof. Hutter (Univ. Zurich) for providing access to the system for development of CP2K.

1.4 Concurrent development work

It should be noted that concurrently with this work, a new sparse matrix library has been introduced into CP2K - DBCSR (Distributed Block Compressed Sparse Row) - developed by Dr. Urban Borstnik at the University of Zurich. This library is designed from the outset to be highly scalable and particularly suited to density functional theory calculations, taking advantage of the blocked structure of the matrices. It also makes use of OpenMP to distribute work across a shared memory node, with MPI for inter-node communication. Since sparse matrix operations take up a significant fraction of the runtime for the types of jobs we have used for testing and benchmarking (typically 30% or more), the introduction of this library has a strong effect on the performance of the code, and is still under development. Where possible, the effects of this have been removed from the results reported, but the final benchmarks (section 4) do show some improvement due to DBCSR as well as the work of the project.

2 OpenMP Implementation

In order to effectively transform CP2K from pure MPI to mixed-mode MPI/OpenMP we took the approach of applying OpenMP selectively to those routines which are known to dominate the runtime for many types of jobs - in particular the FFT, Realspace to Planewave transfer and Collocate and Integrate routines optimised in the earlier dCSE project. In addition, the dense matrix algebra would be targetted by the use of threaded BLAS libraries available on HECToR (Cray LibSci), and the sparse matrix algebra with the new DBCSR library (Section 1.4). This approach would allow effort to be concentrated in the areas that would yield most benefits.

A mixed-mode code would be expected to scale better than pure MPI for several reasons. Firstly, when running on the same total number of cores, the number of MPI processes can be reduced while still harnessing all the cores using OpenMP threads. This reduces the impact of algorithms which scale poorly with the number of processes, for example, the MPI_Alltoallv collective operation used in the FFT. Secondly, as HPC systems become more increasingly multi-core (for example HECToR has been upgraded from a 2-way node, to 4-way, then 24-way since its installation in 2007), a fully-populated node using only MPI greatly increases contention for access to the network. This effect was particularly seen on HECToR Phase 2b, and in this case we assert that a hybrid programming model fits more closely to the architecture and has resulting performance benefits.

Of course, while there is some gain due to reduced time in communication, this is offset by the fact that an efficient OpenMP implementation of the computational parts of the code is required so that several cores working on shared data using OpenMP threads would take around the same time as the same cores processing the same amount of distributed data as independent MPI processes. As we shall see, this is not always straightforward, although good performance has ultimately been obtained.

When combining OpenMP and MPI, a certain level of thread-safety is required from the MPI implementation. In our case we have adopted a very simple mechanism where all MPI calls are made outside of OpenMP parallel regions, which corresponds to the MPI_THREAD_FUNNELED model in MPI, where MPI calls are guaranteed only to be made by the master thread.

2.1 FFT

In CP2K, the 3D Fast Fourier Transform (FFT) is a key step in the Quickstep algorithm, efficiently transforming from a real-space representation of e.g. electronic density, or potential, to the fourier-space (or planewave) representation and vice versa. The algorithm involves in general 5 steps (for a 2D-distributed planewave grid - the 1D case requires less). This is illustrated in figure 1, and a fuller discussion of the algorithm can be found e.g. in Jagode 2006[7].

1. Perform 1D FFT of local data (for example in the Z-direction)
2. Transpose the data using MPI_Alltoallv to bring the Y-dimensional data local
3. Perform 1D FFT of local data (Y-direction)
4. Transpose again using MPI_Alltoallv to bring the X-dimensional data local

5. Perform 1D FFT of local data (X-direction)

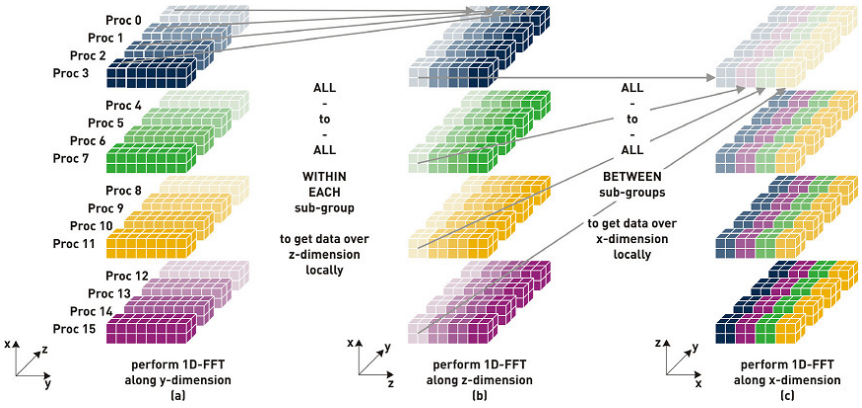

Figure 1: Stages of a 3D-distributed 3D Fourier Transform, reproduced from [7]

There are two parts of this process that we can parallelise using OpenMP - firstly, the 1D FFT itself, and secondly the packing and unpacking of the buffers used by the MPI_Alltoallv.

For the 1D FFT, each MPI process has to perform a block of M FFTs each of length N. The actual FFT is performed by a call to an FFT library (typically FFTW3[8]), but three different approaches were tested on how to divide up the work between the available threads.

1. Using FFTW threading - the FFTW3 library has inbuilt threading capability, and it is possible to plan an FFT for a given number of threads. Using this method, the number of OpenMP threads available is found using `omp_get_num_threads()` and passed to the FFTW planner using `fftw_plan_with_nthreads()`. Then when the plan is executed the specified number of threads are used.

2. Parallel loop over single length-N FFTs - here we generate an FFTW plan for a single length-N FFT, running on a single thread. The M FFTs are then performed in a loop, which is parallelised using OpenMP. This results in each thread performing (approximately) the same number of FFTs.

3. Parallel blocks of M/nthreads length-N FFTs - similar to the previous approach this requires creation of multiple plans (actually only 1 or 2 are required to cover all cases). Each thread then executes one of the plans starting at an appropriate offset into the data array, and FFTs a set number of rows of data. Some care must be exercised here to ensure that each thread always starts its FFT on an

element which is aligned on a 16-byte boundary, due to the requirements for SSE vectorization, but this can be achieved by dividing the array up in pairs of rows if there is an odd number of elements in each row and single precision arithmetic is being used.

As it turns out, the FFTW threading is not particularly efficient, giving only a maximum speedup of 1.7x when using 4 threads on a node compared to only a single thread (on HECToR Phase 2a). The second and third methods perform similarly, with the third slightly better due to the reduced overhead of executing many small FFTs compared to a small number of larger blocks of FFTs. With this method we see a speedup of 2.9x on 4 threads. Part of the reason for not seeing the full 4x speedup in the case is that as well as being computationally intensive, the FFT also requires memory bandwidth as the data is streamed from memory into cache. Adding more threads gives access to more FLOPs but not any additional memory bandwidth as access to memory is shared between all 4 cores. Of course, this is also a problem for the MPI implementation, except the same total bandwidth must be shared between four processes, rather than four threads.

In addition to the FFT itself, OpenMP was also used to parallelise the packing of the MPI buffers for the transpose. In this case some of the loops to be parallelised are perfectly nested and so can be coalesced into a single loop using the OpenMP `collapse` clause, which increases the iteration space and therefore allows more parallelism to be exploited. This is particularly advantageous where one of the loops is over the number of MPI processes, which may be small (and will be made smaller as more threads are used). However, this feature is only in OpenMP version 3.0, which is not yet supported by all compilers (notably the Pathscale compiler), and so a new preprocessor macro `__HAS_NO_OMP_3` is used to turn this feature off for compilers that do not support it.

The final OpenMP-parallel FFT code performs well in benchmarks, proving to be giving the same performance at low core counts, but scaling much better. As shown in figure 2 the peak performance for the 3D FFT of a 125^3 grid is at 512 cores for the MPI-only code, 1024 cores when using 2 threads per task, and 2048 cores when using 4 threads per task. Also, the peak performance of the 4-threaded version is 2.8 times higher than that of the MPI-only version, so the increases scalability also delivers real improvements in time-to-solution.

Figure 3 shows the equivalent results on Rosa. Here we see a similar trend, however, there is no additional benefit from using more than 6 threads per MPI task. This reflects the fact that the 12-way SMP node on the Cray XT5 is in fact two hexacore chips connected together via a hypertransport bus, and therefore there is a significantly higher penalty for accessing memory in the memory banks attached to the other processor. This results is any benefit of the extra cores in terms of FLOPs available being nullified by the increased memory access latency, and contention in the cache heirarchy. Nevertheless, using 2 MPI processes per node, each with 6 threads appears to be a performant solution, at least for the 3D FFT. It is also worth noting that the FFT as a whole scales worse on Rosa than on HECToR Phase 2a. This is due to the increased number of cores (12 c.f. 4) on a node which share a single SeaStar2+ network interface. Especially in the regime where there are many processes communicating, and the message size is small, the message throughput rate of the SeaStar can become a limiting factor. We will see this effect even more strongly on the 24-core nodes of HECToR Phase 2b.

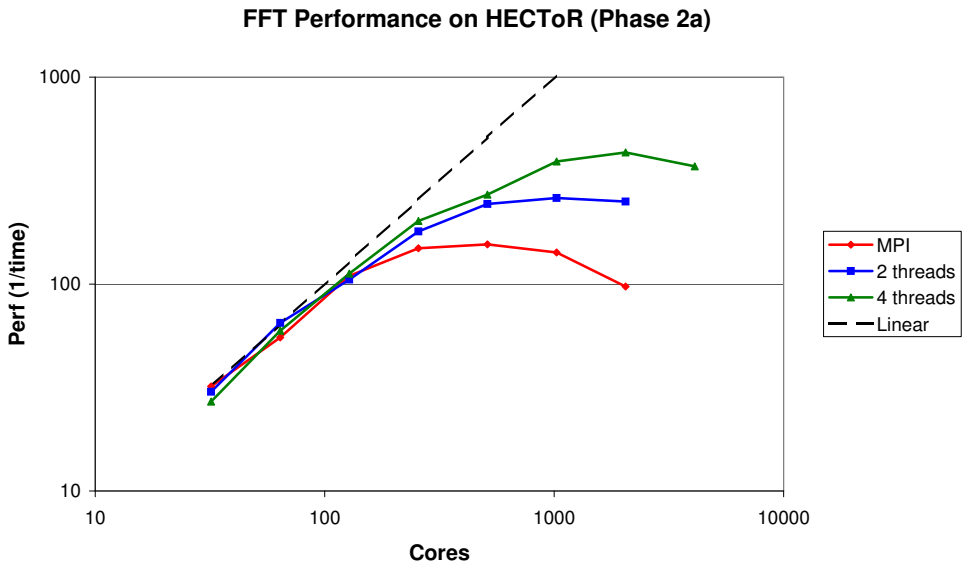

Figure 2: Performance of 125^3 FFT on HECToR Phase 2a

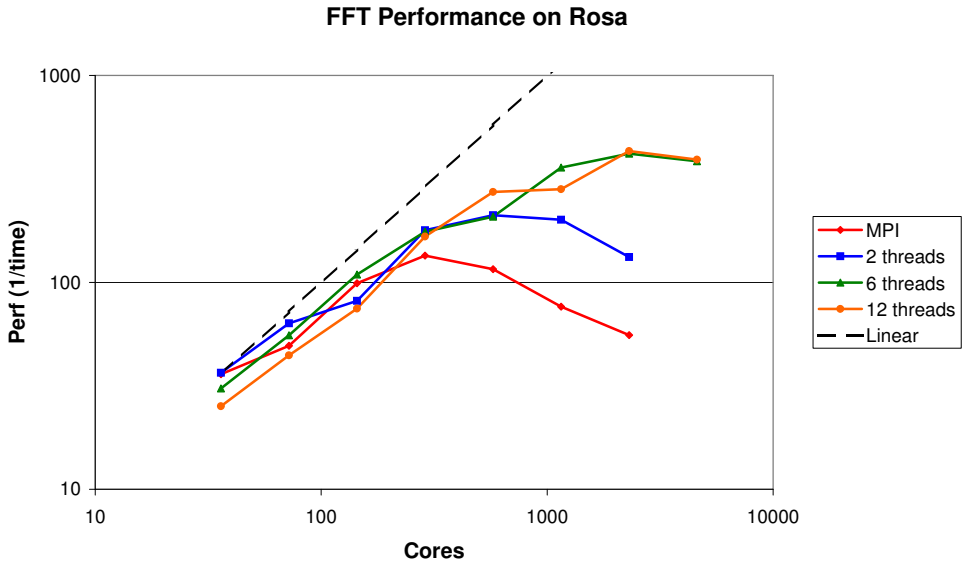

Figure 3: Performance of 125^3 FFT on Rosa

2.2 Realspace to Planewave transfer

Realspace to Planewave transfer (or `rs2pw`) is responsible for the halo swap and alltoall communication necessary for transforming between the realspace (replicated or 1D, 2D, 3D distributed) grids and the corresponding (1D or 2D distributed) planewave grids

in prepartion for th FFT which transforms the data on the planewave grids into its reciprocal representation. Much of the time spent in this routine is in communication, but there are several loops that can be parallelised using OpenMP. Like the FFT, MPI buffer packing was parallelised, with each thread packing a portion of the buffer with data from the grids and vice versa. Many of these buffer packing operations are conveniently described by Fortran90 array syntax e.g. (pseudocode)

```
send_buf(:) = rs % r (lb:ub)
```

This type of operation can be easily parallelised using the Openmp `workshare` directive, which ensures each thread is responsible for copying a particular subset of the array. However, the GNU implementation of OpenMP (gfortran is a popular compiler for CP2K) prior to the recently released version 4.5.0 implements `workshare` in the same way as `single` i.g. serialising the array operation on a single thread. While technically standards-compliant this obviously does not give good performance, so these operations were manually parallelised by defining a given range for each thread based on its thread ID and the array bounds.

Another loop that was parallelised was responsible for calculating the amount of data to be sent to and from each process. This took the form of a nested loop over pairs of processes (see below). However, a conditional inside the loop would only allow the loop body to execute if one of the two indices was the ID of the process in question. So out of the P^2 loop iterations, only $2P$ actually did anything useful. This posed a problem for OpenMP parallelisation, as most iterations of the outer loop would take the same amount of computation (only a single iteration of the inner loop would execute), but one iteration would take P time longer as the entire inner loop would execute.

```
do i = 1, group_size
   do j = 1, group_size
      if (i == my_id || j == my_id) then
         <do stuff...>
      end if
   end do
end do
```

By transforming this nested loop into two independent loops from 1 to `group_size` not only did it become much easier to efficiently parallelise with OpenMP, it also significantly reduced the amount of time wasted in needless loop iterations even when threads were not used.

Finally, after observing that the following alltoallv is fairly sparse (that is, only a few pairs of processes communicate, rather than an true all-to-all), this was replaced with non-blocking point-to-point MPI operations between the pairs of processes that actually have data to transfer. This gave a speedup of $\tilde{5}$-10% over the existing MPI_Alltoallv implementation.

The result of all these optimisations can be seen on the graph below (figure 4). Clearly the algorithm itself does not scale as well as the FFT. However, the impact that the loop restructuring and sparse alltoall have is clear from the difference between the original and 'New MPI' results, causing the routine to scale out to 1152 cores for this problem size

(125^3 realspace grid). Again, using 2 or 6 threads per MPI task gives some improvement both in the raw performance and also the scalability of the code. The best performace overall is achieved using 768 MPI tasks, each with 6 threads. However, this is only twice as fast as the pure MPI code using only 288 cores in total. Nevertheless, by removing the steep drop-off beyond this point seen in the with the original MPI implementation, this will help the code as a whole to scale, as seen in section 4.

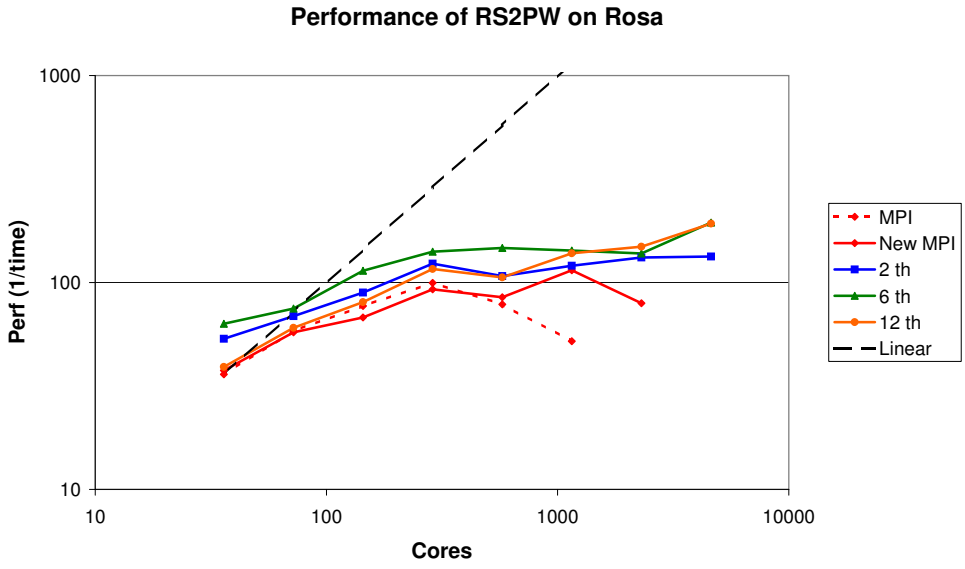

Figure 4: Performance of RS2PW on Rosa

2.3 Collocate and Integrate

The third major area to be parallelised was the Collocate (`calculate_rho_elec`) and Integrate (`integrate_v_rspace`) routines. These take a list of Gaussians (products of atom-centred Gaussian basis functions) and map these to regular grids, prior to Fourier Transformation, and vice versa. Again, see [1] for how this step makes up a key part of the Quickstep dual-basis algorithm.

Taking the collocate case as an example, the input to the routine is a list of 'tasks' which correspond to the gaussians to be mapped. These tasks are represented as large integers, which encode (among other things) the IDs of the relevant atoms, and the grid level to which the task is to be mapped. The atomic IDs are used to index into the density matrix `deltap` to retrieve the coefficients of the associated primitive Gaussian functions. The product is then calculated and is summed onto the assigned grid level. Integration is precisely the reverse - for each task, a region of the grid is read, and this data is then stored in the relevant location in the matrix (specified by the task parameters).

Although similar, both of these procedures present different challenges to efficient OpenMP parallelisation. For collocation, the matrix is read-only, so can be shared by all threads, but the grids are writable, and moreover, in general the Gaussian products may

overlap on a given grid level so we need to ensure that writes to the grids from multiple threads are summed correctly, rather than allowing a data race condition to occur. For integration, the grids are read-only, but this time the matrix must be protected so that a single block of the matrix is only updated by a single thread. The matrix is in fact stored in DBCSR format, which makes provision for threading by allowing each thread to update a 'work matrix' - a working copy of the matrix. Additions or modifications to the work matrix are combined in a process called 'finalization'. For correct finalization, each work matrix should contain a different subset of blocks of the whole matrix, so we require that each thread only works on tasks corresponding to a particular atom pair before the matrix is finalized.

Most of the difficulties can be overcome by pre-processing the task list at the time it is created to determine the indices at which the tasks for each grid level and each atom pair start and end (the list is already sorted in this order). Thus we can transform the existing loop:

```
DO ipair = 1, SIZE(task_list}
  <process each task>
END DO
```

into a parallel loop:

```
DO ilevel = 1, ngrid_levels
!$omp parallel do
  DO ipair = 1, task_list%npairs(ilevel)
    DO itask = task_list%taskstart(ilevel,ipair), task_list%taskstop(ilevel,ipair)
      <process each task>
    END DO
  END DO
!$omp end do
END DO
```

This preserves the high-level ordering of the tasks (by grid level) which retains the benefit of keeping only one grid level (which could be several megabytes in size) in cache at any given time.

To correctly perform the integration, we need to add the creation of work matrices at the start of each `ilevel` loop, and finalize the matrix at the end of each loop. Thus the threads can partition the loop over atom pairs arbitrarily at each grid level (and the number of tasks per pair can vary over the grid levels), while maintaining the condition that each matrix block is only updated by a single thread before each finalization.

For collocation, the situation is slightly more complicated, due to the fact that we need to be able to sum data onto potentially overlapping regions of the grids. This is solved by allocating a copy of the grid for each thread (local grids or `lgrids` in the code), and then performing a reduction in parallel at the end of each `ilevel` loop. During each loop, a thread writes the data corresponding to its set of tasks to a private region of the lgrids (figure 5).

At the end of the loop, the grids are reduced in parallel back onto the original realspace grid. Two methods were implemented for this reduction. In the first method,

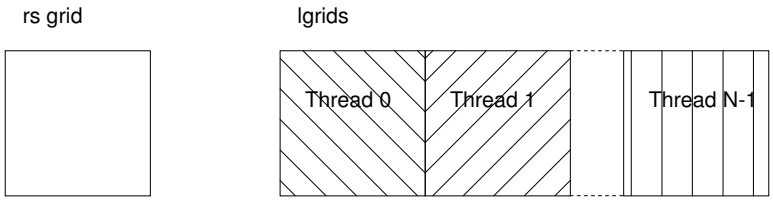

Figure 5: Local grids (lgrids) written to be multiple threads during collocation of Gaussian products

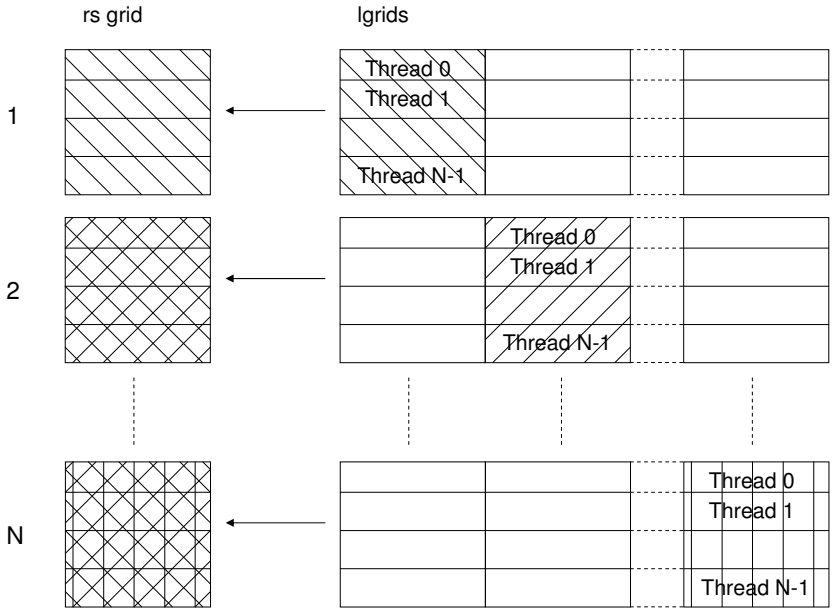

Figure 6: Parallel lgrid reduction (first method)

each thread sums the contributions from each threads lgrid back into a single region of the rsgrid (figure 6).

This has the advantage that every thread can operate without synchronisation since it writes to a disjoint part of the grid. However, it has to read from data that was previously written by other threads, which can be relatively costly as this data may be in cache on another core. Because of this, a second method was implemented which performs somewhat better. In this case (figure 7) each thread sums a region of its lgrid into the rsgrid. The regions are distributed cyclically, so at each step, only a single thread writes to a given region of the rsgrid. Synchronisation is required between each step, but this cost is outweighed by the benefit of a reduced number of reads of remote data, and so this method was implemented in the current CVS version of CP2K.

It is not possible to benchmark the collocate and integrate routines directly as for the FFT and RS2PW, however it is easy to extract the time spent in these routines from short runs of a whole-code benchark H20-64. This is a short molecular dynamics run with 64 water molecules is a 12 Åcubic unit cell (as used in the previous dCSE project [1]). The results of running on 36 nodes of HECToR Phase 2b with a single MPI

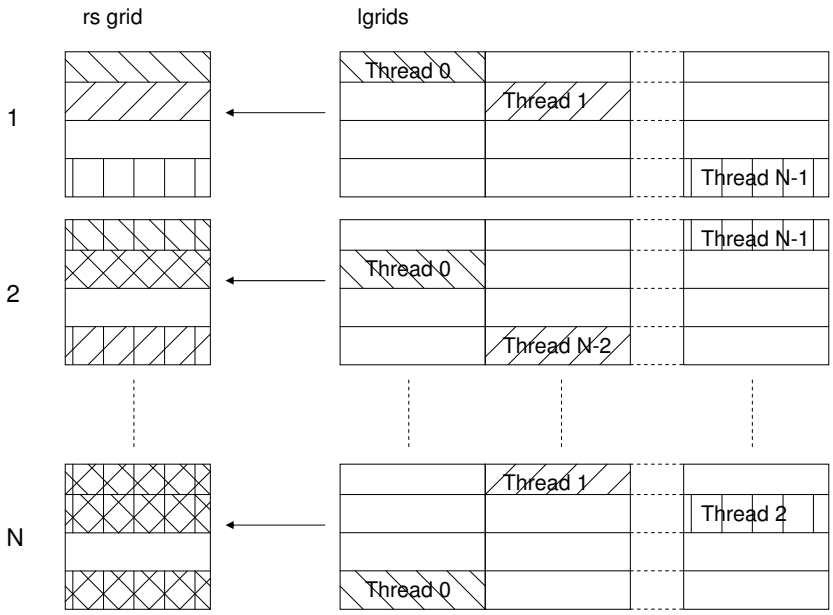

Figure 7: Parallel lgrid reduction (improved)

Threads	1	2	3	4	6	12	24
`calculate_rho_elec`	20.4	11.1	7.9	6.5	5.2	7.5	20.1
Speedup	1	1.8	2.6	3.2	3.9	2.7	1
`integrate_v_rspace`	22.5	11.7	7.9	6.1	4.2	2.8	2.2
Speedup	1	1.9	2.9	3.7	5.5	8.2	10.4

Table 1: Times (in seconds) and speedup for collocate and integrate on HECToR Phase 2b

task per node and a varying numbers of OpenMP threads are shown in table 1. The table shows that integration scales well as the number of threads is increased since the memory heirarchy is used effectively and there is little data sharing except at the matrix finalizatio, which only occurs once per grid level. However, with collocation, due to the expensive grid reduction step, speedup is only achieved up to 6 cores (a single processor) due to the aforementioned costs of remote memory access and synchronisation. It may be possible to replace this scheme with a tree-based algorithm, aiming to reduce the number of copies from one processors memory bank to anothers, but this was not investigated within the time available for the project.

2.3.1 `distribute_matrix`

As mentioned earlier, the matrix used in the collocate and integrate routines is distributed, so it is required that before tasks are are mapped, the required matrix blocks are gathered on each MPI process, and conversely, after integration, each matrix block must be scattered back to the process it originated on. This is done by the subroutine `distribute_matrix` which simply traverses the task list, packs matrix blocks into an

Region	Before	After	Speedup
Packing Send Buffer	2.25	1.60	29%
Unpacking Recv Buffer	3.00	2.02	33%
Total user time	7.07	5.04	29%

Table 2: Times (in seconds) and speedup for distribute_matrix (using 6 threads on Rosa)

MPI send buffer, and distributes them using MPI_alltoallv. The corresponding recieved tasks are then unpacked into the correct position in the local matrix.

While a significant proportion of this routine is taken up by the communication, the buffer packing step can be accelerated using OpenMP. This is slightly complicated by the fact that the packing loops maintain a running count of the number of data elements sent to each process, so the loop is not easily parallelisable as is.

```
DO i = 1, SIZE(atom_pair_send)

  l = <process to send this block to>
  <pack buffer using send_disps(l) + send_sizes(l) as offset>
  send_sizes(l) = send_sizes(l) + <size of block packed>

END DO
```

We need to ensure that the data for a single process is packed only by one thread, thus avoiding a data race on elements of the `send_sizes` array. To do this, we first traverse the array of atom pairs (in fact this is already done earlier to calculate the total size required for the send buffer), and find the number of pairs (and displacement in the list) for each processor. Thus we can now transform the loop into a parallel form:

```
!$omp parallel do
DO l = 1, group_size
  DO i = 1, send_pair_count(l)

    <pack buffer using send_disps(l) + send_sizes(l) as offset>
    send_sizes(l) = send_sizes(l) + <size of block packed>
  END DO
END DO
!$omp end parallel do
```

In order to measure the speedup (excluding the communication time, which remains constant), the code was instrumented with CrayPAT regions. The results are shown in table 2. Clearly the speedup is not as large as might be hoped, however the main limitation here is memory bandwidth, which does not increase strongly with the number of threads used. It is also suspected that this loop is significantly imbalanced (particularly as the region of the buffer that is sent to self is typically much larger than for other processes). Investigation into an appropriate OpenMP schedule might improve this somewhat, but was not investigated in the time available.

Threads	1	2	3	4	6	12	24
`pbe_lda_eval`	7.98	4.05	2.73	2.08	1.42	0.75	0.45
Speedup	1	1.97	2.92	3.84	5.62	10.64	17.73

Table 3: Times (in seconds) and speedup for PBE evaluation on HECToR Phase 2b

2.4 Functional Evaluation

One subroutine that was not initially planned to be parallelised, but began to show up in the CP2K timing report as other areas of the code were parallelised was the evaluation of the correlation functional. In this case, only the PBE functional [9] was parallelised, but the method should generalise to the other implemented functionals easily if required.

The bulk of the functional evaluation is done as a single loop over the points on the real-space density grids. At each point, the (complicated) calculation of the the functional is performed, and the result written onto a corresponding point on the derivative grids. This loop is trivially parallel since each iteration is entirely independent, so we see very good OpenMP efficiency (93% efficiency with 6 threads, and 74% using all 24 cores on the node), as shown in table 3.

3 Compiler Comparison

An additional objective of the project was to evaluate the performance of the compilers available on the Cray XT for CP2K. At the start of the earlier CP2K dCSE project[1] the Pathscale 3.1 compiler was found to give around 5% greater performance than the PGI (8.0.2) or gfortran (4.3.2) compilers. Around 2 years after these results, new versions of all three compilers are available, as well as the new Cray Compiler Environment (CCE). In addition, the ability of the compilers to handle the mixed-mode OpenMP code was also evaluated.

The H20-64 benchmark, running on 72 cores (6 nodes) of the Cray XT5 'Rosa' was used for this comparised. For this configuration, less than 30% of the runtime is spent in communication, so the performance of the compiled code is strongly dependent on the compiler's ability to generate a well-optimised binary. The results for the MPI-only code are shown in table 4. In contrast to the previous results, the gfortran compiler now produces results that are in fact slightly better than either Pathscale or PGI. Further details of each of the compilers are below:

Compiler	Optimisation flags	Time(s)
PGI 10.6.0	-fastsse	143.7s
Pathscale 3.2.99	-O3 -OPT:Ofast -OPT:early_instrinsics=ON -LNO:simd=2	139.8s
gfortran 4.4.4	-O3 -ffast-math -funroll-loops -ftree-vectorize	136.1s
crayftn 7.2.4	-O 2 -O ipa1	184.7s

Table 4: Comparison of compilers on Rosa, using bench_64

- PGI

  Although the PGI compiler gives a reasonably well-performing executable, it does suffer from some drawbacks. In particular, it appears to have some difficultly compiling several parts of the code, and in order to achieve correctness for this benchmark 15 seperate source files had to be compiled without optimisation. Even in this case, around 25% of the regression test suite still failed to give correct results. The mixed-mode OpenMP build was also a failure as it generated segfaults at runtime.

- Pathscale

  The Pathscale compiler is fairly robust for compiling the MPI-only code, and still gives good performance. However, it was not possible to build a working mixed-mode executable. It is hoped that when the Pathscale 3.3 compiler is released this may resolve some of the OpenMP issues as it contains a new, OpenMP 3.0 compliant, implementation.

- Gfortran

  Gfortran is now the compiler of choice for CP2K. It is well tested by the developer and user community, and now gives performance on a par with, or exceeding the commercial compilers tested. Furthermore, it was the only compiler capable of producing a working mixed-mode executable. The centrally installed CP2K executables on HECToR are now compiled with gfortran.

- CCE

  The Cray fortran compiler is able to successfully compiler CP2K since version 7.2.4. However, the performance is much poorer (35% slower than gfortran), mostly due to poor optimisation of the collocate and integrate kernel routines. At the time of writing, it is not possible to run a mixed-mode code succesfully without disabling some features of DBCSR. However, Cray have been responsive to these issues and it is expected that they will be resolved in a future release of the compiler.

4 Benchmark Results

In this section we present some benchmark results on all three Cray systems, HECToR Phase 2a (figure 8), Rosa (figure 9), and HECToR Phase 2b (figure 10). Again, it should be stated that as well as the work reported here, these benchmarks also include the results of other development on CP2K, in particular the DBCSR library.

Firstly it is clear from comparing the three systems that as the number of cores in a node increases (4 in HECToR 2a, 12 in Rosa, 24 in HECToR 2b), the scalability of the code decreases, with the maximum performance of the pure MPI code being achieved on 256, 144, and 144 cores respectively. We note, however, that the Seastar 2+ network interface (used by all three systems), is soon to be replaced by the new Gemini interconnect on HECToR Phase 2b. This will bring higher message throughput, and also especially improved performnace for small messages. We expect this to give scalability similar to or better than the XT4.

Secondly, we see that the performance of the MPI-only code has improved by 40-70% at around 1000 cores. While this has not had the effect of allowing the MPI code to scale any further, it will help improve performance at higher core counts for larger problems.

Thirdly, we see that using threads does indeed help to improve the scalability of the code. Suitable numbers of threads to use are between 2 and 6 (the number of cores in a single processor), depending on the balance between performance for low core counts, and the desired scalability. The overall peak performance of the code has been increased by about 30% on HECToR Phase 2a and Rosa when using mixed-mode OpenMP, and by 60% on HECToR Phase 2b, due to the fact that it reduces the number of messages being required to pass through each SeaStar dramatically.

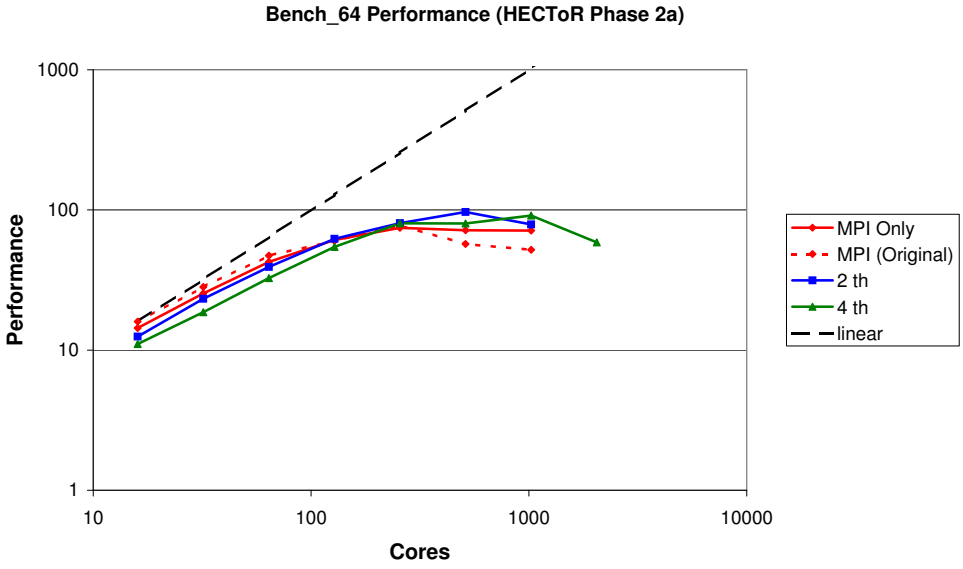

Figure 8: Performance of bench_64 on HECToR Phase 2a

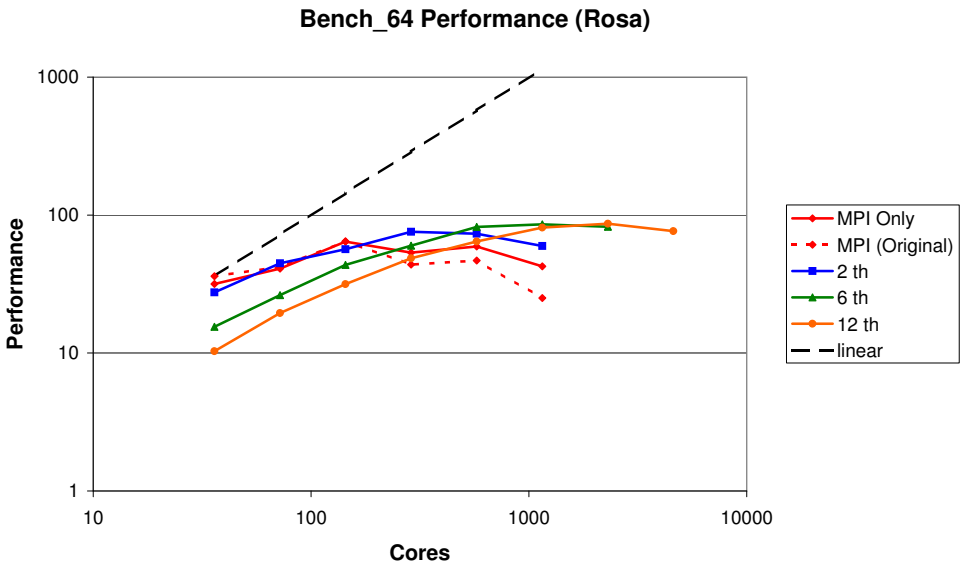

Figure 9: Performance of bench_64 on Rosa

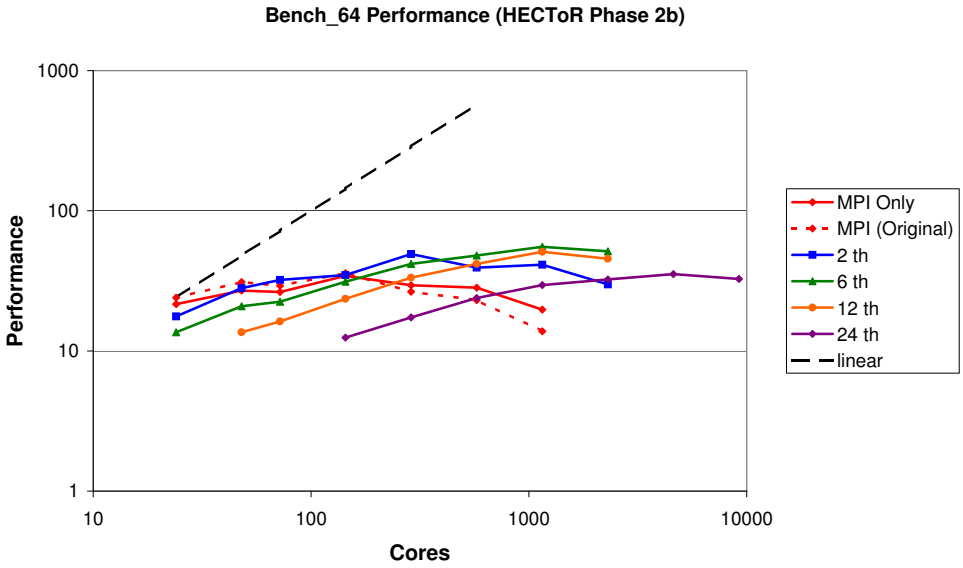

Figure 10: Performance of bench_64 on HECToR Phase 2b

Figure 11 shows the performance of the W216 benchmark on Rosa. W216 is a larger system than bench_64, having 3 times as many atoms, and a unit cell of 20 times the volume. Here the benefits of using a mixed-mode approach are shown very clearly. Using only a single MPI task per node, with 12 OpenMP threads, the maximum performance

achieved is 2.5 times that of the pure MPI code. This does come at a premium in terms of efficiency however, as to achieve the speedup 16 times as many cores are used. However, using 2 threads per task it is possible to achieve a genuine speedup of 20% on 576 cores.

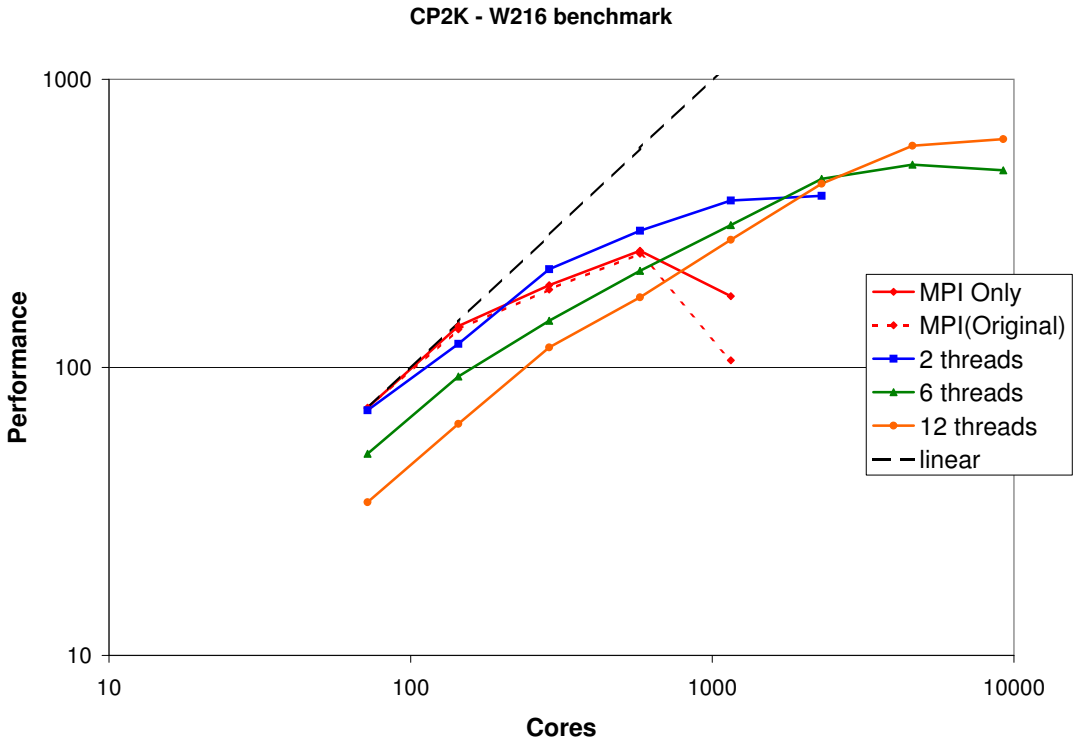

Figure 11: Performance of W216 on Rosa

5 Conclusion

This project has successfully implemented OpenMP parallelism in many key routines of CP2K, allowing the code to be run effectively in a mixed-mode OpenMP/MPI manner. We believe this programming model is very appropriate for applications on modern multi-core HPC systems such as HECToR. This is demonstrated by improvements in the peak performance of the code as well as improved scalability compared to what is possible using only MPI. These benefits have already been delivered for HECToR users, as the centrally installed version of the code has been updated to include these changes, which are also available via the CP2K CVS repository.

Looking to the future, larger systems of thousands of atoms are now of interest to researchers using CP2K, and such calculations are mainly dominated by sparse linear algebra operations using the DBCSR library. We have been successful in obtaining a further six months of dCSE funding which will allow us to build on this work by optimising the usage of both MPI and OpenMP within DBCSR, delivering even better performance for the CP2K user community.

References

[1] Improving the performance of CP2K on HECToR: A dCSE Project, Iain Bethune, 2009, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k/cp2k_final_report.pdf

[2] CP2K website, http://cp2k.berlios.de

[3] QUICKSTEP: Fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassasing and J. Hutter, Comp. Phys. Comm. 167 (2005) 103-128

[4] HECToR: UK National Supercomputing Service, http://www.hector.ac.uk

[5] TOP 500 List - June 2010, http://www.top500.org/list/2010/06/100

[6] CSCS Swiss National Computing Centre, http://www.cscs.ch

[7] Fourier Transforms for the BlueGene/L Communications Network, H. Jagode, 2006, http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0506/hjagode.pdf

[8] The Design and Implementation of FFTW3, M. Frigo and S. Johnson, Proceedings of the IEEE 93 (2), 216.23 1 (2005).

[9] Generalized gradient approximation made simple. J.P. Perdew, K. Burke, and M. Ernzerhof, Phys. Rev. Lett., 77 (1996) 3865-68

A Detailed Timings for Figures

This appendix contains the measured runtimes of the various benchmarks used to generate the figures throughout the report.

Cores	32	64	128	256	512	1024	2048	4096
MPI Only	37.7	21.9	11.0	8.1	7.8	8.5	12.4	
2 threads	40.0	18.6	11.5	6.7	4.9	4.6	4.8	
4 threads	44.7	20.3	10.7	6.0	4.4	3.1	2.8	3.3

Table 5: Runtimes of 125^3 FFT on HECToR Phase 2a (Figure 2)

Cores	36	72	144	288	576	1152	2304	4608
MPI Only	40.4	29.4	14.7	10.8	12.5	19.0	26.1	
2 threads	39.7	22.9	17.8	8,1	6.9	7.2	11.0	
6 threads	47.3	26.2	13.3	8.3	7.0	4.0	3.5	3.8
12 threads	57.6	32.3	19.5	8.7	5.3	5.1	3.4	3.7

Table 6: Runtimes of 125^3 FFT on Rosa (Figure 3)

Cores	36	72	144	288	576	1152	2304	4608
MPI	21.2	13.1	9.9	7.7	9.7	14.7		
New MPI	20.3	13.3	11.3	8.2	9.0	6.7	9.6	
2 threads	14.2	11.1	8.5	6.2	7.1	6.3	5.8	5.7
6 threads	12.1	10.2	6.7	5.4	5.2	5.3	5.5	3.9
12 threads	19.5	12.6	9.5	6.6	7.2	5.5	5.1	4.0

Table 7: Runtimes of RS2PW on Rosa (Figure 4)

Cores	16	32	64	128	256	512	1024	2048
MPI (Original)	293	166	99	78	60	82	90	
MPI Only	326	185	110	77	63	65	66	
2 threads	376	202	120	75	58	48	60	
4 threads	424	251	144	86	58	58	51	80

Table 8: Runtimes of bench_64 on HECToR Phase 2a (Figure 8)

Cores	36	72	144	288	576	1152	2304	4608
MPI (Original)	151	129	84	124	116	217		
MPI Only	172	133	85	102	92	128		
2 threads	198	122	96	72	74	91		
6 threads	350	207	125	91	66	63	66	
12 threads	527	279	172	112	84	67	63	71

Table 9: Runtimes of bench_64 on Rosa (Figure 9)

Cores	24	48	72	144	288	576	1152	2304	4608	9126
MPI (Original)	190	147	156	127	172	198	330			
MPI Only	211	169	173	133	155	161	232			
2 threads	259	162	142	131	93	116	111	153		
6 threads	335	219	203	146	109	95	82	89		
12 threads		335	281	194	137	109	89	100		
24 threads				366	263	191	155	141	129	140

Table 10: Runtimes of bench_64 on HECToR Phase 2b (Figure 10)

Cores	72	144	288	576	1152	2304	4608	9216
MPI (Original)	5728	3041	2214	1662	3897			
MPI Only	5694	2964	2137	1623	2335			
2 threads	5810	3419	1881	1383	1086	1047		
6 threads	8230	4439	2842	1907	1323	914	816	854
12 threads	12113	6477	3515	2356	1487	950	701	665

Table 11: Runtimes of W216 on Rosa (Figure 11)