Available online at www.prace-ri.eu

**Partnership for Advanced Computing in Europe**

# CP2K – Scalable Atomistic Simulation for the PRACE Community

Iain Bethune[a]*, Adam Carter[a], Kevin Stratford[a], Paschalis Korosoglou[b,c]

[a]*EPCC, The University of Edinburgh, James Clerk Maxwell Buidling, The King's Buildings, Edinburgh, EH9 3JZ, United Kingdom*
[b]*AUTH, Aristotle University of Thessaloniki, Thessaloniki 52124, Greece*
[c]*GRNET, Greek Research & Technology Network, L. Mesogeion 56, Athens 11527, Greece*

**Abstract**

This report describes the work undertaken under PRACE-1IP to support the European scientific communities who make use of CP2K in their research. This was done in two ways – firstly, by improving the performance of the code for a wide range of usage scenarios. The updated code was then tested and installed on the PRACE CURIE supercomputer. We believe this approach both supports existing user communities by delivering better application performance, and demonstrates to potential users the benefits of using optimized and scalable software like CP2K on the PRACE infrastructure.

Application Code: CP2K

## 1. Project Overview

CP2K [1] is a freely available program (under GPL), written in Fortran 95, to perform atomistic and molecular simulations of solid state, liquid, molecular and biological systems. It provides a general framework for different methods such as density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW), and classical pair and many-body potentials. Recently, linear-scaling DFT and Møller-Plesset 2$^{nd}$ order perturbation (MP2) methods have been added, broadening the applicability of the code to a wider range of users.

CP2K is a popular and important code for materials science, life sciences and computational chemistry throughout Europe, and it supports the work of many research communities. In the UK, there are many users in the Material Chemistry HPC Consortium, led by University College London, and the code is the third most heavily used on HECToR, the UK National HPC Service. CP2K also dominates usage on the Cray XT/XE systems at CSCS, Switzerland. Citations of the main CP2K paper in Computer Physics Communications [1] indicate a wide and growing user base, and the code is also used by researchers under the DEISA and HPCEuropa-2 projects, and INCITE grant winners in the US.

One feature of CP2K which makes it a particularly important code with regards to use on Petascale systems in PRACE is its excellent scalability. Results showing scalability on tens of thousands of CPU cores are in the public domain [2] and we demonstrated similar performance on the PRACE system JUGENE in a previous PRACE project [3]. This is achieved in part by a hybrid MPI/OpenMP parallelization approach, which allows the power of large numbers of CPU cores to be harnessed while reducing the impact of algorithms which scale less than linearly with the number of MPI processes used. In some cases such as Hartree-Fock Exchange (HFX) calculations, using OpenMP is required to allow each process to access the entire memory of a compute node in

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

order to store tables of commonly re-used integrals, which provides excellent performance. We also assert that hybrid MPI/OpenMP maps well to the fat-node architecture of modern multi-core node supercomputers such as CURIE and HERMIT, where MPI can be used between NUMA regions (and compute nodes), and OpenMP within a single NUMA region.

Therefore, to best support the user communities of CP2K, this project ported and tested CP2K on the CURIE PRACE system. In addition, we have improved and extended the implementation of OpenMP within the code, focusing on several areas where we believed performance or scalability was an issue.

## 2. Porting CP2K to CURIE

The initial steps towards porting CP2K to the CURIE Tier-0 system were the compilation of the libint and the libsmm libraries.

Libint is comprised of C/C++ functions for the efficient evaluation of two-body molecular integrals. The libint source files are available through Sourceforge [4] but one may also find prebuilt packages of the library for various Operating Systems. To build libint from source two different compiler options were considered - the GNU compiler collection (version 4.5.1) and the Intel compiler suite (the default on CURIE).

Libsmm is a library for identifying and selecting the most efficient strategy for the multiplication of small matrix blocks (2x2, 3x3, 5x5, 7x7 etc.) that occur frequently in CP2K. Essentially, at compile-time, it makes a comparison between various generated multiplication kernels and the BLAS dgemm implementation and chooses the fastest method for each block size of interest. The libsmm source files are available directly within the CP2K source tree under `${CP2K_ROOT}/tools/build_libsmm`. To build libsmm one configuration file config.in containing information regarding the compiler and BLAS libraries to be compared and the total number of tasks to be used has to be edited. The configuration files used for building libsmm on CURIE for the GNU Compiler Suite and the Intel Compiler Suite are included in Appendix A. Once the configuration file is ready the procedure of building libsmm involves the submission of a batch job allocating a full CURIE node. The code generation and testing procedure takes place in parallel, and can take up to several hours depending on the number of block sizes considered.

Initially three different configurations (compilers, libraries etc) have been considered for compiling CP2K on the CURIE Tier-0 system. These are:

- GNU Compiler Suite for both CP2K and for building all required libraries (i.e. BLAS, LAPACK etc)
- GNU Compiler Suite for building CP2K, libint and libsmm and usage of prebuild MKL (Intel Math Kernel Library) libraries during linking stage
- Intel Compiler Suite for building CP2K, libint and libsmm and usage of prebuild MKL libraries during linking stage

Notice that for the first configuration several additional libraries had to be build from scratch as these were not available on the CURIE headnode. These were:

- BLAS
- LAPACK
- ScaLAPACK
- BLACS
- FFTW3

For each one of these three configurations appropriate CP2K arch files have been developed and used in order to produce three versions of CP2K per configuration (thus resulting in a total of nine executables). These three versions are appended the name extension sopt (for the serial version), popt (for the MPI only version) and psmp (for the hybrid MPI+OpenMP version).

Using the regression test suite provided with the CP2K source code we have been able to test these nine versions of the application. The CP2K regression test suite is based on a batch submission script that needs to be modified depending on the queuing system used. For testing the serial versions of CP2K (sopt) one compute node per run was allocated. Within the script 32 concurrent and asynchronous processes were used to run the tests in a parallel manner. For testing the parallel versions of CP2K (popt and psmp) 4 cpu cores were allocated per run. In this case, the tests were not executed concurrently but rather sequentially.

When running CP2K regression tests for the first time the outcome of most of the test results are marked as NEW (unless the test failed outright), since there is no reference result to compare to. Depending on the results of subsequent runs the tests are either marked as CORRECT, WRONG or FAILED depending on whether the outcome (per test) is the same with the first one (CORRECT), whether it differs by any amount from the first one (WRONG) and whether the test has finished unsuccessfully (FAILED). The tests using the psmp version of CP2K build with the Intel Compiler Suite have not been run to completion since for several test files the execution stalled (did not progress from some point on) until eventually the job was terminated by the batch system. This problem is still under investigation. For the remaining versions of CP2K available to us the initial

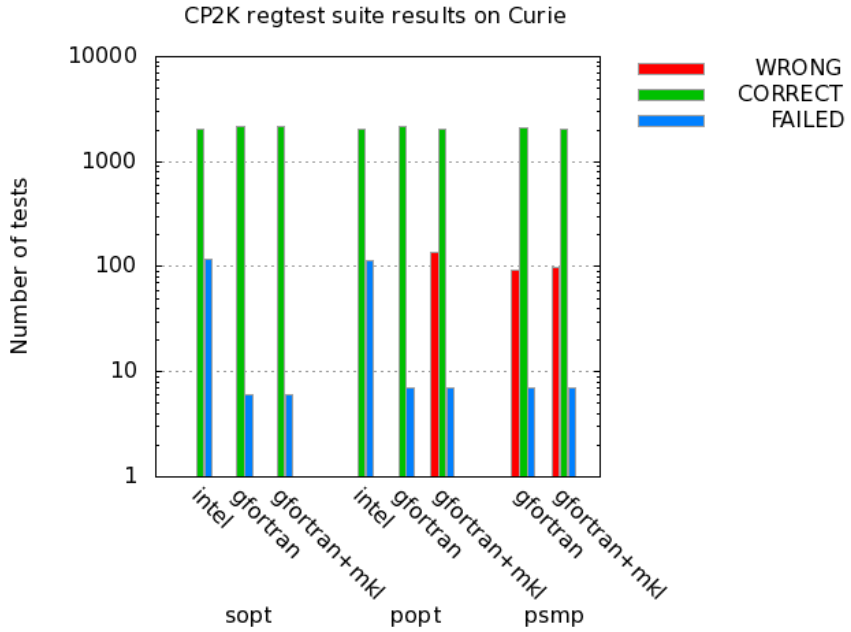



Figure 1: Initial results of CP2K regression test suite

results taken are shown in Figure 1.

Notice that for the psmp versions the relatively high number of WRONG results is due to numerical errors (driven by the non-deterministic aspects of shared memory parallelization such as dynamic task scheduling and reduction operations). These errors are relatively small in magnitude and thus they may be safely ignored.

In order to obtain some preliminary benchmark results a small benchmark CP2K input file H2O-64 was used which performs 10 MD steps on a system of 64 water molecules in a 12.42 Å cubic cell. Using this input file we quickly noted that the parallel versions built with the GNU Compiler Suite could not be executed on more than one compute node. This was due to certain MPI calls being made using Fortran array constructors which caused a crash in the Bull MPI library. A fix for this behavior was made and committed to the CP2K repository. However, several more problems with the GNU compiled versions linked with the custom build linear algebra libraries (BLAS, LAPACK etc) were identified thus further investigations using this configuration of CP2K were dropped.

Using the psmp version of CP2K build with the GNU Compiler Suite and linked with the prebuild MKL libraries and the 2 MPI only versions (popt) compiled with the GNU and Intel Compiler Suites respectively the performance results displayed in Figure 2 have been calculated. We note that for this small test case the scaling of the performance is not expected to be good when using more than 2 CURIE nodes (64 cores), but gives an estimate of how good each compiler is at serial optimization of the code.

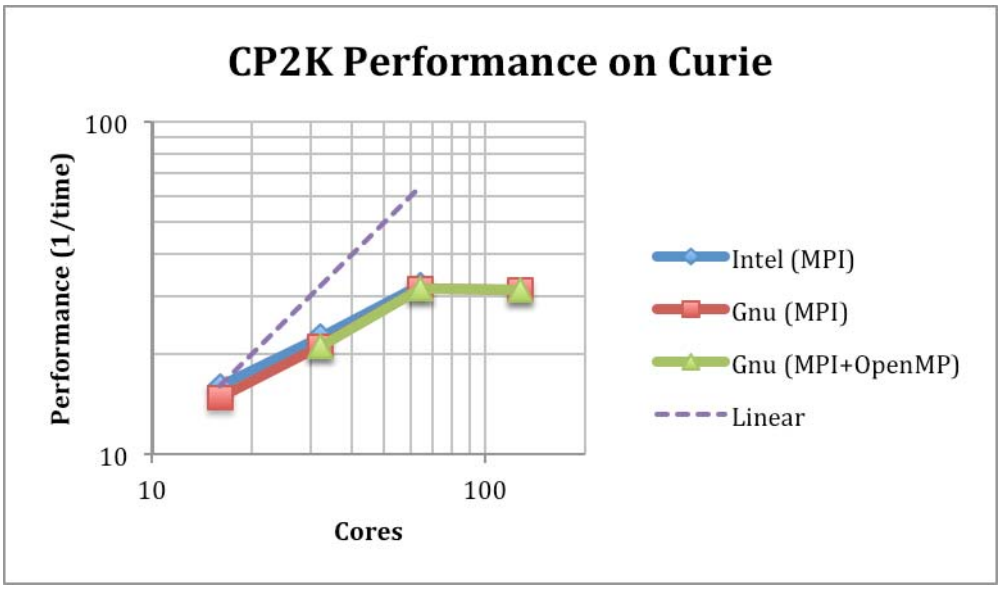



Figure 2: Performance results of CP2K on CURIE

After updating the source code with the changes described above and after dropping further investigations on the versions of CP2K linked with the custom built BLAS and LAPACK libraries a complete regression test was resubmitted. The results of this new test are shown in Figure 3. Once again the numerical errors when using the MPI+OpenMP version were investigated and it was concluded that these could be safely ignored. The arch files used to build these 5 "production" CP2K binaries are given in Appendix B.

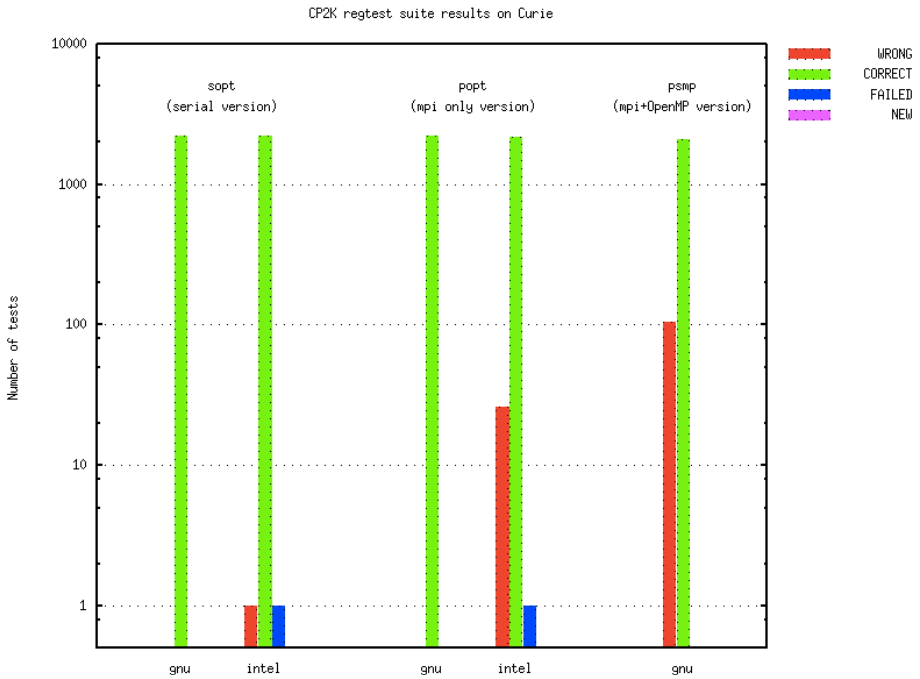



Figure 3: Final CP2K regression test suite results

## 3. Using OpenMP in CP2K for improved scalability on multi-core clusters

Prior work by the author [5][6] has already introduced or improved OpenMP parallelisation in various areas of CP2K. However, it has also identified several other region of code which take up significant amounts of time for certain calculations and for which OpenMP parallelization could be added or improved. The work done to address these areas is detailed in the following sections.

### *3.1. Exchange-correlation functionals*

An important part of any Density Functional Theory calculation is the contribution to the energy caused by the interactions between pairs of electrons, approximated in DFT by the Exchange-Correlation functional. There are many variants of these in use, which may give more accurate energies for certain types of system. In CP2K 26 different functionals are implemented, and can be selected by the `XC_FUNCTIONAL` keyword in a CP2K input file. One of the most commonly used (the PBE functional) was parallelized in [5], but many of the others were not, or were only partially parallelized.

After surveying the current state of OpenMP for each XC functional, we fully parallelized all the functionals using OpenMP, and validated the results of these changes by running the CP2K regression test suite. A summary of the changes made can be found in Table 1.

Table 1. Overview of parallelization of XC functionals

| Input keyword | File | OpenMP state |
|---|---|---|
| BECKE88 | xc_xbecke88.F | Refactored |
| BECKE88_LR | Xc_xbecke88_long_range.F | Added |
| BECKE88_LR_ADIABATIC | xc_xbecke88_lr_adiabatic.F | Added |
| BECKE_ROUSSEL | xc_b97.F | Added |
| BECKE97 | xc_xbecke_roussel.F | Added |
| CS1 | xc_cs1.f | Completed |
| GV09 | xc_xbr_pbe_lda_hole_t_c_lr.F | Added |
| HCTH | xc_hcth.F | Present |
| KE_GGA | xc_ke_gga.F | Completed |
| LDA_HOLE_T_C_LR | xc_xlda_hole_t_c_lr.F | Added |
| LYP | xc_lyp.F | Added |
| LYP_ADIABATIC | xc_lyp_adiabatic.F | Added |
| OPTX | xc_optx.F | Added |
| P86C | xc_perdew86.F | Present |
| PADE | xc_pade.F | Present |
| PBE | xc_pbe.F | Present |
| PBE_HOLE_T_C_LR | xc_xpbe_hole_t_c_lr.F | Added |
| PW92 | xc_perdew_wang.G | Present |
| PZ81 | xc_perdew_zunger.F | Present |
| TF | xc_thomas_fermi.F | Present |
| TFW | xc_ftw.F | Present |
| TPSS | xc_tpss.F | Completed |
| VWN | xc_vwn.F | Completed |
| XALPHA | xc_xalpha.F | Present |
| XGGA | xc_exchange_gga.F | Completed |
| XWPBE | xc_xwpbe.F | Added |

The OpenMP performance obtained was overall very good. For two typical examples taken from the CP2K test directories `QS/regtest-hole-funct` and `QS/regtest-hybrid` we show the time taken and

speedup for various numbers of threads in Table 2. These tests were run on a Cray XE6 with 24 cores per node (two 12-core processors). Achieving over 95% efficiency within a single NUMA region, and up to 92% across an entire node is an excellent result and will ensure that XC functional evaluation is much less likely to contribute significantly to the runtime of simulations using hybrid MPI and OpenMP.

Table 2. XC functional OpenMP performance

| Number of Threads | 1 | 2 | 4 | 6 | 8 | 12 | 16 | 24 |
|---|---|---|---|---|---|---|---|---|
| `H2O_GV09_1.0.inp` (Time / s) | 26.7 | 13.4 | 6.8 | 4.6 | 3.5 | 2.4 | 1.8 | 1.2 |
| (Speedup) | 1.00 | 1.99 | 3.90 | 5.79 | 7.72 | 11.43 | 15.14 | 22.14 |
| `Li-hybrid-rcam-b3lyp.inp` (Time /s) | 5.32 | 2.69 | 1.38 | 0.94 | 0.72 | 0.50 | 0.40 | - |
| (Speedup) | 1.00 | 1.98 | 3.85 | 5.65 | 7.39 | 10.57 | 13.33 | - |

### *3.2. Realspace grid operations*

Another key element within CP2K is the use of realspace grids, which are a key step in the transformation from the plane-wave basis (stored as complex values on planewave grids), to the atom-centered Gaussian basis (stored as coefficients in a sparse matrix). In the routine `calculate_rho_elec`, Gaussian basis functions are received in matrix form and are written to the realspace grids in parallel by the team of OpenMP threads. Details of how this is achieved can be found in [5], but since in general each Gaussian can overlap any other, each thread writes to its own copy of the grids (the `lgrid`), which are then summed onto the final, output grid. This had been observed to be a costly operation which did not scale well with increasing numbers of OpenMP threads, so it was proposed to replace the existing reduction algorithm with a tree-based one which would minimize the total volume of data copied, and in particular reduce copies from one NUMA region to another.

Three new tree-reduction algorithms were implemented and compared with the existing implementation, hereafter referred to as Version 0.

In Version 1 the strategy was for each thread to take responsibility for a given region of the real space grid, and accumulate the total to the first `lgrid`. This has the advantage that it is in principle load balanced for even numbers of threads, and dealing with unusual numbers of threads is painless. The disadvantage is that each thread must read data from (and write read to) other threads' `lgrid`. This is illustrated in Figure 4. For four threads there are two levels of the tree. The first level (top) includes two copies for each thread. The z-decomposition of the local grid is represented in the vertical direction, with each thread being responsible for a fixed portion

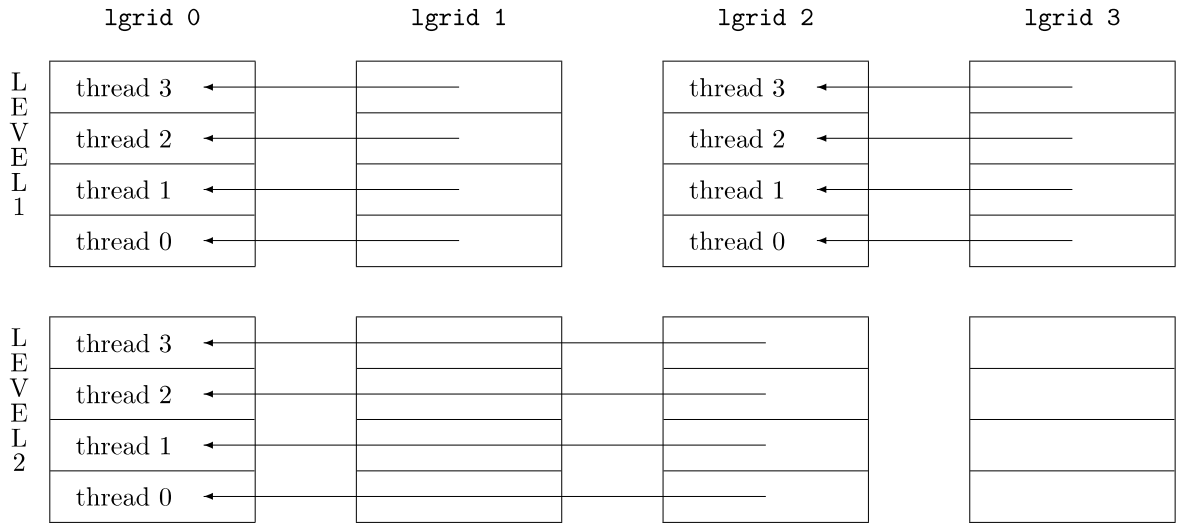
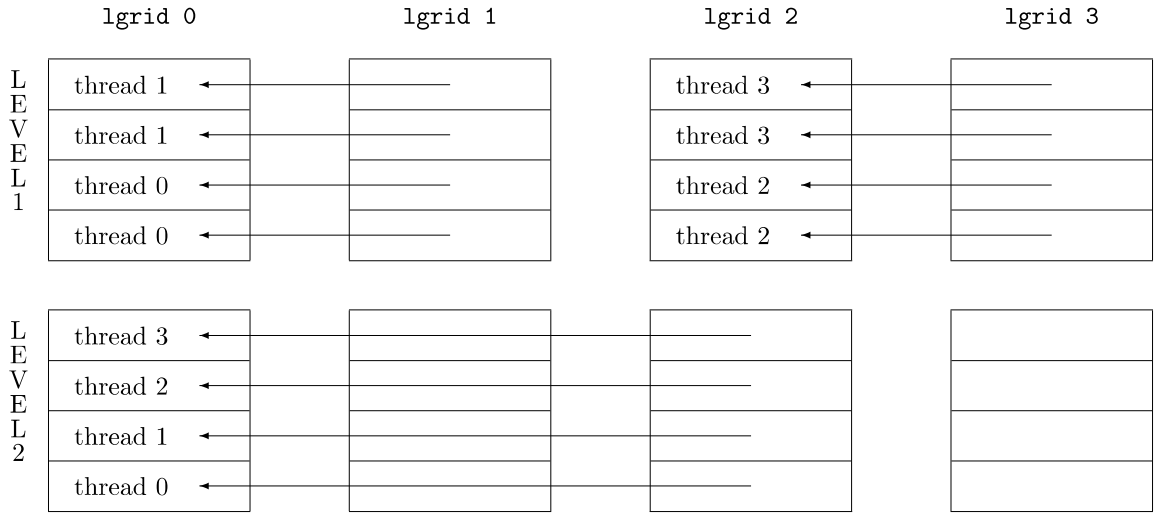



Figure 4: Schematic of tree reduction (Version 1)

In Version 2 an attempt was made to achieve some data locality like the original implementation. At each level, a subset of the threads works on a given `lgrid`. Care must be taken that it does not exceed the number of threads available at that level when there are non power-of-two numbers of threads. This is illustrated in Figure 5. At level 1, thread 0 and thread 1 are responsible for the accumulations between `lgrid` 0 and `lgrid` 1, and so on. Contiguous regions of the `lgrid` are coalesced into a single copy (using `daxpy()`).




5igure 5: Schematic of tree reduction (Version 2)

In Version 3 each entire `lgrid` is accumulated by the "owning" thread. This leaves progressively more threads without work. Even-numbered threads perform a single `daxpy()` at the different levels. This is illustrated in Figure 6.

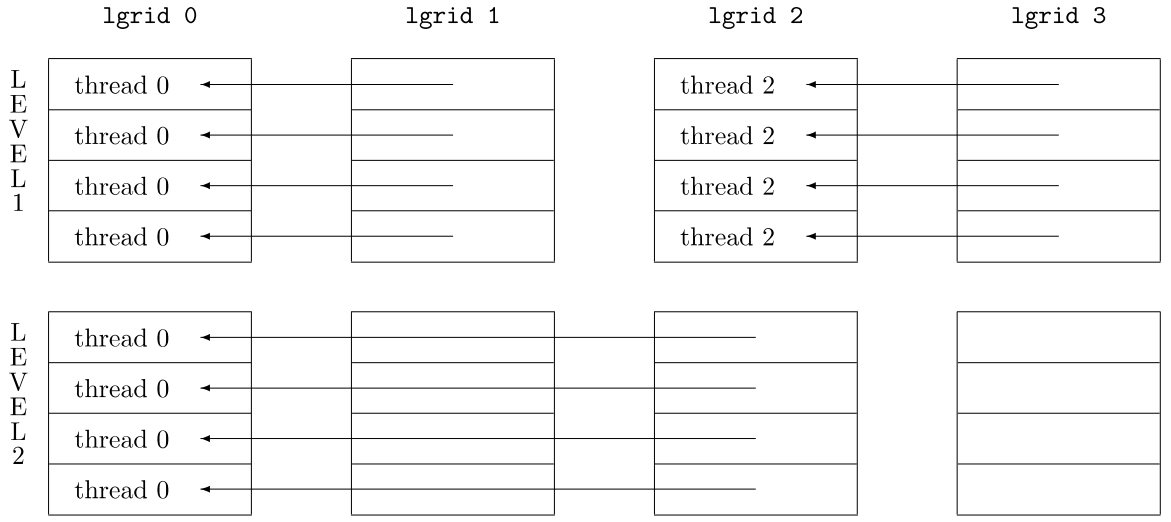



Figure 6: Schematic of tree reduction (Version 3)

The performance of each of the versions was measured using the aforementioned H2O-64 input file, running on 36 MPI tasks with a varying number of OpenMP threads. The time taken for the reduction operation is shown in Table 3. Note that Versions 2 and 3 scale much better when there are more threads than in a single NUMA region. Version 2 was selected as it performed marginally better than Version 3, both of which were a significant

improvement over the original implementation.

Table 3. Comparison of time taken (seconds) for parallel grid reduction algorithms

| Number of Threads | 1 | 2 | 4 | 6 | 8 | 12 | 16 | 24 |
|---|---|---|---|---|---|---|---|---|
| Version 0 | 0.0000019 | 0.0028 | 0.0040 | 0.0059 | 0.0069 | 0.0082 | 0.011 | 0.013 |
| Version 1 | 0.0000027 | 0.0026 | 0.0037 | 0.0052 | 0.0069 | 0.0097 | 0.011 | 0.013 |
| Version 2 | 0.0000028 | 0.0029 | 0.0038 | 0.0058 | 0.0062 | 0.0072 | 0.0071 | 0.0083 |
| Version 3 | 0.0000027 | 0.0026 | 0.0039 | 0.0061 | 0.0062 | 0.0073 | 0.0073 | 0.0084 |

As a side-effect of the detailed profiling carried out during the development of the new grid reduction algorithm, it was observed that the main performance bottleneck in `calculate_rho_elec` was not the reduction, but in fact the initial zeroing of the `lgrid`s before new data is written to them. The cause of this behavior is due to the fact that the `lgrid`s are local variables and are allocated, used, and deallocated each time `calculate_rho_elec` is called. Since the `lgrid`s are in fact a large shared array (to facilitate the reduction), rather than OpenMP private arrays, this means they are allocated once outside the parallel region, and then the first time they are accessed by the team of threads (during the zeroing), there is a large cost associated with pulling data which is local to thread 0 - the allocating thread - into the thread's own local cache hierarchy.

To avoid this, a refactoring of the realspace grid types was made so that the `lgrid`s (and the `rs_grid`s) are no longer repeatedly created and destroyed, but rather persist for the lifetime of the application. As well as amortising the large 'first access' cost for the `lgrid`s, this also saves a great deal of memory allocation and deallocation. The effect of this change can be clearly seen in Figure 7, giving around a 10x speedup over the original code.

One final optimization was implemented in this region of the code. In the routine `distribute_matrix` all the matrix data required for each process to complete its assigned tasks of mapping Gaussians onto the realspace grid is packed into a buffer and distributed via an `MPI_Alltoallv` call. However, due to the construction of the domain decomposition, most processes already have most of the data they need, and only a relatively small fraction of the data is actually sent to or received from remote processes. For the local data, there is no need to pack it into the buffer and unpack it again, so potentially two large data copies can be avoided. In addition, since the packing and unpacking is done in a parallel loop over processes, having a single process with much more data that the others causes severe thread load-imbalance when using the default static OpenMP loop schedule.

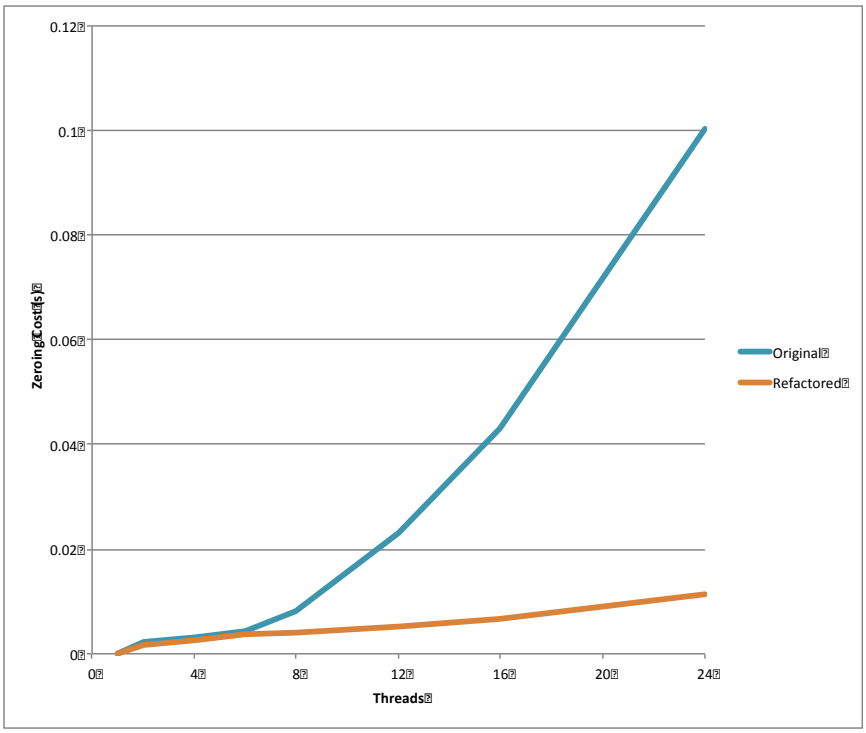



Figure 7: Comparison of `lgrid` initialisation costs in original code vs refactored version

By avoiding copying the local data, approximately 5% improvement was observed for the H2O-64 test case when running on a single thread per process, rising to 35% on 8 threads. In principle, this data is no longer required to be passed to the `MPI_alltoallv` (which might save a further copy within MPI), but in practice it was found that omitting the data (i.e. setting the local send and recv sizes to 0) actually caused the `MPI_alltoallv` to take twice as long. This is believed to be because reducing the data size causes MPI to choose a different, less efficient communication algorithm. Since the local copy is essentially free - it takes less time than the latency for the remote data to arrive - a compromise was made where the `MPI_alltoallv` still handles the large local data block, but it is never read or written by the packing and unpacking loops.

### *3.3. Core Hamiltonian calculations*

Calculation of the Core Hamiltonian matrix can take a significant amount of time for certain calculations, particularly those with large basis sets, or when using OpenMP since there was no existing OpenMP parallelization in this region of the code. The goal was to introduce OpenMP loop-level parallelisation to the `build_core_ppl` subroutine in the file `core_ppl.F`. It was known that the program spent a considerable amount of time in this loop and furthermore, the approach used here could be adapted with only minimal changes in other parts of the code.

The main change to the code is the introduction of a parallel region around a loop over all of the particles in a neighbor list, that is the loop beginning, "`DO WHILE (neighbor_list_iterate(nl_iterator)==0)`". One of the main challenges in such a parallelisation is determining the breakdown between shared and private variables. The choice as to which class the variables fall into was determined through inspection of the code. Also, the code uses an iterator object to iterate over a fairly complex data structure, so it is difficult to tell a priori to what extent the iterations of the loop are independent.

It was determined from inspection of the code, and from examining the output from instrumented test runs, that it was possible to break down the iterations of the loop into independent tasks each corresponding to one or more iterations of the original loop. The modification to the code therefore consists of:

- Introducing a new data structure to hold the data describing a task
- Iterating over the data structure in serial, building an array of tasks
- Introducing a parallel region around a new loop which loops over the independent tasks

In fact, the tasks are not completely independent, as they all update a shared array force containing forces between particles. This is accounted for by introducing OpenMP critical regions within the parallel region around the relevant updates.

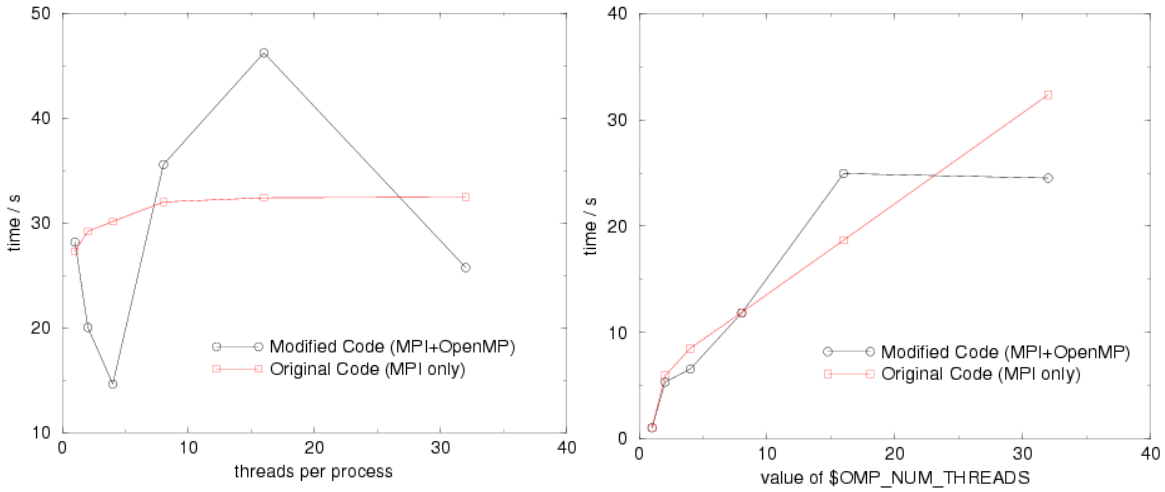



Figure 7: Performance comparisons of original and modified code. (a) 16 MPI processes, varying number of threads; (b) 512 total cores, varying number of MPI processes and threads

The performance of the code was measured in a series of timing runs performed on the HECToR Test and Development system (TDS) using the `H2O-dfs-ls-2` benchmark, a linear-scaling DFT calculation of 256

water molecules using the `MOLOPT` basis set distributed with CP2K. Tests were performed with the modules listed in Appendix C. During the course of the work described here, the HECToR TDS contained nodes with either AMD Magny-Cours or Interlagos processors. The results described here were obtained using only the newer Interlagos nodes. These are similar to the processors found in the PRACE HERMIT Tier-0 system. Timings were obtained by inserting CrayPAT calls around the whole of the contents of the build_core_ppl routine, and show the performance of this region in isolation from the rest of the code. Results are shown in Figure 7.

The results imply that, for a fixed number of MPI processes, there is a performance benefit from running with two or four cores per process. It also appears that by running on 32 threads per process, a performance benefit can be obtained. It is not understood why the performance is worse for 8 and 16 threads than it is for 32 threads.

Other than the speedup observed with two and four threads per process, the speed-up results were disappointing, so an alternative parallelisation was also attempted. After further inspection of the code, it appeared that it would be possible to use the iterator as it currently exists in the code from within a parallel region, using its optional arguments to specify the thread from which it was being accessed. The planned changes were made to the code but unfortunately this version of the code does not yet work, crashing when it is run. Some work was put in to debug the code, but without success. Further effort would be required in order to debug the code in order to find the cause of the crash.

## 4. Conclusion

As a result of continued PRACE support through Task 7.2 (“Applications Enabling with Communities”) in the 1st implementation phase (PRACE-1IP), we have improved the performance of CP2K by additional parallelization and optimization targeted at mixed-mode OpenMP/MPI usage of the code, suitable more multi-core HPC platforms such as the CURIE and HERMIT systems available to the PRACE community. Specifically, we have parallelized the full range of 24 Exchange-Correlation (XC) functionals available in CP2K, which will give better performance across a wider range of use cases that are of interest to the materials science and computational chemistry communities. We have also improved the performance of key grid operations which are core to the Quickstep DFT implementation – these will impact almost all users of the code. Thanks to ongoing close collaboration with the CP2K development team, these improvements have been incorporated in the CP2K source code, and are already available to users of the code. Work was begun on parallelization of the core Hamiltonian calculation, but the expected performance improvement was not obtained, and work is ongoing to resolve this.

To maximize the benefits to the scientific communities supported by PRACE, a recent version of CP2K has been installed on the CURIE system and is centrally available to all users via the module environment. We hope this report will help emphasise to the user community the benefits of using optimized and scalable software like CP2K on the PRACE infrastructure. As a result of this work, a successful Preparatory Access Type C project (“High Performance MP2 for condensed phase simulations”) has been awarded, allowing us to provide further development directly in support of ongoing research using CP2K.

## 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-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources CURIE (CEA, France).

We are very grateful to Prof. Joost VandeVondele (ETH Zürich) and Prof. Jürg Hutter (Univ. Zürich) for help and advice on code development and for access to the Cray XE6 at CSCS for development and testing.

## References

1. 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).
2. CP2K: High Performance Computing, http://www.nanosim.mat.ethz.ch/research/CP2K
3. Million Atom KS-DFT with CP2K, Iain Bethune, Adam Carter, Xu Guo, and Paschalis Korosoglou, 2012, http://www.training.prace-ri.eu/uploads/tx_pracetmo/CP2K.pdf
4. Libint library, http://sourceforge.net/projects/libint/
5. Improving the scalability of CP2K on multi-core systems: A dCSE Project, 2010, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k02/cp2k02_final_report.pdf
6. CP2K - Sparse Linear Algebra on 1000s of cores: A dCSE Project, 2012, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k03/cp2k03.pdf

## Appendix A

Configure file used for building libsmm using GNU Compiler suite and MKL libraries on CURIE:

```
transpose_flavor=1
data_type=1
target_compile="gfortran -O2 -funroll-loops -ffast-math -ftree-vectorize -march=native -
fno-inline-functions"
OMP_NUM_THREADS=1
blas_linking="$MKL_LIBS"
dims_small="1 4 5 6 9 13 16 17 22"
dims_tiny="1 2 3 4 5 6 7 8 9 10 11 12"
host_compile="gfortran "
tasks=32
```

Configure file used for building libsmm using Intel Compiler suite and MKL libraries on CURIE:

```
transpose_flavor=1
data_type=1
target_compile="ifort -O2 -unroll -xSSE4.2"
OMP_NUM_THREADS=1
blas_linking="$MKL_LIBS"
dims_small="1 4 5 6 9 13 16 17 22"
dims_tiny="1 2 3 4 5 6 7 8 9 10 11 12"
host_compile="gfortran "
tasks=32
```

## Appendix B

Arch file used for building serial version (**sopt**) using GNU Compiler suite and MKL libraries on CURIE:

```
# On CURIE use the following commands prior to make:
#
# $ module load fftw3
# $ module load gnu
#
CC       = cc
CPP      =
FC       = gfortran
LD       = gfortran
AR       = ar -r
CPPFLAGS = -I$(FFTW3_INC_DIR)
DFLAGS   = -D__GFORTRAN -D__FFTSG -D__FFTW3 -D__HAS_smm_dnn -D__LIBINT
FCFLAGS  = $(CPPFLAGS) $(DFLAGS) -O2 -ffast-math -funroll-loops -ftree-vectorize -
march=native -ffree-form
LDFLAGS  = $(FCFLAGS)
LIBS     = $(MKL_LIBS) \
           $(FFTW3_LIB_DIR)/libfftw3.a \
           $(WORKDIR)/libssm/gfortran-mkl/lib/libsmm_dnn.a \
           $(WORKDIR)/libint/gnu/lib/libderiv.a \
           $(WORKDIR)/libint/gnu/lib/libint.a \
           -lstdc++

OBJECTS_ARCHITECTURE = machine_gfortran.o
```

Arch file used for building MPI only version (**popt**) using GNU Compiler suite and MKL libraries on CURIE:

```
# On CURIE use the following commands prior to make:
#
# $ module load fftw3
# $ module load gcc
# $ export OMPI_MPIFC=gfortran
#
CC       = cc
CPP      =
FC       = mpif90
LD       = mpif90
AR       = ar -r
CPPFLAGS = -I$(FFTW3_INC_DIR)
DFLAGS   = -D__GFORTRAN -D__FFTSG -D__parallel -D__BLACS -D__SCALAPACK -D__FFTW3 -
D__HAS_smm_dnn -D__LIBINT
FCFLAGS  = $(CPPFLAGS) $(DFLAGS) -O2 -ffast-math -funroll-loops -ftree-vectorize -
march=native -ffree-form
LDFLAGS  = $(FCFLAGS)
LIBS     = $(MKL_SCA_LIBS) \
           $(FFTW3_LIB_DIR)/libfftw3.a \
           $(WORKDIR)/libssm/gfortran-mkl/lib/libsmm_dnn.a \
           $(WORKDIR)/libint/gnu/lib/libderiv.a \
           $(WORKDIR)/libint/gnu/lib/libint.a \
           -lstdc++

OBJECTS_ARCHITECTURE = machine_gfortran.o
```

Arch file used for building MPI+OpenMP version (**psmp**) using GNU Compiler suite and MKL libraries on CURIE:

```
# On CURIE use the following commands prior to make:
#
# $ module load fftw3
# $ module load gcc
# $ export OMPI_MPIFC=gfortran
#
CC       = cc
CPP      =
FC       = mpif90 -fopenmp
LD       = mpif90 -fopenmp
AR       = ar -r
CPPFLAGS = -I$(FFTW3_INC_DIR)
DFLAGS   = -D__GFORTRAN -D__FFTSG -D__parallel -D__BLACS -D__SCALAPACK -D__FFTW3 -
D__LIBINT
FCFLAGS  = $(CPPFLAGS) $(DFLAGS) -O3 -ffast-math -funroll-loops -ftree-vectorize -
march=native -ffree-form
LDFLAGS  = $(FCFLAGS)
LIBS     = $(MKL_SCA_LIBS) \
           $(FFTW3_LIB_DIR)/libfftw3.a \
           $(WORKDIR)/libssm/gfortran-mkl/lib/libsmm_dnn.a \
           $(WORKDIR)/libint/gnu/lib/libderiv.a \
           $(WORKDIR)/libint/gnu/lib/libint.a \
           -lstdc++

OBJECTS ARCHITECTURE = machine gfortran.o
```

Arch file used for building serial version (**sopt**) using Intel Compiler suite and MKL libraries on CURIE:

```
# On CURIE use the following commands prior to make:
#
# $ module load fftw3
#
CC       = cc
CPP      = cpp
FC       = ifort
LD       = ifort
AR       = ar -r
DFLAGS   = -D__INTEL -D__FFTSG -D__FFTW3 -D__HAS_smm_dnn -D__LIBINT
CPPFLAGS = -C -traditional $(DFLAGS) -I$(FFTW3_INC_DIR)
FCFLAGS  = $(DFLAGS) -O2 -xSSE4.2 -heap-arrays 64 -funroll-loops -fpp -free
FCFLAGS2 = $(DFLAGS) -O1 -xSSE4.2 -heap-arrays 64 -fpp -free
LDFLAGS  = $(FCFLAGS)
LIBS     = $(MKL_LIBS) \
           -L$(FFTW3_LIB_DIR) -lfftw3 \
           -L$(WORKDIR)/libssm/intel/lib -lsmm_dnn \
           -L$(WORKDIR)/libint/intel/lib -lderiv -lint -lstdc++

OBJECTS_ARCHITECTURE = machine_intel.o

graphcon.o: graphcon.F
       $(FC) -c $(FCFLAGS2) $<

qs_vxc_atom.o: qs_vxc_atom.F
       $(FC) -c $(FCFLAGS2) $<
```

Arch file used for building the MPI only version (**popt**) using Intel Compiler suite and MKL libraries on CURIE:

```
# On CURIE use the following commands prior to make:
#
# $ module load fftw3
#
CC       = cc
CPP      = cpp
FC       = mpif90
LD       = mpif90
AR       = ar -r
DFLAGS   = -D__INTEL -D__FFTSG -D__parallel -D__BLACS -D__SCALAPACK -D__FFTW3 -
D__HAS_smm_dnn -D__LIBINT
CPPFLAGS = -C -traditional $(DFLAGS) -I$(FFTW3_INC_DIR)
FCFLAGS  = $(DFLAGS) -O2 -xSSE4.2 -heap-arrays 64 -funroll-loops -fpp -free
FCFLAGS2 = $(DFLAGS) -O1 -xSSE4.2 -heap-arrays 64 -fpp -free
LDFLAGS  = $(FCFLAGS)
LIBS     = $(MKL_SCA_LIBS) \
           -L$(FFTW3_LIB_DIR) -lfftw3 \
           -L$(WORKDIR)/libssm/intel/lib -lsmm_dnn \
           -L$(WORKDIR)/libint/intel/lib -lderiv -lint -lstdc++

OBJECTS_ARCHITECTURE = machine_intel.o

graphcon.o: graphcon.F
        $(FC) -c $(FCFLAGS2) $<

qs_vxc_atom.o: qs_vxc_atom.F
        $(FC) -c $(FCFLAGS2) $<
```

## Appendix C

This appendix shows the modules loaded during measurements of the performance of `build_core_ppl`.

```
Currently Loaded Modulefiles:
  1) modules/3.2.6.6                                                      15) xt-mpich2/5.4.3
  2) nodestat/2.2-1.0400.29866.4.3.gem                                    16) atp/1.4.2
  3) sdb/1.0-1.0400.30000.6.18.gem                                        17) xt-asyncpe/5.07
  4) MySQL/5.0.64-1.0000.4667.20.1                                        18) pmi/3.0.0-1.0000.8661.28.2807.gem
  5) lustre-cray_gem_s/1.8.4_2.6.32.45_0.3.2_1.0400.6221.1.1-1.0400.30252.1.29  19) xt-libsci/11.0.05
  6) udreg/2.3.1-1.0400.3911.5.6.gem                                      20) gcc/4.6.2
  7) ugni/2.3-1.0400.3912.4.29.gem                                        21) pbs/11.2.0.113417
  8) gni-headers/2.1-1.0400.3906.5.1.gem                                  22) packages-phase2b
  9) dmapp/3.2.1-1.0400.3965.10.12.gem                                    23) budgets/1.0
 10) xpmem/0.1-2.0400.29883.4.6.gem                                       24) xtpe-interlagos
 11) hss-llm/6.0.0                                                        25) fftw/3.3.0.0
 12) Base-opts/1.0.2-1.0400.29823.8.1.gem                                 26) papi/4.2.0
 13) xtpe-network-gemini                                                  27) perftools/5.3.0
 14) PrgEnv-gnu/4.0.30
```