Technical Report for ARCHER2-eCSE08-10: Improving multi-threaded scaling of CONQUEST

Tuomas Koskela[1], Ilektra Christidi[1], Connor Aird[1], and David Bowler[2]

[1]**Advanced Research Computing Centre, UCL**
[2]**London Centre for Nanotechnology, UCL**

ABSTRACT

This report describes the work done to improve the multi-threaded performance of the CONQUEST large-scale DFT code under the eCSE08-10 project. The aim of the project was to improve the scalability and sustainability of CONQUEST. Scalability on modern CPU architectures with MPI only parallelism was found to scale poorly. To improve the scaling, previous work on OpenMP threading of matrix multiply kernels was refactored and expanded to cover more compute-intensive kernels in the code. We also sought to further improve the MPI scaling by implementing and testing the efficacy of the overlap of communication and calculation in matrix multiplication. The sustainability of the code was improved by introducing automated testing and build system automation. Automated tests now run in a continuous integration workflow on all pull requests to the main development branch on GitHub. The Makefile build system has been documented and version controlled on various HPC systems and automation with the Spack HPC package manager has been introduced.

Keywords: eCSE, ARCHER2, CONQUEST

1 INTRODUCTION

CONQUEST is a large-scale and linear-scaling density functional theory (DFT) code, which can model systems of up to 10,000 atoms with diagonalisation, and which has demonstrated calculations on more than 2,000,000 atoms with linear scaling, where the upper limit is set by the number of processes available. In linear scaling mode, it shows exceptional weak scaling, demonstrated on just over 196,000 processes on the K computer.

The main computational load in linear scaling mode is matrix multiplication (MM), while for diagonalisation mode it is the eigensolver from SCALAPACK (pzhegvx); much of the remaining computational time is spent on grid-based operations. The parallelisation was designed at a time when there were relatively few cores per node (typically 4-8), and the strong and weak scaling of CONQUEST can be adversely affected by large numbers of MPI processes per node: the main issue is contention between MPI processes for communications bandwidth. A previous project showed that combining MPI and OpenMP threading could restore the weak scaling for MM for fixed process count (reducing the number of MPI processes per node, and introducing threads for MM). However, other parts of the code which were not multi-threaded necessarily performed more slowly on low numbers of MPI processes per node.

This report documents the findings and improvements made to CONQUEST in the eCSE08 project ”Improving multi-threaded scaling of CONQUEST”. The overall objective of the project was to improve the scalability and sustainability of CONQUEST, particularly improving its threaded performance on architectures with large core counts. The work was done in collaboration between the PI, professor David Bowler, and a team of research software engineers, Tuomas Koskela, Ilektra Christidi and Connor Aird from the UCL Centre for Advanced Research Computing.

2 SUSTAINABILITY IMPROVEMENTS

The sustainability of the code was improved by introducing a suite of tests and benchmarks, a continuous integration workflow and build system automation.

2.1 Suite of tests for Continuous Integration

Input and reference output files for three end-to-end tests were added to the main development branch of CONQUEST. The first two tests are simple eight atom bulk Si systems (one atom moved off site to ensure non-zero forces) for both diagonalisation and linear scaling (#184). The third test tests polarisation for a BaTiO3 system (#199). Unit tests were considered but deemed to be out of scope for the project due to the substantive refactoring that would have been needed (#187).

A continuous integration workflow was then added on GitHub actions (#189). It checks out the code, builds it, runs the tests and verifies the results using a python script that is executed with pytest (#192). The tests are executed in both serial and parallel on two processes with both MPI and OpenMP. The test suite can be extended when new code features are implemented. The process of adding new tests is documented in the README file of the testsuite. (#201). The GitHub actions workflow is set to run on all pull requests and pushes to the develop branch to detect bugs before new code is merged. By default the tests are built using the `default` matrix multiply kernel. A feature was later added to enable testing of all multiply kernels (#292). This requires launching the workflow manually with the `multiply_kernel_test` variable set to true in the GitHub UI. See Section 3.3 for more details on multiply kernels.

2.2 Store inputs for performance benchmarks

A set of performance benchmarks was added under the benchmarks directory in the repository, so that they are easier to reference and share, and reuse in later projects (#261). For details of the benchmarks, see the README.md files in the `benchmarks/` directory. The benchmarks include a weak scaling benchmark `matrix_multiply`, that contains a set of input files scaling up from 64 to 262144 atoms that can be used to run a weak scaling study. The CONQUEST performance benchmarks have been automated using ReFrame in the excalibur-tests project. See documentation of the project for more details.

2.3 Build system improvements

Version control of host specific build files To build CONQUEST, the user provides a `system.make` file that contains system-specific information, such as compilers, flags, paths to libraries etc. Originally this was done manually by each user based on documentation in the repository. As part of this project, we have written `system.make` files for commonly used HPC systems, including ARCHER2, and added them into the repository (#272). The build system has been refactored such that the system being built on is selected by setting a variable. Automatic detection of the host system was investigated, but found unpractical by the users of CONQUEST.

Spack package A Spack package has been developed and merged upstream to spack (#40718) such that CONQUEST and all of its dependencies can be installed with Spack. The CONQUEST package requires Spack v0.21 or later. If Spack isn't available or up to date on the system, it is relatively straightforward to install it with user permissions following the install instructions. The installation documentation has been updated to include installing with Spack. More details can be found in the Spack CONQUEST package.

3 PERFORMANCE IMPROVEMENTS

3.1 Performance bottlenecks in CONQUEST

Performance profiling with Intel VTune, Intel Advisor, Linaro MAP and Scalasca were heavily utilised in the development and optimisation workflow. We verified that matrix multiplications, SCALAPACK diagonalisation and grid-based operations were significant performance bottlenecks in performance benchmarks that were introduced (#262). We did not find significant time being spent in Fourier transforms in these benchmarks, contrary to what was previously assumed.

3.2 Multi-threading grid-based operations

Thread loops over blocks The `K222_G200` benchmark was used as a reference when analysing performance of loops over grid blocks. In the benchmark, four subroutines, summarized in table 1, were identified as spending significant time in loops over grid blocks and selected for multi-threading.

In `calc_matrix_elements_module`, threading has been implemented in the subroutines `get_matrix_elements_new` and `act_on_vectors_new` (#195). Both subroutines have a deep loop nest where the loop over grid blocks is the outermost loop. In `get_matrix_elements_new` the

subroutine	fraction of total run time
act_on_vectors_new	28.3 %
get_matrix_elements_new	8.0 %
single_pao_to_grad	35.7 %
single_pao_to_grid	4.1 %

Table 1. Subroutines selected for multi-threading

innermost loop accumulates data in send_array, therefore a reduction is done at the end of the parallel region. Array reductions are natively supported in Fortran, but it was found that on some systems the OMP_STACKSIZE environment variable has to be set to have enough memory available for the array reduction #267. In get_matrix_elements_new, gridfunctions%griddata is updated with a gemm call in the innermost loop. The updates are done in separate indices by each loop iteration, it can be safely updated by all threads in parallel.

In PAO_grid_transform_module, the single_PAO_to_grid subroutine has been rewritten so that threading over blocks can be done. The innermost loop updates gridfunctions%griddata with each loop iteration updating at a different index. However the index was calculated sequentially inside the loop, which made it unsafe for threading. The rewritten subroutine first precomputes the indices and stores them in an array, then does the loop over blocks which can now safely be threaded (#245). The single_PAO_to_grad subroutine was removed in and the functionality merged with single_PAO_to_grid (#251).

Reduce code duplication in PAO_grid_transform_module The subroutines single_PAO_to_grid and single_PAO_to_grad were found to duplicate almost all of their code with only a difference of a subroutine call in the innermost loop (#244). They have been combined to a single subroutine PAO_or_gradPAO_to_grid. (#251). An interface for the called subroutine has been defined, and the subroutine to call is passed as an argument to PAO_or_gradPAO_to_grid. Any subroutine can be passed, as long as it conforms to the interface. In addition to reducing code duplication, this got rid of OpenMP overheads in single_PAO_to_grad by using the refactored code written for single_PAO_to_grid.

Table 2 shows a performance comparison to the development branch. These were run on the MMM hub Young cluster using 8 MPI ranks. Both serial and multi-threaded performance is improved, with a 2.6x parallel speedup on 4 threads.

	develop	optimised
Without -fopenmp	95.331s	63.482s
With -fopenmp, 1 OMP thread	95.597s	62.324s
With -fopenmp, 4 OMP threads		36.491s

Table 2. Performance comparison before and after optimisation of grid-based loops

3.3 Optimising the performance of matrix multiply

The matrix_multiply benchmark was used as a reference when analysing performance of the matrix multiplication kernels. In this benchmark, main bottlenecks are the multiply_kernel_* kernels, and MPI communications. We investigated the performance of the previously implemented multi-threaded matrix multiply kernels, and aimed to further improve the multi-threaded performance.

Reducing OpenMP overhead The creation of the !$omp parallel region was moved out of the multiply kernels, to the main loop in multiply_module and orphaned !$omp do directives were used on loops inside the multiply kernels (#266). This was done to reduce the overhead of spawning threads each time the multiply kernel is called. Since the main loop handles both MPI communications and the multiply kernels, the MPI communications were wrapped in omp master directives, to restrict them to the main thread. To do that, we had to introduce OpenMP barriers around the MPI communication to ensure data has arrived before distributing work to OpenMP threads. This was previously guaranteed because the communication was done outside the parallel region.

Overlapping communication and computation The multiply_module is already threaded with OpenMP as described in the previous section. However, in every iteration over partitions, the data has to first be brought from remote processes via MPI blocking point-to-point communications before the multi-threaded computation can begin, in order to use the correct data.

We investigated overlapping the communication and computation in this module (#265 and #290), by allowing data from one partition to be processed while data from another partition is being received by non-blocking MPI calls. For this scheme to work, we had to double the memory buffers where data are received. Available memory limitations do not allow for a more general scheme, where all data would be received simultaneously and asynchronously, and computations on any partition would commence as soon as the data are safely received.

The matrix_multiply benchmark did not show a performance improvement with the new scheme, although weak scaling on ARCHER2 seemed to stabilise for higher number of threads and processes (see figure 1). Removing MPI barriers from the code where they were not needed for correctness, only marginally improved performance. Delving deeper into code traces and the order in which data are received and processed, revealed that the bottleneck is not actually the time that the communication takes, but the order in which the different data are received: all the non-blocking MPI sends are issued by all processes before the loop, therefore data from any partition can be ready to be received in any order, and not necessarily the one that the loop over partitions imposes (see figure 3) Receiving them in order introduces un-necessary load imbalance in the communication time between processes.

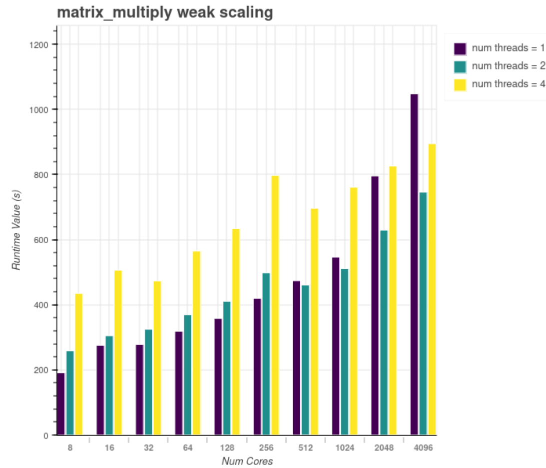

Figure 1. Weak scaling of the matrix multiplication communication-computation overlap.

In future work to optimise the communication-computation interplay in the multiply_module, we recommend that computations are performed in the order data is received, rather than the sequential order that is imposed by the main loop. Such a scheme can be implemented using MPI_Iprobe followed by blocking MPI receives (see preliminary attempts in branch ic-mm-comms-optimise-order).

Optimizing the ompGemm_m multiply kernel The performance of all available multiply kernels was investigated in the matrix_multiply benchmark using 8 MPI ranks and 4 OpenMP threads per rank. The results are summarized in table 3. The ompGemm_m kernel was found to perform the best and was therefore chosen as the target of further optimisation, with the intention that it should be used in most (if not all) cases and the worse performing multiply kernels could be eventually removed.

kernel	default	gemm	ompDoii	ompDoik	ompDoji	ompGemm	ompGemm_m
runtime	140.5s	107.9s	83.0s	69.6s	85.2s	72.2s	69.1s

Table 3. Total runtime of the matrix_multiply benchmark with different multiply kernels. Note that the ompTsk kernel was also available but did not execute successfully.

The ompGemm_m had a memory leak that had to be fixed before doing any optimisation work. The arguments of m_kern_max and m_kern_min contained remote index arrays of fixed size mx_part,

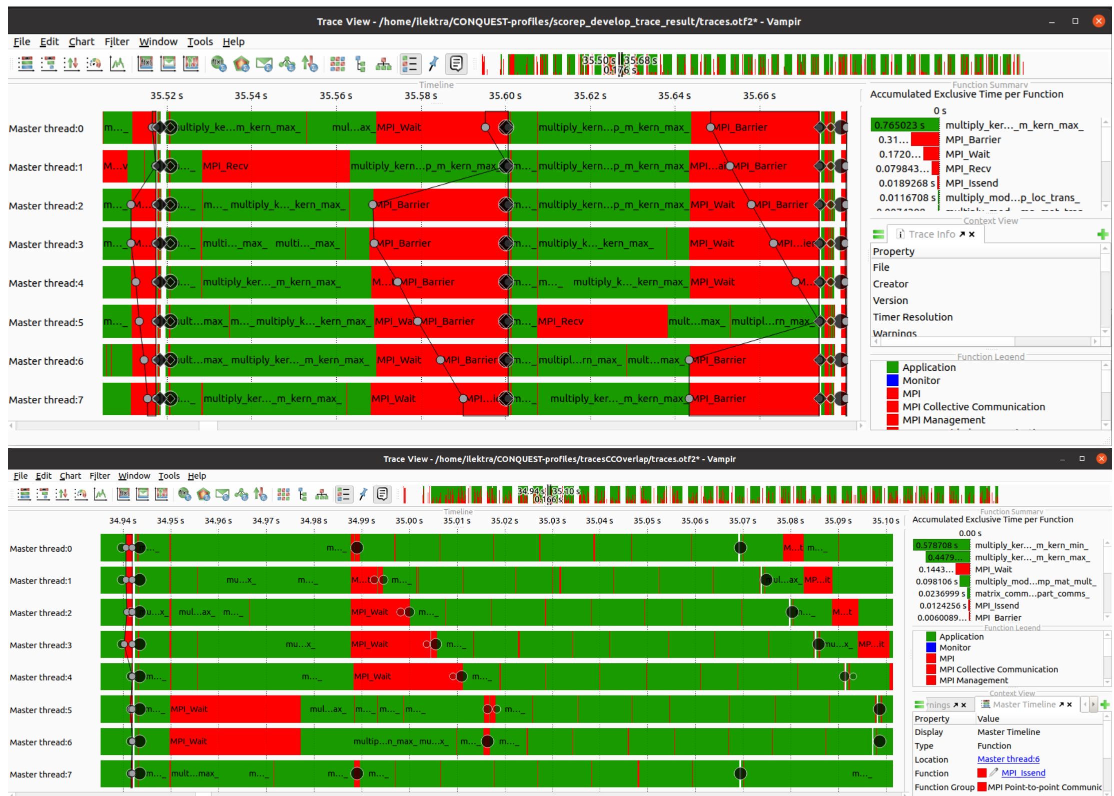

Figure 2. Code traces of two loops in the multiply_module, for 8 MPI processes.
Top: before any changes to implement communication-computation overlap. All the MPI sends are issued at the time of the black circles with rhombuses just after the 35.52s mark. Then each process executes the loop, receiving messages with MPI_Recv from other partitions in order. For most processes this happens quickly and we cannot even see the MPI_Recv in the trace, but process 1 has to wait for considerable time at some point, because the message for that particular partition has not arrived yet. As a result, the rest of the processes have to wait for process 1 to finish at the MPI_Barrier at the end of the loop. This pattern repeats in the next iteration (starting at around 35.60s), with process 5 now causing everyone else to wait.
Bottom: after the communication-computation overlap implementation and removal of MPI barriers. Even though it is harder to identify without any MPI barriers at the end of the loop, we can see that the same communication pattern exists: in the loop starting after the MPI sends are issued at the time of the black circles just after 34.94s, processes 5 and 6 are waiting at an MPI_Wait for messages to arrive. As a result, the rest of the processes have to wait for them to finish at their MPI_Wait before proceeding to the next iteration at the time indicated by the next set of black circles (now not in sync).

but the arrays passed in to the kernel were not always of this size. Because these were passed in as pointers, the size of the array was never explicitly allocated. The arrays were then passed into the `maxval` function to find a size of a another allocatable array. `maxval` looks at all elements from 1 to `mx_part` and for some of the pointers it found uninitialized memory, which would lead to allocation errors. The issue was fixed by using assumed shape arrays for the remote index arrays (#293).

The multiply kernel consists of two subroutines `m_kern_max` and `m_kern_min`. On a high level, the kernel does a sparse matrix-matrix multiplication and accumulates the results in a sparse matrix. We found most of the time spent in the kernel was spent copying data from a sparse matrix storage format to a dense matrix, and vice versa. In the `ompGemm_m` implementation, the multiplication operation itself is done by a BLAS `dgemm` call. We used the Intel Math Kernel Library as the BLAS backend with the assumption that on an Intel system it is sufficiently optimised. The optimisations to the kernel then focus on reducing memory copies as much as possible (#327).

In `m_kern_max` we avoid making temporary copies of the input matrices A and B completely by calling `dgemm` on the whole matrices, including the zero elements. This is the main performance gain. The zero elements are skipped when copying the temporary result back to C, keeping the end result sparse. A and B are stored as 1D arrays which can be passed into `dgemm` as arguments. The only remaining work before the `dgemm` call is computing the start and end indices into A and B. The result is stored into a temporary array and copied into the sparse representation of C after the `dgemm` call. In `m_kern_min` we still make temporary copies of B and C because both of them are stored in a sparse data structure. The copies could be vectorized to improve performance but we found the benefits are not great due to the typically small values of the dimensions `nd1` and `nd3`. No temporary copy is needed for the result A. Some code duplication was found in `m_kern_min` and `m_kern_max` in the index calculations and they have been refactored into a separate subroutine `precompute_indices`. The impact of the optimisations is summarised in Table 4. Note that these are purely optimisations on the serial performance of the kernel. Profiling suggested that the multi-threaded performance is improved more than the 20 % that we see in serial performance, this could be due to an improvement in the load balance.

	total	dgemm	m_kern_max	m_kern_min
before	505s	70s	105s	75s
after	401s	69s	42s	69s

Table 4. Time spent in serial execution of the matrix_multiply benchmark with 256 atoms before and after optimisations. The most significant optimisation is the reduction of memory copies in m_kern_max which reduces its time by 2.5x.

The main source of parallel inefficiency remaining in the matrix_multiply benchmark is Serial code outside parallel regions, rather than thread imbalance in the parallel region. The serial code is fairly evenly split between a large number of subroutines with no obvious bottlenecks standing out. It would require more RSE effort to thread the remaining serial parts of the code.

3.4 Multi-threading exact exchange calculation

Multi-threading was added to the exact exchange kernel in CONQUEST. This work was not included in the proposal, but was decided to be added during the project. The optimisations, described in more detail below, yielded good multi threaded performance on up to 8 threads, shown in Figure 3.

m_kern_exx_cri kernel: To improve this kernel's performance the serial performance was first improved by combining three individual instances of the same nested loop structure into one. An array deduction inside a triple nested loop was then replaced with a call to the blas function dot. This required refactoring some allocatable arrays into 1d buffers and using pointer arrays to reference them. Some additional unused allocatable arrays were also removed. Finally, OMP threading was added around two nested loops containing the bulk of the calculations. Collapsing these two loops with OMP lead to good scaling of the threaded region from 1 to 16 OMP threads.

m_kern_exx_eri kernel: This kernel matches `m_kern_exx_cri` closely. Therefore, the inner calculation, including the newly introduced blas call, was extracted into a separate subroutine and called from both kernels which helped reduce code duplication. OMP threading was then added in a similar place to the `m_kern_exx_cri` kernel with the exception of only one loop rather than two nested ones. This immediately showed good scaling from 1 to 8 OMP threads.

m_kern_exx_eri_gto kernel: Again, this kernel has many similarities with the previous. Therefore, the inner calculation (the part which differed from `m_kern_exx_eri`) was extracted into a separate subroutine and called from within the newly refactored `m_kern_exx_eri` kernel with an additional parameter `is_gto`. This allowed an immediate benefit from the OMP threading added in the previous step.

Threading exx_phi_on_grid The subroutine *exx_phi_on_grid* is called several times from all of the above kernels and showed up as a hotspot in our VTune profiles. Therefore, OMP threading was added to the nested loops over x, y and z within. This showed a good improvement to overall performance and scaling. Before this threading was implemented, some serial improvements were made such as removing unused calculations and unnecessary zeroing of variables.

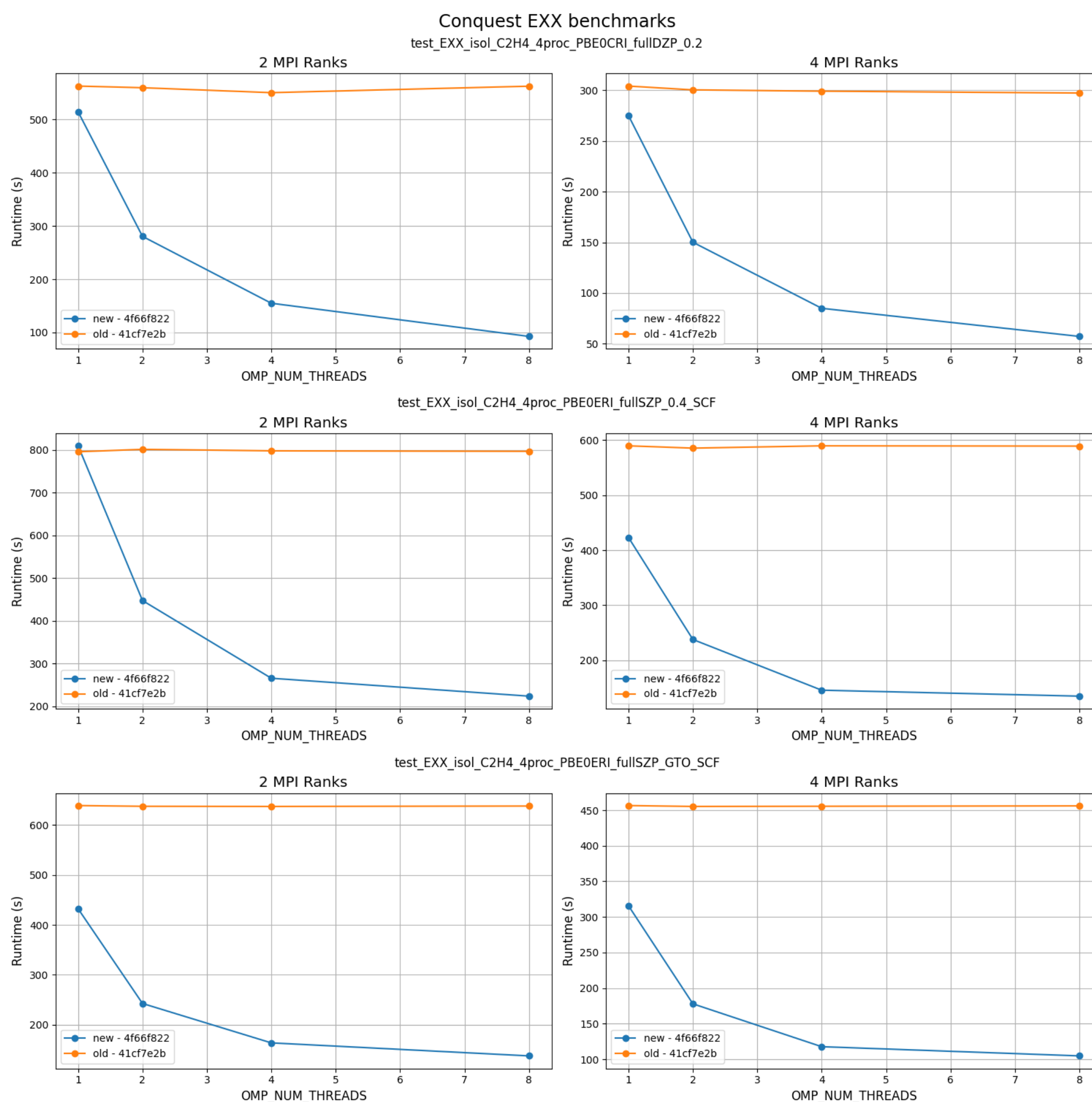

Figure 3. Results of running benchmarks of the three optimised EXX kernels using Reframe. The benchmark *test_EXX_isol_C2H4_4proc_PBE0CRI_fullDZP_0.2* demonstrates the performance of the *m_kern_exx_cri* kernel, whereas *test_EXX_isol_C2H4_4proc_PBE0ERI_fullSZP_0.4_SCF* and *test_EXX_isol_C2H4_4proc_PBE0CRI_fullSZP_GTO_SCF* demonstrate that of *m_kern_exx_eri* with *is_gto == .false.* and *m_kern_exx_eri* with *is_gto == .true.* respectively. Results can be seen for two code versions, before our changes (old - 41cf7e2b) and after (new - 4f66f822).

4 SUMMARY

In this project, we have improved the sustainability and multi-threaded performance of CONQUEST. The sustainability has been improved by adding automated tests, continuous integration, and automation of the build system. The performance improvements can be summarised in three parts. In grid-based operations, code has been refactored to reduce duplication, and OpenMP parallelisation has been added on loops over blocks. In Matrix multiplication, the serial multiply kernel has been optimised, the OpenMP parallel region has been moved to a higher level, and overlapping communication and computation has been investigated. In the exact exchange calculation, code has been refactored to merge loops over support functions, and the resulting fused loop has been parallelised with OpenMP.

5 ACKNOWLEDGMENTS

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk). We are grateful to the UK Materials and Molecular Modelling Hub for computational resources, which is partially funded by EPSRC (EP/T022213/1, EP/W032260/1 and EP/P020194/1).

REFERENCES