Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

High Performance MP2 for Condensed Phase Simulations

Ruyman Reyes[a], Iain Bethune[a]*

[a]*EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK*

Abstract

This report describes the results of a PRACE Preparatory Access Type C[b] project to optimise the implementation of Møller-Plesset second order perturbation theory (MP2) in CP2K, to allow it to be used efficiently on the PRACE Research Infrastructure. The work consisted of three stages: firstly serial optimisation of several key computational kernels; secondly, OpenMP implementation of parallel 3D Fourier Transform to support mixed-mode MPI/OpenMP use of CP2K; and thirdly - benchmarking the performance gains achieved by new code on HERMIT for a test case representative of proposed production simulations. Consistent speedups of 8% were achieved in the integration kernel routines as a result of the serial optimisation. When using 8 OpenMP threads per MPI process, speedups of up to 10x for the 3D FFT were achieved, and for some combinations of MPI processes and OpenMP threads, overall speedups of 66% for the whole code were measured. As a result of this work, a proposal for full PRACE Project Access has been submitted.

Application Code: CP2K

1. Introduction

CP2K

CP2K [1] is an open-source program written in Fortran 95 for atomistic and molecular simulation. The code is most well-known for its implementation of the QUICKSTEP [2] linear scaling DFT algorithm, but has been designed in an extensible and efficient manner thus allowing users a wide choice of simulation methods from classical, semi-empirical and DFT to Hartree-Fock and the recently added Møller-Plesset second order perturbation theory (MP2) [3]. CP2K consists of over 800,000 lines of code and is developed by a distributed team of researchers from across Europe.

Tödi

'Tödi' is a Cray XK6 system with hybrid GPU/CPU capabilities, installed at CSCS (the Swiss National Supercomputing Centre) [4]. It features 176 GPU nodes, each one equipped with a 16-core AMD 'Interlagos' Opteron CPU, 32 GB DDR3 memory and one NVIDIA Tesla X2090 GPU with 6GB GDDR5 memory and 96 non-GPU nodes, each with the same CPU but no GPU for a total of 4352 cores in the entire machine

For this work we have used Todi extensively as development and testing platform, although we have not made use of the GPU capabilities of the system. Access to this system was kindly provided by Prof. Jürg Hutter and Prof Joost VandeVondele.

* Corresponding author. *E-mail address:* ibethune@epcc.ed.ac.uk

[b] Preparatory Access Type C provides PRACE users with access to PRACE RI resources for code development and optimisation with the support of PRACE experts. Full details of how to apply can be found at http://www.prace-ri.eu/Call-Announcements

HERMIT

HERMIT, one of the PRACE Tier-0 HPC systems, is a Cray XE6 system and consists of 3552 compute nodes. Each node contains two Dual Socket AMD 'Interlagos' CPUs running at 2.3 GHz, with 16 cores each making a total of 32 cores per node and 113664 cores in the entire system. Hermit is installed at the HLRS [5], and access was provided via the PRACE Preparatory access project 2010PA0723, submitted by Prof. Joost VandeVondele.

2. Profiling of MP2 Calculations

Throughout the project we have used a test system `NH3_32_bulk` which computes the MP2 energy of 32 Ammonia molecules arranged in a cubic cell of side 10Å (see Appendix A). Periodic boundary conditions and a TZV2P basis set are employed. The calculation takes just over 15 minutes on 2400 cores. Initial analysis of the routine timings output by CP2K (Table 1) showed that most of the time was spent in the `integrate_v_rspace` routine, which computes the matrix elements corresponding to a potential stored on real-space multigrids.

SUBROUTINE	CALLS	SELF TIME(S) AVERAGE	SELF TIME(S) MAXIMUM	TOTAL TIME(S) AVERAGE	TOTAL TIME(S) MAXIMUM
CP2K	1	0.070	0.192	1040.238	1040.251
mp2_gpw_compute	1	40.766	185.838	934.151	934.293
integrate_v_rspace	97	716.798	742.346	731.760	757.871
cp_dbcsr_multiply_d	195	0.002	0.002	123.330	131.867

Table 1 Breakdown of timings by routine for CP2K MP2 calculation

More detailed analysis using the Cray Performance Analysis Toolkit (CrayPAT) showed that in this case, much of this time was spent in the `xyz_to_vab` and `integrate_core_*` routines (Table 2). `xyz_to_vab` prepares for the integration operation by transforming from a Cartesian representation of a Gaussian basis function to a spherical polar representation. `integrate_core_*` carries out the integration of the Gaussian products, with a specialised version for each case of the total angular momentum quantum numbers of the two basis functions.

Samp%	Samp	Imb. Samp	Imb. Samp%	Group Function
9.9%	6434.3	400.7	5.9%	xyz_to_vab$integrate_pgf_product_rspace$qs_integrate
7.4%	4814.0	312.0	6.1%	integrate_core_2_
7.0%	4547.9	396.1	8.0%	dgemm_kernel
6.1%	3992.6	357.4	8.2%	integrate_core_1_
4.4%	2836.4	286.6	9.2%	blas_memory_alloc
4.1%	2675.8	249.2	8.5%	integrate_ortho$integrate_pgf_product_rspace$qs_integ
3.8%	2495.4	219.6	8.1%	integrate_core_3_
2.9%	1860.3	178.7	8.8%	_EXP_15
2.9%	1855.6	194.4	9.5%	integrate_core_0_
2.8%	1826.0	227.0	11.1%	dgemv_t
2.7%	1749.9	191.1	9.8%	integrate_pgf_product_rspace$qs_integrate_potential_
2.6%	1661.1	229.9	12.2%	exp_radius$qs_util_
2.0%	1330.6	155.4	10.5%	exp_radius_very_extended$qs_interactions_
1.7%	1095.9	101.1	8.4%	build_pgf_product_list$hfx_pair_list_methods_
1.6%	1059.0	140.0	11.7%	_dgemv_
1.6%	1037.8	122.2	10.5%	integrate_v_rspace$qs_integrate_potential_
1.5%	1002.6	144.4	12.6%	dgemm_oncopy
1.4%	884.6	140.4	13.7%	dgemm_otcopy
1.3%	831.7	159.3	16.1%	integrate_core_4_
1.2%	774.3	89.7	10.4%	build_pair_list_pgf$hfx_pair_list_methods_
1.0%	652.5	117.5	15.3%	int2pair$task_list_methods_

Table 2 CrayPAT sampling analysis of CP2K MP2 calculation. "Imb. Samp." is a measure of load imbalance calculated as (max – mean) over all processing elements.

The implementation of MP2 in CP2K is essentially embarrassingly parallel – processes compute the integrals for a subset of the occupied and virtual orbital pairs independently. Therefore to speed up the overall calculation, we investigated serial optimisation of these routines.

3. Specialized/improved version of xyz_to_vab

According to discussions with the CP2K developers, the values of the angular momentum quantum numbers `la_max_local` and `lb_max_local` are typically integers between 0 and 7 inclusive. An analysis of the testcase provided (Figure 1) highlighted that the most common parameter combinations are in fact combinations of 0 and 1, thus making it worthwhile investing additional time optimizing these combinations.

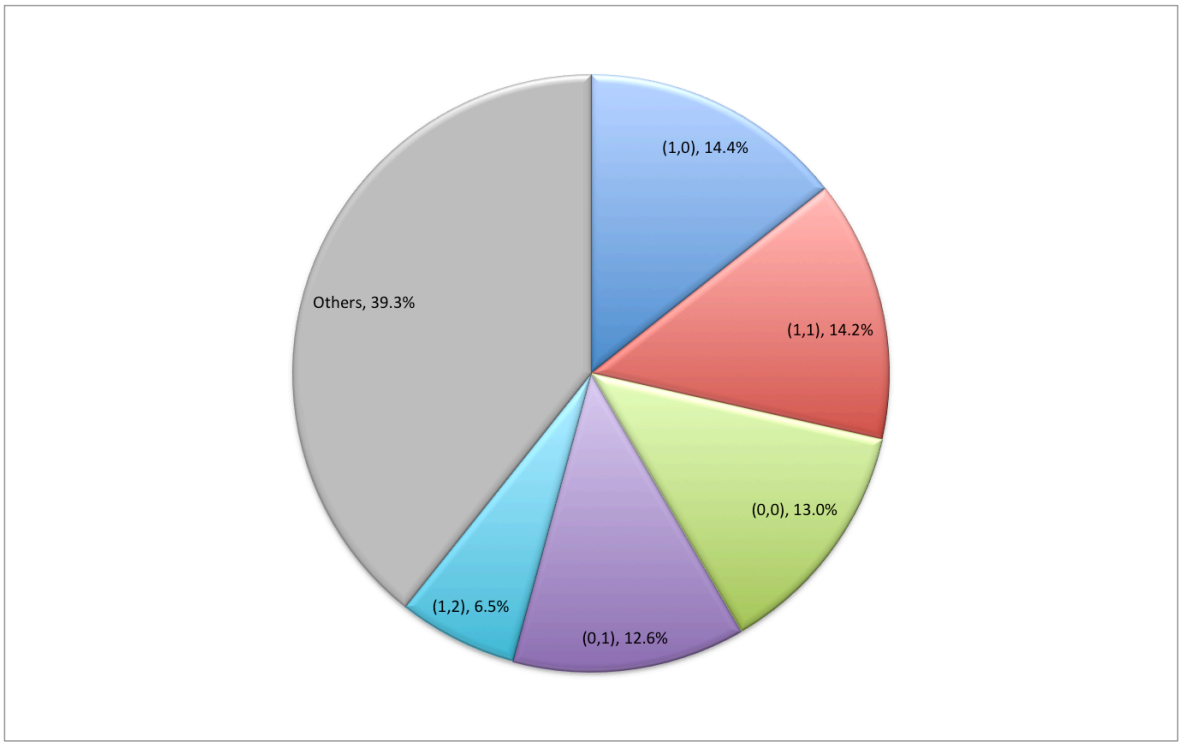

Figure 1 Percentage of the total number of calls in the test problems for each combinations of values (`la_max_local`,`lb_max_local`).

Template system

`xyz_to_vab` has a complex nested loop structure depending on the values of the paramaters `la_max_local` and `lb_max_local` which inhibits the compiler from generating efficient code. We planned to generate specific versions of the routine for common values of `la_max_local`, `lb_max_local`, thus enabling further optimizations to the routine by the compiler. Generating dozens of versions of the same routine would reduce the sustainability of the code, so to facilitate future addition or replacement of subroutines or optimizations, we have used a template system to specify the structure of the routines. The template is then parsed by a python script, which generates the different versions of the routine.

The template variables are accessed using `@<var_name>@` (e.g. `@<la_max_local>@`)). Fortran `DO` loops are replaced by `<$--(for ...)-->` tags, and each correspondent `ENDDO` statement is replaced by a `<$--(end)-->` tag. The python script unrolls loops by generating the contents of the loop with the iteration value. For example, when parsing the code in Listing 1 the python script will generate the code in Listing 2.

```
1  <$--(for value in range (0,3))-->
2  a[@<value>@] = b[@<value>@]
3  <$--(end)-->
```

Listing 1 Example of loop using the template syntax

```
1  a[0] = b[0]
2  a[1] = b[1]
3  a[2] = b[2]
```

Listing 2 Code generated by the python script when parsing the code in Listing 1

The code of the `xyz_to_vab` routine has been re-written using the template subsystem, using the values of `la_max_local` and `lb_max_local` as template variables. For each combination of `la_max_local` and `lb_max_local` we generate the code of the routine with the template and write all combinations to a file. A caller routine with all combinations is generated as well. This caller routine uses the original, generic version of the `xyz_to_vab` routine if the values of `la_max_local` and `lb_max_local` passed in is not implemented by a specialised routine. CP2K developers only need to modify two parameters in the python script to generate routines for other intervals e.g. if performing simulations with unusually large values of the L quantum number.

Optimizations applied to the template

The routine is divided on two steps. First (as shown in Listing 3), the routine computes the polynomial expansion coefficients, which are stored on a local temporary array named `alpha`. The array is de-allocated at the end of the subroutine. The loop in line 3 has a constant number of iterations, and loops in lines 4 and 5 have at maximum `la_max_local` or `lb_max_local` iterations. Boundaries of the loops in lines 8 and 11 are computed in terms of the values of the loops 4 and 5. The variable `lp` = `la_max_local` + `lb_max_local`, so it is also possible to compute its value at compile time for the specialized subroutines.

```
1  ALLOCATE(alpha(0:lp,0:la_max_local,0:lb_max_local,3))
2  alpha(:,:,:,:)=0.0_dp
3  DO iaxis=1,3
4   DO lxa=0,la_max_local
5   DO lxb=0,lb_max_local
6    binomial_k_lxa=1.0_dp
7    a=1.0_dp
8    DO k=0,lxa
9     binomial_l_lxb=1.0_dp
10    b=1.0_dp
11    DO l=0,lxb
12     alpha(lxa-l+lxb-k,lxa,lxb,iaxis)=alpha(lxa-l+lxb-k,lxa,lxb,iaxis)+ &
                                  binomial_k_lxa*binomial_l_lxb*a*b
13     binomial_l_lxb=binomial_l_lxb*REAL(lxb-l,dp)/REAL(l+1,dp)
14     b=b*(rp(iaxis)-(ra(iaxis)+rab(iaxis)))
15    ENDDO
16    binomial_k_lxa=binomial_k_lxa*REAL(lxa-k,dp)/REAL(k+1,dp)
17    a=a*(-ra(iaxis)+rp(iaxis))
18   ENDDO
19  ENDDO
20 ENDDO
```

Listing 3 Computation of the polynomial expansion coefficients.

When `la_max_local` and `lb_max_local` are known, we can fully unroll the loop nest facilitating further optimization steps by the compiler. However, we performed a number of higher-level optimisations based on our understanding of the algorithm. We replaced the dynamic allocation of the alpha array by the static declaration of three separate alpha arrays, one per each `iaxis` value. This reduces the number of indirect addressing to access memory locations. The `iaxis` loop is written now in the template, thus the code is still easy to follow.

Listing 4 shows the second part of the `xyz_to_vab` routine, where the `vab` vector is computed. This requires several nested loops whose boundaries are computed in terms of `la_max_local` and `lb_max_local`, so they can be expanded in the specialised routines. Some loop nests can be simplified or entirely removed when `la_max_local` or `lb_max_local` are zero or one. For example, the loop in Line 4 can be replaced by a single iteration if `lp` = 1 (i.e `la_max_local` + `lb_max_local` = 1). Loops from Line 26 to Line 29 can be expanded in the template as well. Internal loops in Lines 30 and 32 feature relatively complex boundary conditions involving the call to a `MAX`() intrinsic. However, if `lzb` + `lyb` is zero, we can simplify the loop boundaries, again allowing the compiler freedom to optimise by removing the function call from the loop bounds. This condition is known when parsing the template so different code will be generated depending on the values of `lzb` and `lyb`.

Many other micro-optimizations have been implemented in the template. Further details are available on the template itself, available in the CP2K Subversion repository [9].

When the values of the variables when `la_max_local` and `lb_max_local` are both zero this greatly simplifies the code of the (0,0) routine into a single assignment. As the (0,0) case is one of the most common, this greatly improves performance, as shown in Figure 2.

```
1  ALLOCATE(coef_ttz(0:la_max_local,0:lb_max_local))
2  ALLOCATE(coef_tyz(0:la_max_local,0:lb_max_local, &
                  0:la_max_local,0:lb_max_local))
3  lxyz=0
4  DO lzp=0,lp
5   coef_tyz=0.0_dp
6   DO lyp=0,lp-lzp
7    coef_ttz=0.0_dp
8    DO lxp=0,lp-lzp-lyp
9     lxyz=lxyz+1
10    DO lxb=0,lb_max_local
11     DO lxa=0,la_max_local
12      coef_ttz(lxa,lxb) = coef_ttz(lxa,lxb)+ &
                    coef_xyz(lxyz)*alpha(lxp,lxa,lxb,1)
13     ENDDO
14    ENDDO
15   ENDDO
16   DO lyb=0,lb_max_local
17    DO lya=0,la_max_local
18     DO lxb=0,lb_max_local-lyb
19      DO lxa=0,la_max_local-lya
20       coef_tyz(lxa,lxb,lya,lyb)=coef_tyz(lxa,lxb,lya,lyb)&
                +coef_ttz(lxa,lxb)*alpha(lyp,lya,lyb,2)
21      ENDDO
22     ENDDO
23    ENDDO
24   ENDDO
25  ENDDO
26  DO lzb=0,lb_max_local
27   DO lza=0,la_max_local
28    DO lyb=0,lb_max_local-lzb
29     DO lya=0,la_max_local-lza
30      DO lxb=MAX(lb_min_local-lzb-lyb,0), &
              lb_max_local-lzb-lyb
31       jco=coset(lxb,lyb,lzb)
32       DO lxa=MAX(la_min_local-lza-lya,0), &
              la_max_local-lza-lya
33        ico=coset(lxa,lya,lza)
34        vab(ico,jco)=vab(ico,jco)+ &
                coef_tyz(lxa,lxb,lya,lyb)&
                *alpha(lzp,lza,lzb,3)
36       ENDDO
37      ENDDO
38     ENDDO
39    ENDDO
40   ENDDO
41  ENDDO
42 ENDDO
```

Listing 4 Sketch of the second part of the `xyz_to_vab` routine

A set of additional Gfortran compiler flags focused on constant optimizations (`-fgcse-sm -fgcse-las -fmerge-all-constants`) was also explored, but the additional building time required did not compensate the performance achieved. It is worth to considering using these flags for the auto-tuning framework (see Section 4), where the building time is not as important for compiling the entire code.

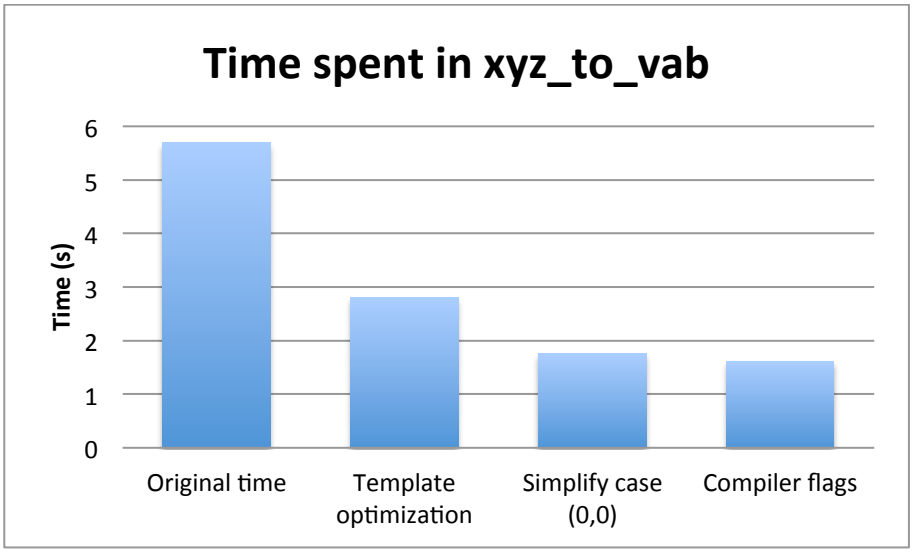

Figure 2 Impact of the different optimizations applied on the execution time of the `xyz_to_vab` routine according to CP2K internal timing.

To avoid extending the build time of the default C2PK code beyond acceptable limits, two different templates were used. The default one completely unrolls all the possible loops, as described before. However, when `la_max_local` or `lb_max_local` are greater than a specific number (currently set to 3) the size of the routine increases noticeably, as does its compilation time.

An alternative template, where only the inner loops are expanded, is provided in the python script. A parameter on this script allows generation of the routine using one or other template depending on the size of `la_max_local` and `lb_max_local`.

4. Auto-tuning framework for integrate kernels

As shown in Section 2 the integration kernels `integrate_core_*` make up a significant fraction of the total computation. There are also kernels for collocation (the inverse of integration). Although these are not important in MP2 calculations, they are very similar, so we applied all our optimisations to both sets of routines, thus benefitting a wider range of users of the code, in addition to the specific application this project supported. Similarly to `xyz_to_vab` these routines feature a set of nested loops accessing to two- and three-dimensional arrays. Both routines are called several times using different parameters. However, the parameter defining the number of iterations of the internal loops (`lp`) is typically between 0 and 7.

Listing 5 shows a sketch of the integrate routine. All occurrences of the `lp` variable are highlighted in red. There are three loops whose boundaries are defined by this variable. If the value of `lp` were known in compile time, loops could be unrolled or written using vector notation to increase efficiency. Similar situation arises in the collocate routines.

We have created versions of the integrate and collocate routines for each possible value of `lp` within a defined range of the most frequent values (set to 7 by default). This facilitates the compiler to generate efficient code for these cases, improving overall performance.

From our experience with different systems, we have found that the optimal sequential micro-optimizations vary greatly across compilers and platforms. An optimization that benefits a particular compiler might hinder the performance on others. The existing code had a particular set of loop optimisations applied, but we found that these did not give performance that was portable from one system to another.

To prevent this, and to provide CP2K users with the best possible performance, we have adapted an existing auto-tuning framework for these routines. Updated reasonable defaults are provided with the C2PK distribution, and the auto-tuning framework is now bundled with the code so that users can generate their own optimized version for their particular system.

The auto-tuning framework generates different implementations of both integrate and collocate routines using various source transformations:

- Unroll the loop in Line 7
- Use vector notation for contents of loop in Line 7
- Unroll the loops in Lines 15 and 16
- Use vector notation in the loop contents in Line 16
- Unroll the loops in lines 26,28 and 29
- Use vector notation for the contents of the loop in Line 29

```
1  DO kg=kgmin,0
2   ...
3   DO jg=jgmin,0
4    ...
5    DO ig=igmin,igmax
6     ...
7     DO lxp=0,lp
8      coef_x(1,lxp)=coef_x(1,lxp)+s01*pol_x(lxp,ig)
9      coef_x(2,lxp)=coef_x(2,lxp)+s02*pol_x(lxp,ig)
10     coef_x(3,lxp)=coef_x(3,lxp)+s03*pol_x(lxp,ig)
11     coef_x(4,lxp)=coef_x(4,lxp)+s04*pol_x(lxp,ig)
12    ENDDO
13   ENDDO
14   lxy=0
15   DO lyp=0,lp
16    DO lxp=0,lp-lyp
17     lxy=lxy+1
18     coef_xy(1,lxy)=coef_xy(1,lxy)+coef_x(1,lxp) &
                          * pol_y(1,lyp,jg)
19     coef_xy(2,lxy)=coef_xy(2,lxy)+coef_x(2,lxp) &
                          * pol_y(1,lyp,jg)
20     coef_xy(1,lxy)=coef_xy(1,lxy)+coef_x(3,lxp) &
                          * pol_y(2,lyp,jg)
21     coef_xy(2,lxy)=coef_xy(2,lxy)+coef_x(4,lxp) &
                          * pol_y(2,lyp,jg)
22    ENDDO
23   ENDDO
24  ENDDO
25  lxyz = 0
26  DO lzp=0,lp
27   lxy=0
28   DO lyp=0,lp-lzp
29    DO lxp=0,lp-lzp-lyp
30     lxyz=lxyz+1 ; lxy=lxy+1
31     coef_xyz(lxyz)=coef_xyz(lxyz)+coef_xy(1,lxy) &
                           * pol_z(1,lzp,kg)
32     coef_xyz(lxyz)=coef_xyz(lxyz)+coef_xy(2,lxy) &
                           * pol_z(2,lzp,kg)
33    ENDDO
34    lxy=lxy+lzp
35   ENDDO
36  ENDDO
37 ENDDO
```

Listing 5 Sketch of the default version of the integrate routine

Similar transformations are performed in the collocate routine. All combinations are tested individually for each value of `lp`. Caller routines, acting as wrappers to the original calls to collocate and integrate, are generated as well.

All the code generated by the auto-tuning framework, is first checked for correctness using provided sample data (avoiding possible miscompiled code at high optimization levels) and then has its performance measured.

When all possible combinations of transformations are explored, a script summarizes the timings and returns the best combination of optimizations for each `lp` value. This combination is finally used to generate the library `libgrid.a`. This library can be linked with CP2K and will override the defaults if the appropriate pre-processor macro (`__HAS_LIBGRID`) is defined when building CP2K.

Figure 3 shows the workflow of the auto-tuning framework. The user needs to set the appropriate compiler options for their architecture by modifying the variables `FC_comp` and `FCFLAGS` in the `config.in` file. Then they can generate the library `Makefile` by running the main script (`generate_makefile.sh`). The default behaviour is to build and test the library.

It is important to note that creating the entire library might take a significant amount of time, particularly when compiling with high optimization levels. The `Makefile` has been designed in such a way that it is possible to split the creation of the library into separate steps. The `README` file included in the package provides the users with the necessary instructions.

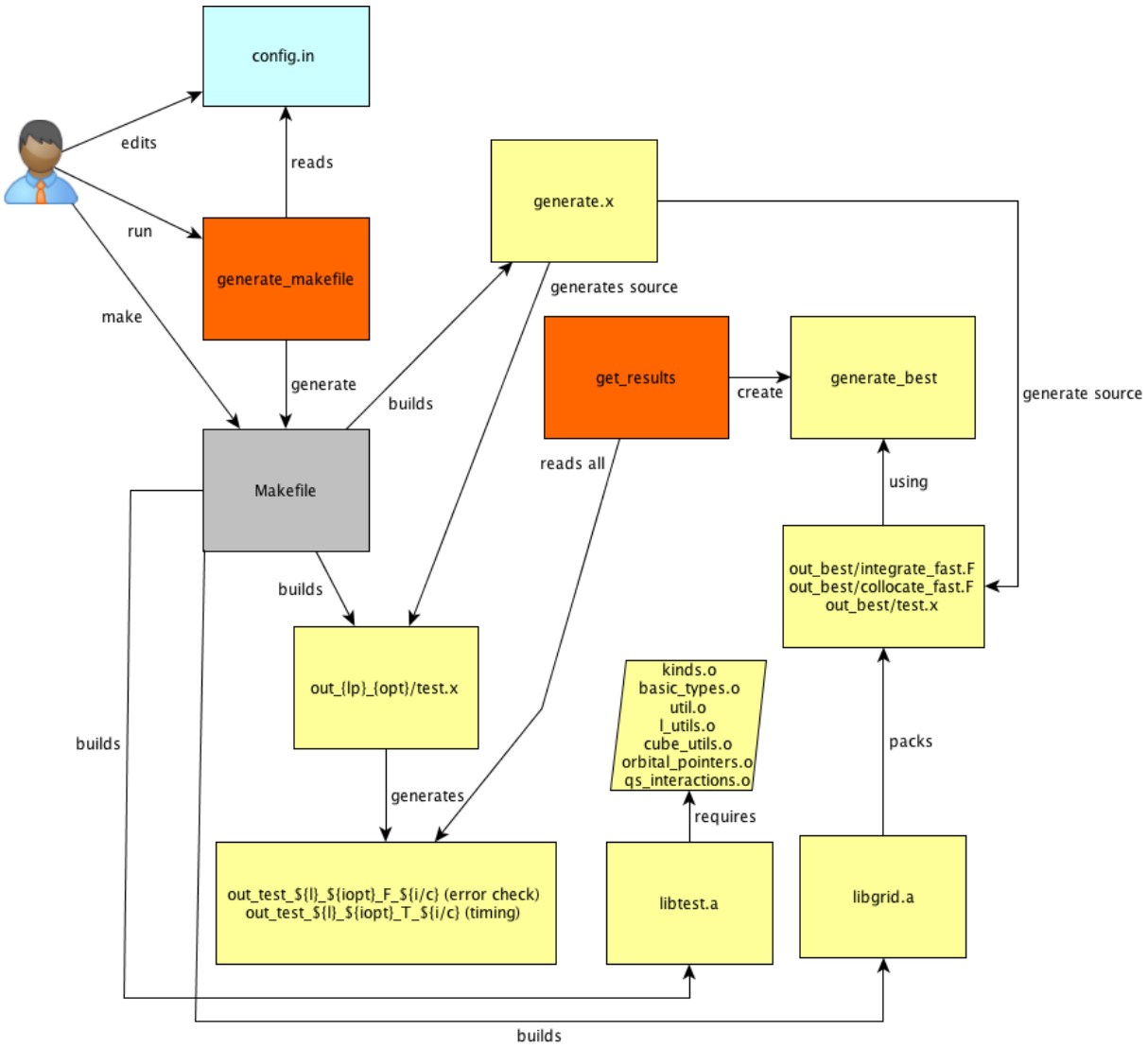

Figure 3 Auto-tuning framework workflow. Bash scripts are highlighted in orange, grey is used for `Makefile` and cyan for user-editable files. Yellow boxes represent `Makefile` targets.

Figure 4 shows the execution time for the collocate routine for each `lp` size using the best combination for different compilers on Todi. For each of these cases a different set of optimisations was applied by the auto-tuning framework to generate the best performing executable. The poor performance of code produced by the Cray compiler compared with GNU or Intel compilers is not yet understood, although Cray are currently investigating the issue.

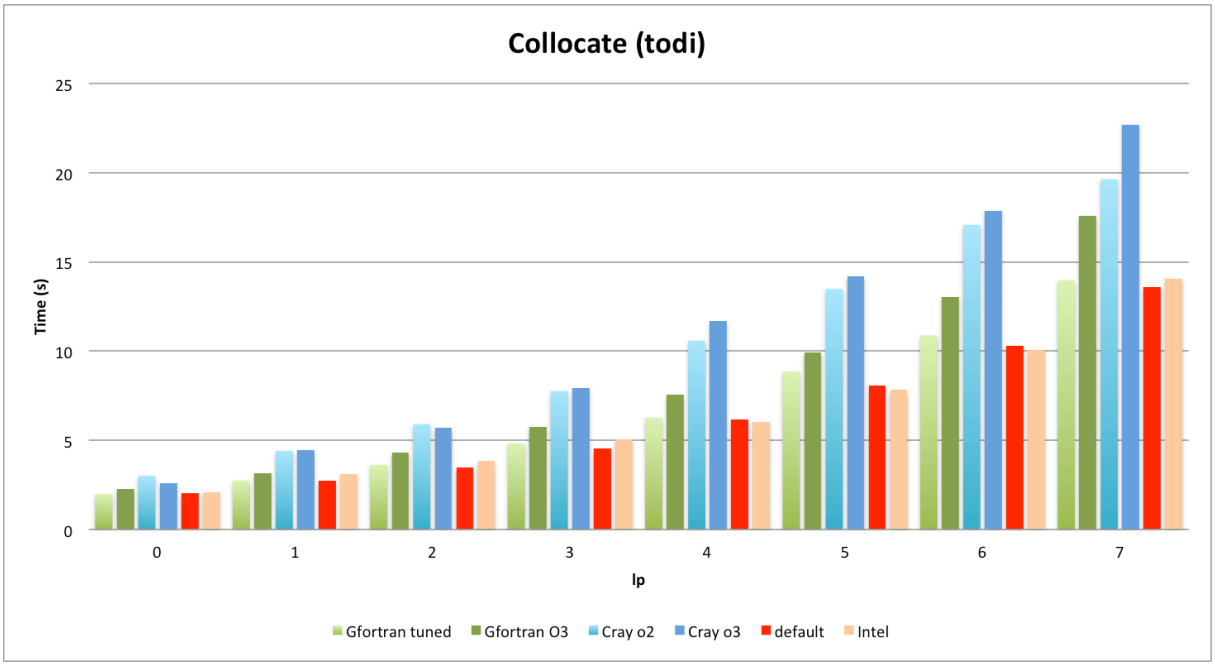

Figure 4 Comparison of execution times for the optimized collocate routines. 'Default' stands for the default `CFLAGS` set in the CP2K SVN and compiled with gfortran 4.7. 'Gfortran tuned' uses a set of additional compilation flags to pre-compute constant values at compile time. 'Cray o2' and 'Cray o3 'represents the time using the Cray Compiler Environment 8.0.6 with flags –O2 and –O3 respectively. 'Intel' stands for the Intel compiler using the –O3 flag.

5. OpenMP 3D FFT routines

As in standard DFT calculations using QUICKSTEP, Fast Fourier Transforms are required to transform the density on real-space grids into a plane-wave grid representation, and transform the corresponding potential found by the Poisson solver back into real-space. While significant effort has been expended on efficient FFT using both MPI and OpenMP [6][7], this has focused on distributed-data FFTs, which are required when the grids are distributed as is the case for most parallel calculations. In MP2 however, due to the distribution of orbital pairs to processes, each MPI process has its own local grids which are transformed in isolation, and thus a serial FFT is performed on each process. To get the best possible scalability from CP2K, and in some cases to access the maximum memory available per process, it is desirable to use mixed-mode MPI and OpenMP. The serial FFT routines were not previously OpenMP-parallel as they were considered unimportant but as a result they can prove to be costly in MP2 calculations where large numbers of threads per process are used. Therefore we investigated adding OpenMP threading to the serial 3D FFT routine in CP2K.

CP2K uses FFTW3 [8] as the default FFT library, although an in-build FFT is also available for systems where FFTW3 is not installed. The module `fft3_lib.F` contains the routines to create, execute and destroy the FFT plans. The subroutine `fftw3_create_plan_3d` receives the plan structure (defined in `fft_types.F`), the input and output arrays and the plan style according to the input parameters from the user, indicating if the plan is `ESTIMATE`, `MEASURE`, `PATIENT` or `EXHAUSTIVE`. These values indicate to the FFTW planner how much time should spend computing the optimal plan, thus trading off setup time for potentially better runtime performance.

The initial code in CP2K calls the FFTW 3D transform routines directly. It is possible to allow FFTW to use available OpenMP threads during execution of an FFT, but prior experience had shown that it was sometimes possible to get better performance by splitting the FFT into independent FFTs on each axis, and parallelising over these with threads.

After exploring different possible strategies for the parallelisation we concluded that the most suitable implementation for the evaluated systems (and potentially for other systems as well) was to use separate one-dimensional plans for each dimension, splitting the Z-axis of the input array across the different OpenMP threads. Three plans (`fftw_plan_nx`, `fftw_plan_ny` and `fftw_plan_nz`) are created. In the case that the size of the Z dimension does not divide evenly into the number of threads, initially we simply performed the remaining FFTs on a single thread.

This implementation of the separated plans uses the FFTW Guru Interface rather than the basic interface used before. The Guru Interface enables usage of different strides for each dimension of the array containing the data to be transformed. This is required so we can manually distribute the work between the threads.

In addition, based on the assumption that for large arrays as the FFT is a memory-bandwidth limited operation, we opted for transposing the output of the FFTW in each dimension so the next FFTW will perform its transformation on contiguous data (i.e. the stride is one), to promote efficient streaming of data from main memory.

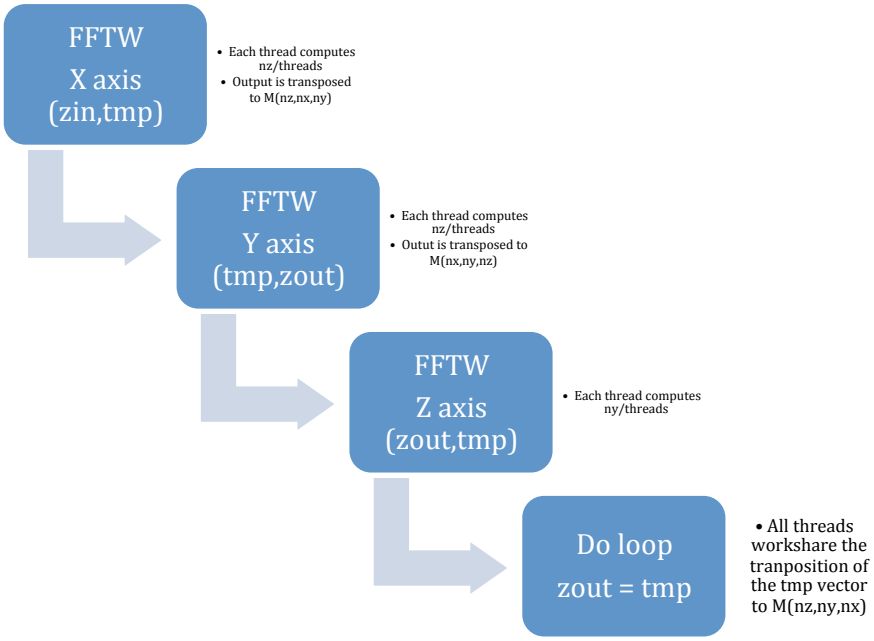

Figure 5 Steps to compute the 3D FFTW in the `fft3d_s` routine for the out-of-place case.

We evaluated different combinations for transposing and computing the arrays of the FFT. The optimal order for computing each plane of the 3D FFTW is shown in Figure 5. In particular, we found that the final transposition is best not computed directly in the output of the Z-axis FFT but using an external loop, parallelized using OpenMP. This method proved to be faster than directly transposing the `tmp` array for many cases. We have prioritised the large grid sizes in our choice of implementation, since these are intrinsically more expensive. There are comments in the source with instructions on how to change this order, facilitating future optimization work on other platforms.

A comparison of the performance of different FFTW-OpenMP implementations is shown in Figure 6. Overall, we see speedups of 2-10x over the serial FFT when using 8 threads with our chosen implementation, depending on the size of the grid. However, we observe that performance is poor when using a large number of threads with a small grid size. This is due to load imbalance - when the grid size is not evenly divided into the number of threads, the remainder part is performed sequentially.

To avoid this issue, instead of doing the remainder sequentially, we distribute the remaining rows over the working threads. For example, if we are using 4 threads and there are 7 FFTs to perform, with the initial implementation four threads will perform one FFT each and then three rows would be performed sequentially by one thread. With our improved load balancing mechanism, three threads will perform the two FFTs while the last one will perform a single FFT, thus minimising the imbalance. The performance of this scheme is shown in Figure 7.

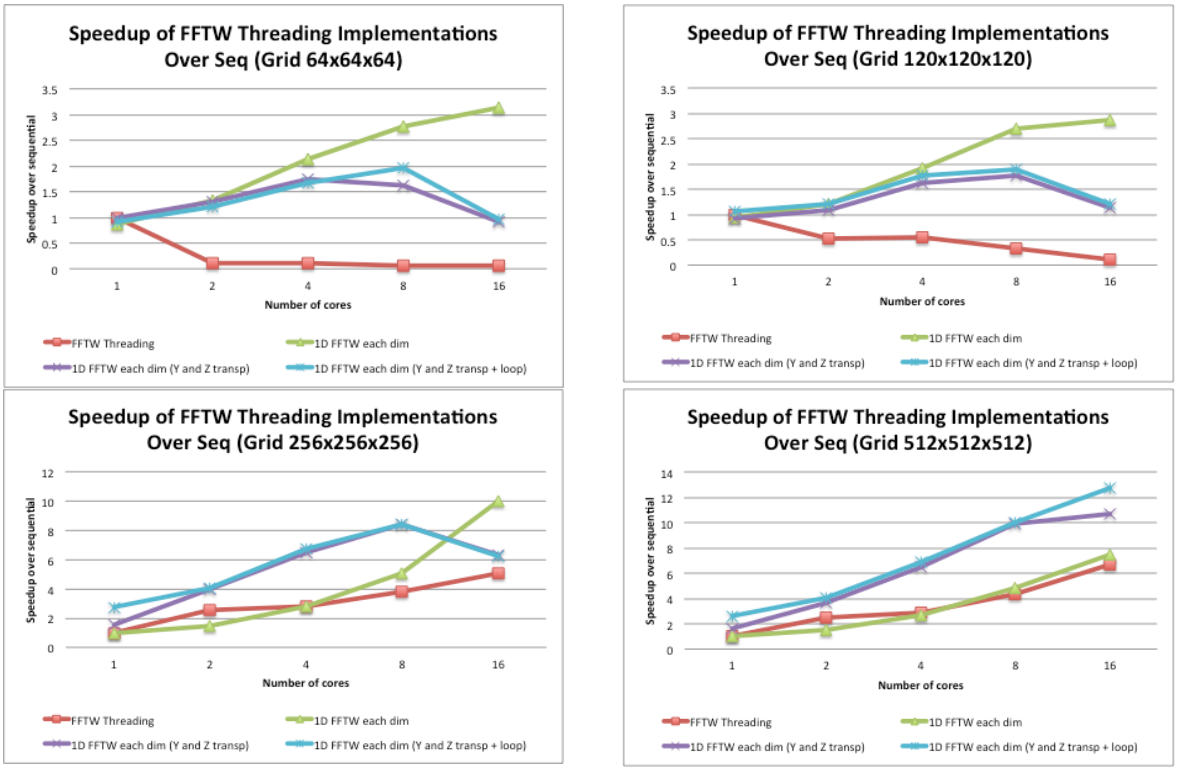

Figure 6 Comparison of speedup over the sequential implementation of the 3D FFT using FFTW threads(red) against three different implementations of FFTW with manual OpenMP. The green line represents an implementation not transposing the Z-axis; the magenta line represents an implementation transposing the output of both Y- and Z-axis; and finally, the cyan line represents an implementation transposing the output of the Z-axis on an external OpenMP loop instead of directly transposing the FFTW output.

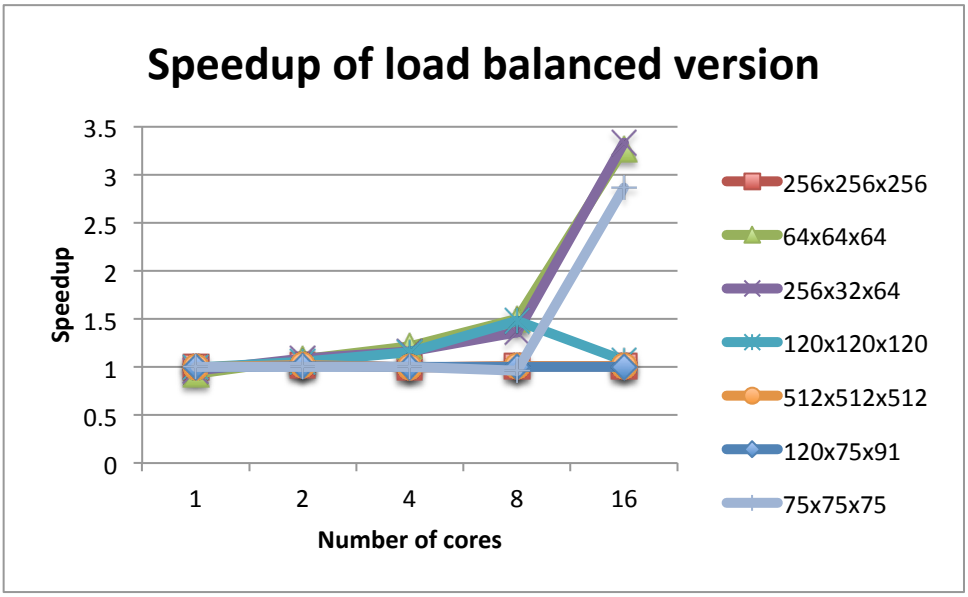

Figure 7 Speedup of the alternative load balancing mechanism over the initial implementation for a variety of grid sizes.

This alternative load balance mechanism cannot be directly applied in the single precision case (`__FFT_SGL`). FFTW requires that if a plan is re-used for multiple arrays (or in our case, by multiple threads at offsets into the same array), each input must have the same alignment with respect to 16-byte boundaries to allow vector

instructions like SSE or AVX to be used. However, with single precision complex numbers each element is 8 bytes and so we have no guarantee that the start of each thread's data has the same alignment if the number of elements per thread is odd. In this case we implemented a different approach where N-1 threads will perform an even number of FFTs whereas the last one will perform the remainder. For example, if the total number of FFTs is 11 and there are 4 threads, the first three threads would perform the FFTW of 2 rows whereas the last one would perform the 5 remaining rows. Although the imbalance in this case is bigger than in the double precision case, we avoid doing additional FFTs sequentially, and still respect the alignment requirements.

6. Overall performance evaluation on HERMIT

In order to evaluate the performance improvements achieved as a result of the above work we have carried out benchmarking of the new version of the code using the PRACE HERMIT system, using the aforementioned `NH3_32_bulk` test case.

Figure 8 shows a comparison of the execution time of MPI-only execution with fully populated nodes - one MPI rank per CPU core. The blue line shows the wall-clock execution time required for the test case with the initial version of the code, whilst the red line shows the execution time using the current version of CP2K including the improvements implemented during this project. Notice that the execution time slightly increases with a larger number of cores with the new version with respect to the previous version. Figure 9 shows the speedup obtained in the modules affected by our project together with the overall speedup of execution. Since we are not using threads in this particular execution, we expect the FFT time to remain unchanged. We see a speedup of 8% from the optimised grid operations across all core counts. The drop in overall speedup at 8192 and 16384 cores is due to the introduction of the routine `replicate_mat_to_subgroup` in the MP2 implementation. This routine uses a message-round-a-ring approach with blocking `MPI_SendRecv` calls to distribute matrix data to the subgroups. As confirmed by our timing reports, the cost of this scales linearly with the number of MPI processes, and may become a bottleneck to further scaling (on 16384 cores it takes ~10% of the total runtime). Investigating a tree-based broadcast is recommended, although we did not have time to do this within the scope of this project.

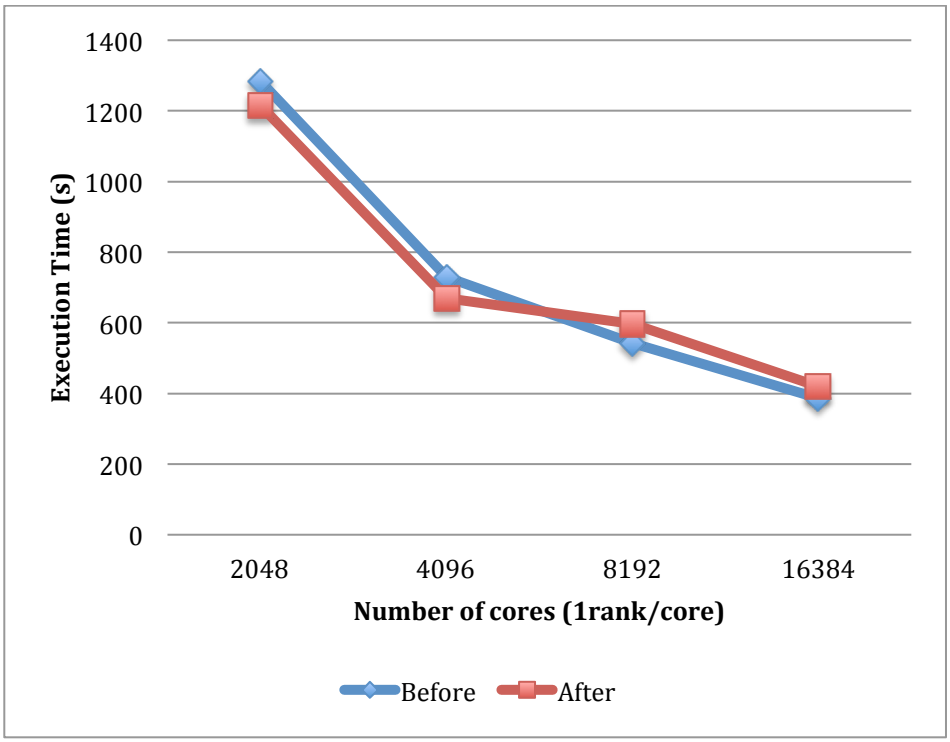

Figure 8 Execution time of the NH3 test case using 1 MPI rank per core.

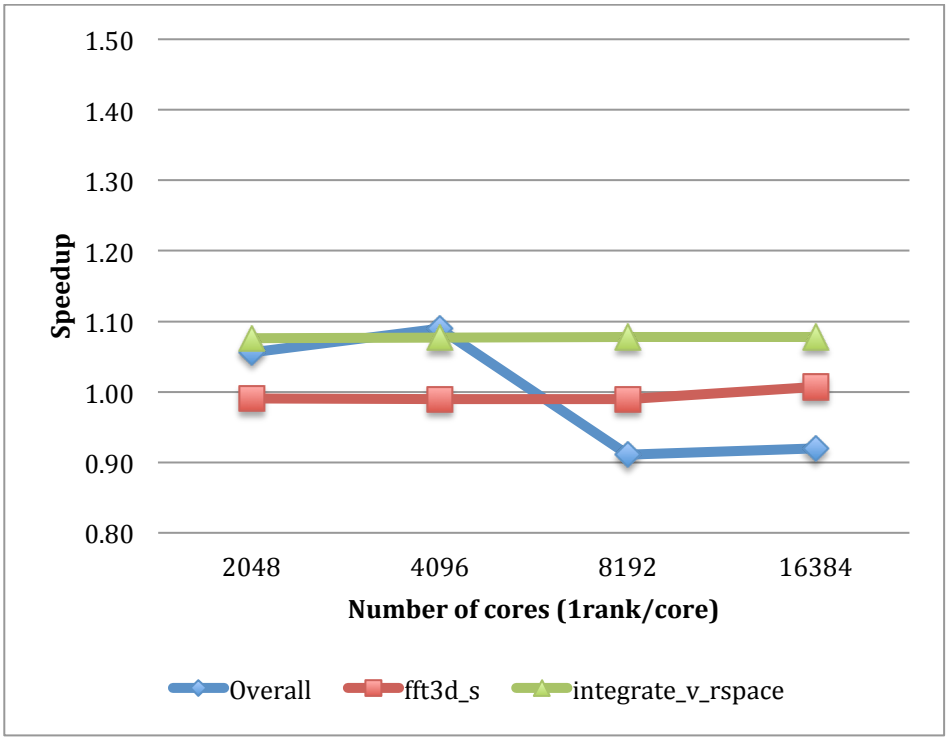

Figure 9 Speedup of each individual routine before and after compared with the overall speedup in HERMIT.

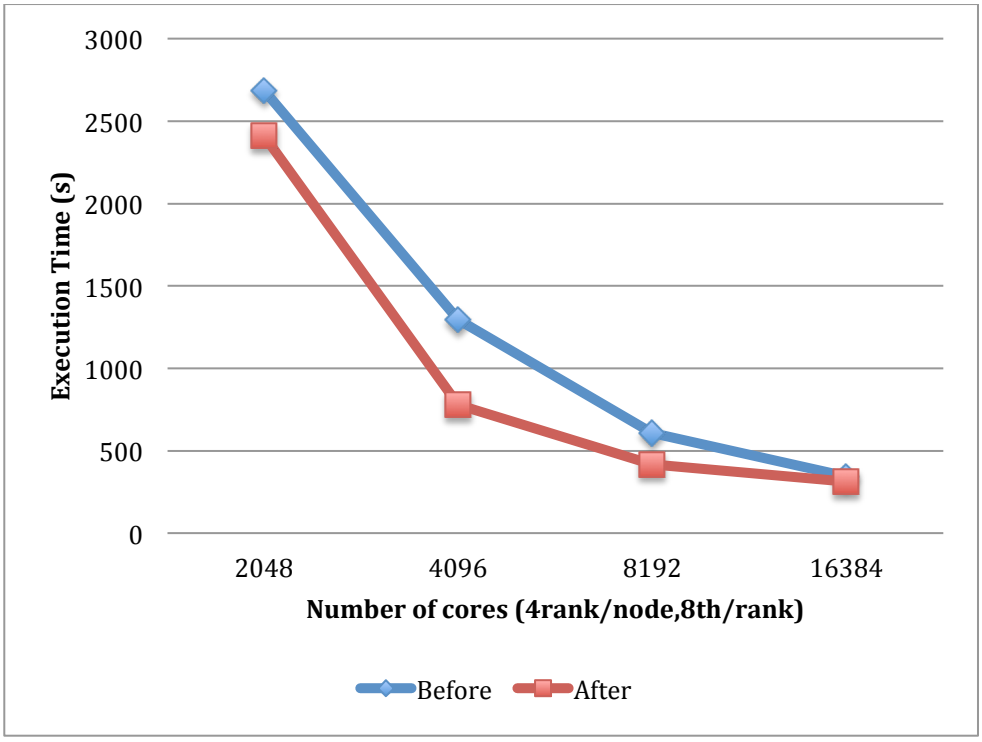

Figure 10 Execution time of the NH3 test case using 1 MPI rank per NUMA node (4 MPI ranks per node, 8 threads per rank)

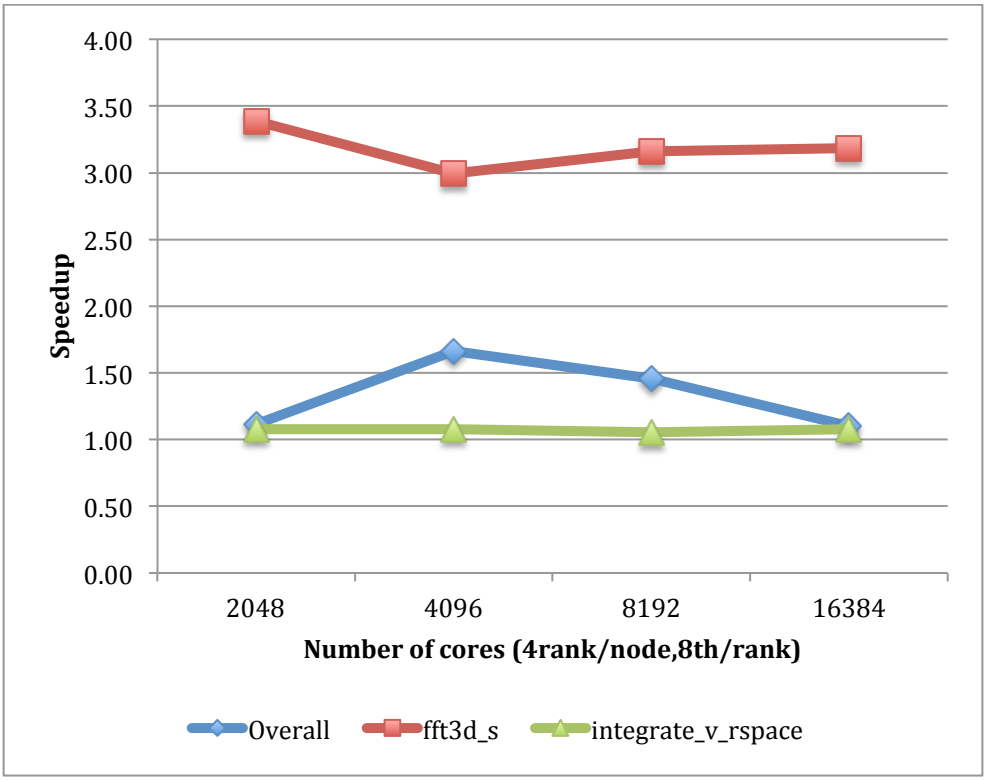

Figure 11 Speedup of individual routines in the current version of CP2K with respect to the initial version of the code (4 MPI ranks per node, 8 threads per rank)

We also measured the wall-clock time for calculations using multiple OpenMP threads per MPI rank. In this case we chose 8 threads to make best usage of the local caches and memory affinity, since there are 8 cores per NUMA region on the AMD 'Interlagos' processor architecture. The overall runtime of the initial and new versions of the code are shown in Figure 10 and the speedup over the initial code is shown in Figure 11. The speedup of the new FFT routines is around 3, which is a good result given that the largest FFT grid used in this calculation is 90^3. Again, the small but constant speedup from the optimised grid operations is observed. Overall, we find speedups of between 8% and 66% over the initial version of the code. The reason for this variation is the relative cost of the FFT compared with the MP2 integrals and Hartree-Fock computation.

It is worth noting that when using 8192 or 16384 cores, the runs with 8 OpenMP threads per rank are 44% and 34% faster respectively than the MPI-only runs, demonstrating the value of using mixed-mode parallelism for these types of calculations at large scale.

7. Conclusion

To summarise, we have optimised two key parts of the CP2K code needed for efficient MP2 calculations at large scale. Firstly, we have used loop structure optimisation via templates and an auto-tuning approach, to generate code for key integration kernels that can be effectively optimised by the compiler, to give maximal performance on any given machine architecture. For our Gfortran on AMD Interlagos environment, we found a speedup of around 8% over the original code for these routines. Larger speedups may be obtained on other architectures for which the initial code is further from optimal. We have also implemented a new OpenMP-parallel 3D FFT routine, which gives speedups of 2-10x when using 8 OpenMP threads depending on the size of the FFT grid. For a test case representative of a real user job we found the FFT performance was improved by a factor of 3. All of our improvements are in the current SVN trunk, available for download at on the CP2K SourceForge project page [9].

We have identified a communication-bound routine `replicate_mat_to_subgroup` which scales poorly with increasing numbers of MPI processes, although we note that already at 8192 cores, it is more efficient to use mixed-mode MPI/OpenMP parallelism, which will mitigate this cost to some extent.

Finally, we note that Prof. Joost VandeVondele, whose use of the PRACE RI this project was designed to support had now submitted a request for 40 million CPU hours to perform ground-breaking MP2 calculations, so we consider this preparatory access project to have been a success.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. We acknowledge that the results of this research have been achieved using the PRACE Research Infrastructure resource HERMIT based in Germany at HLRS.

We are very grateful to Prof. Joost VandeVondele (ETH Zurich) and Prof. Jürg Hutter (Univ. Zurich) for help and advice on code development and for access to the Cray XK6 at CSCS for development and testing.

References

[1] CP2K Project Website, http://www.cp2k.org

[2] Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassaing and J. Hutter, Comp. Phys. Comm. 167, 103 (2005).

[3] Second Order Møller-Plesset Perturbation Theory in the Condensed Phase: An Efficient and Massively Parallel Gaussian and Plane Waves Approach, M. Del Ben, J. Hutter and J. VandeVondele, Journal of Chemical Theory and Computation (2012)

[4] Swiss National Supercomputing Centre, http://www.cscs.ch

[5] High Performance Computing Center Stuttgart, http://www.hlrs.de

[6] Improving the performance of CP2K on HECToR, I. Bethune, HECToR dCSE Report, (2009)

[7] Improving the scalability of CP2K on multi-core systems, I. Bethune, HECToR dCSE Report (2010)

[8] The Design and Implementation of FFTW3, M. Frigo and S. Johnson, Proceedings of the IEEE 93 (2), 216-231 (2005)

[9] CP2K SVN repository, http://sourceforge.net/p/cp2k/code

Appendix A

CP2K Input file `NH3_32_bulk` used for profiling and testing throughout this project:

```
&GLOBAL
  PROJECT periodic_NH3_32_real_TZ
  PRINT_LEVEL MEDIUM
  RUN_TYPE ENERGY
  &TIMINGS
     THRESHOLD 0.01
  &END
&END GLOBAL
&FORCE_EVAL
  METHOD Quickstep
  &DFT
    BASIS_SET_FILE_NAME  HFX_BASIS
    POTENTIAL_FILE_NAME  GTH_HF_POTENTIALS
    &MGRID
      CUTOFF 200
      REL_CUTOFF 25
    &END MGRID
    &QS
      METHOD GPW
      EPS_DEFAULT 1.0E-15
      EPS_PGF_ORB 1.0E-30
    &END QS
    &SCF
      SCF_GUESS RESTART
      EPS_SCF 1.0E-7
      MAX_SCF 100
      ADDED_MOS 15000 15000
    &END SCF
    &XC
      &XC_FUNCTIONAL NONE
      &END XC_FUNCTIONAL
      &HF
        FRACTION 1.0000000
        &SCREENING
          EPS_SCHWARZ 1.0E-8
          SCREEN_ON_INITIAL_P FALSE
        &END SCREENING
        &INTERACTION_POTENTIAL
          POTENTIAL_TYPE TRUNCATED
          CUTOFF_RADIUS  5.0
          T_C_G_DATA t_c_g.dat
        &END
      &END HF
      &MP2
       METHOD  MP2_GPW
       &MP2_GPW
       &END
       MEMORY  1300.
       NUMBER_PROC  1
      &END
    &END XC
  &END DFT
  &SUBSYS
    &CELL
      ABC 10.182  10.182  10.182
    &END CELL
    &COORD
 <Full coordinates removed, contact ibethune@epcc.ed.ac.uk for a copy of the full system>
    &END COORD
    &KIND H
      BASIS_SET  cc-TZV2P-GTH
      POTENTIAL  GTH-HF-q1
    &END KIND
    &KIND N
      BASIS_SET  cc-TZV2P-GTH
      POTENTIAL  GTH-HF-q5
    &END KIND
    &TOPOLOGY
      &CENTER_COORDINATES
      &END
    &END TOPOLOGY
  &END SUBSYS
&END FORCE_EVAL
```