ARCHER2 eCSE07-02: Improving the Performance of Linear-Scaling BigDFT for ARCHER2 and Beyond

Arno Proeme (EPCC, University of Edinburgh), Laura E. Ratcliff (School of Chemistry, University of Bristol)

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk).

Abstract

This report details efforts to address BigDFT stability issues and performance on ARCHER2, initially linked to one-sided MPI communication failures. Various bugs and stability issues in BigDFT's use of one-sided MPI were identified and corrected (including improper window object handling and zero-sized window issues), leading to significantly improved stability on ARCHER2 and enabling larger calculations. Attempts to prototype a two-sided MPI alternative based on both point-to-point and collective communications were made but proved challenging to implement without wider modifications to the existing code structure. Systematic parallel scaling analysis showed hybrid MPI+OpenMP execution is more performant than pure MPI, especially at scale. Scaling results detailed in this report provide a useful reference enabling users of BigDFT on ARCHER2 to better exploit resources efficiently. Profiling suggested that MPI communication (one-sided operations, collectives like MPI_Allreduce, and load imbalance) remains the primary performance bottleneck limiting overall scalability despite effective OpenMP parallelization in key routines. While stability was enhanced, optimizing MPI communication is crucial for future performance improvements.

Objectives 1 & 2: Replacing One-sided Communications

Determination of suitable test case for development

Prior to project start BigDFT exhibited a variety of application failure scenarios on ARCHER2, including segmentation faults and apparent deadlock ("hanging" behaviour) occurring non-deterministically. Prior investigation had suggested the underlying cause for this to be issues with how BigDFT uses one-sided MPI and/or issues with support for one-sided MPI in MPI libraries or other parts of the underlying software stack that implement support for the same. On ARCHER2 for example, switching MPI transport layer from the default of OFI to UCX or setting MPICH_SMP_SINGLE_COPY_MODE - the choice of which mechanism to use when sending large messages between ranks on the same node in a way that leverages their ability to access a single shared memory - from Cray MPICH's default of XPMEM to CMA (which may perform less well) showed some reduction in the incidence of application failure. These workarounds however had limited success, and failures continued to occur at unpredictable points during application execution.

In order to be able to determine the success in resolving these issues of a two-sided MPI implementation or of any possible fixes made to the one-sided implementation, initial project work therefore focused on identifying a test case that on ARCHER2 failed not only deterministically but also demonstrably as a result specifically of issues associated with BigDFT's use of one-sided MPI. This test would ideally fail in the communication of the potential across ranks, as this is an important step and the code includes optional correctness checks of the implementation of this functionality during initialisation, i.e. occurring entirely predictably and early on during application execution, which is ideal for testing and development. A suitable test case was found by compiling BigDFT with `-O0` optimisation level and running the linear scaling 2CzPN_1mols benchmark with initialisation checks turned on (see https://github.com/aproeme/bigdft-benchmarks) with the default (OFI) transport layer and without using CMA as a single-copy workaround. This test could be run on just 2 ranks with 1 thread in order to further simplify debugging during development.

Exploration of failure sensitivity of one-sided MPI

The DDT debugger was used in the process of determining a suitable test case in order to confirm whether a given segmentation fault arises from one-sided MPI and if so to try to shed light on and potentially even fix the failure sensitivity of BigDFT's usage of one-sided MPI. Figure 1 shows that when BigDFT is run with two ranks that share the same node, a call to mpi_get within the subroutine start_onesided_communication (see Figure 2) resolves to a call to _cray_mpi_memcpy_rome (provided in /opt/cray/pe/lib64/libmpi_gnu_91.so.12) to copy the data in question within shared memory. The segmentation fault that arises during this operation is reported by DDT as a read or write taking place before the start of a memory allocation.

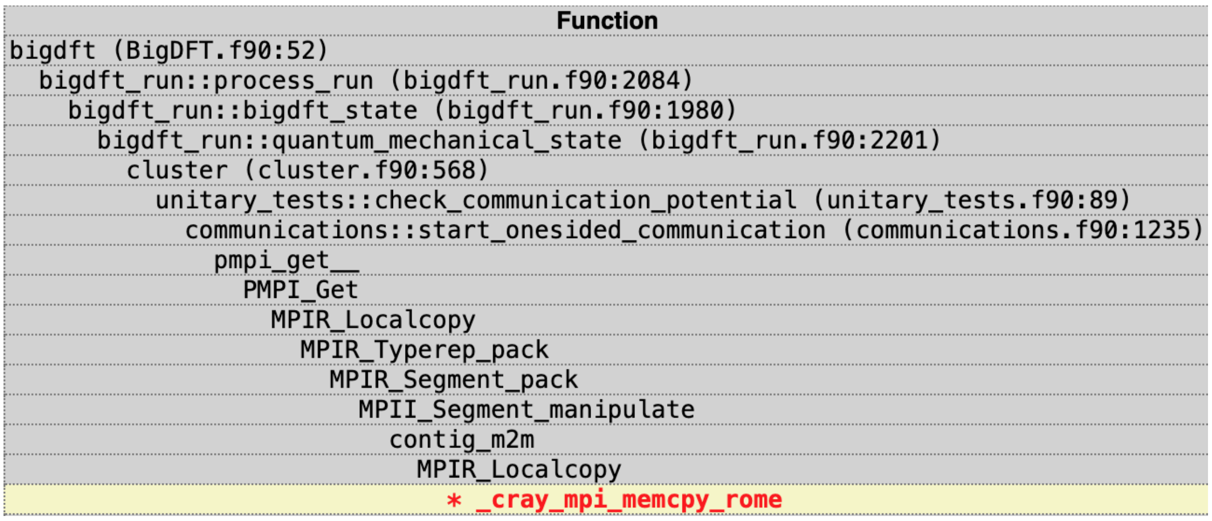
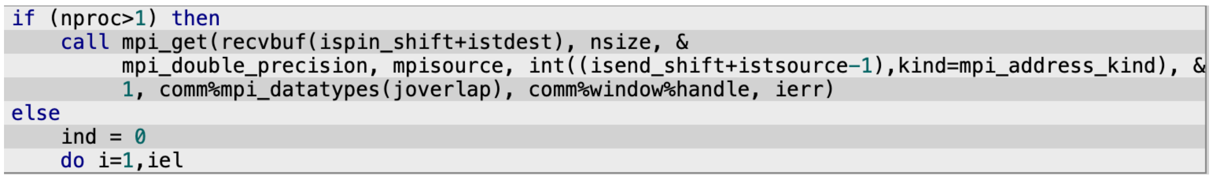

```
                              Function
bigdft (BigDFT.f90:52)
  bigdft_run::process_run (bigdft_run.f90:2084)
    bigdft_run::bigdft_state (bigdft_run.f90:1980)
      bigdft_run::quantum_mechanical_state (bigdft_run.f90:2201)
        cluster (cluster.f90:568)
          unitary_tests::check_communication_potential (unitary_tests.f90:89)
            communications::start_onesided_communication (communications.f90:1235)
              pmpi_get__
                PMPI_Get
                  MPIR_Localcopy
                    MPIR_Typerep_pack
                      MPIR_Segment_pack
                        MPII_Segment_manipulate
                          contig_m2m
                            MPIR_Localcopy
                              * _cray_mpi_memcpy_rome
```

Figure 1: DDT stack trace for deterministically failing test (2CzPN_1mols with `-O0` compiled BigDFT) showing segmentation fault resulting from call to mpi_get (see Figure 2 for code).



```
if (nproc>1) then
    call mpi_get(recvbuf(ispin_shift+istdest), nsize, &
         mpi_double_precision, mpisource, int((isend_shift+istsource-1),kind=mpi_address_kind), &
         1, comm%mpi_datatypes(joverlap), comm%window%handle, ierr)
else
    ind = 0
    do i=1,iel
```

Figure 2: code context for call to MPI_Get in subroutine start_onesided_communication responsible for segmentation fault shown in Figure 1.

Setting the Cray MPICH environment variable MPICH_OPTIMIZED_MEMCPY to 0 instructs mpi_get to use the system's memcpy (memcpy_ssse3) provided by glibc instead of Cray's version optimised for the AMD EPYC Rome processor (cray_mpi_memcpy_rome), however this suffered from the same memory read/write error and segmentation fault.

Fixes to improve stability of one-sided MPI communication

Although the one-sided MPI communication problems in LS-BigDFT were far more severe on ARCHER2 than on other supercomputers, following the start of the project, some seemingly related issues were observed on other clusters. Therefore, in parallel with implementing the prototype two-sided MPI communication approach, the one-sided communication implementation was investigated in further detail to search for possible explanations. This revealed the existence of a bug in the usage of an mpi window. BigDFT uses the futile library to wrap a number of low-level Fortran operations, including MPI calls. However, the communication of the density in LS-BigDFT (in bigdft/src/modules/rhopotential.f90) mistakenly used the native mpi_get call, instead of the f_mpi_get_wrapper, while passing the wrapped window object (collcom_sr%window, of type fmpi_win) instead of the window handle (collcom_sr%window%handle). This mismatch between wrapped and native MPI objects/calls was not picked up by compilers as the address of the structure collcom_sr%window coincided with the address of the handle (collcom_sr%window%handle). This bug led to undefined behaviour on ARCHER2, and has now been fixed, with the code now calling the futile fmpi_get routine rather than the native mpi_get call.

In addition to the above bug, a number of other small changes were made to MPI communications which led to improved stability. First, calls to mpi_alltoallv were behind the scenes being replaced with multiple calls to mpi_get, which had been shown to be quicker than mpi_alltoallv in tests several years ago. This was being activated by default, whatever the size of the data to be communicated, but more recent experiences seem to suggest this approach is less stable. This approach (activated according to VARIABLE_ONE_SIDED_GET_ALGO in futile/wrappers/mpi/alltoall.f90) is now only activated if an environment variable has been set. Second, the window creation for one-sided communications in LS-BigDFT was in some cases creating a zero-sized window. While in principle fine according to the MPI standard, this again appeared to cause unstable behaviour, and so the futile library has been updated to be more explicit about how to handle such cases. Finally, calls to mpi_allreduce for communicating small scalars, have been modified to use mpi_iallreduce, for example for calculating the trace of the Hamiltonian matrix in LS-BigDFT (in bigdft/src/modules/get_basis.f90). For large communications, there is a fallback to all_reduce. This removal of blocking communications has led to improved load balancing, which again appears to have improved stability.

Individually, none of these changes were sufficient to fix the stability problems seen with LS-BigDFT on ARCHER2, but collectively have led to significant improvements, enabling calculations to be performed for a range of system sizes (recent tests have gone up to around 5000 atoms) and node counts, which was not possible before the start of this project. All of these changes have been implemented in the latest release of BigDFT, and are therefore available to all users.

Prototype two-sided communication

The primary target for exploring the use of two-sided as an alternative to one-sided MPI is in the communication of the potential in the subroutine start_onesided_communication defined in /src/modules/communications.f90. The goal was to do so with minimal changes to pre-existing code structure and logic, hence development focused on making additions in situ with this in mind rather than designing and implementing a communication pattern from scratch.

In the one-sided implementation each rank executes the core logic shown in simplified form in pseudocode Listing 1. MPI window creation and fence synchronisation are handled by calls to the futile library wrappers fmpi_* but otherwise native MPI calls are used. The comm object passed to the subroutine is of type p2pComms, defined in src/helpers/communications_base.f90, and contains parameter values - potentially different for each rank - used in communication. The argument comm(joverlap) where it appears in an MPI call in Listing 1 indicates that one or more arguments for the call are based on values extracted from the comm held by each rank that vary depending on the loop index joverlap.

```
subroutine start_onesided_communication(comm, sendbuf, recvbuf, other_args)

  type(p2pComms) comm

  do ispin = 1,comm%nspin
    if ispin == 1 then
      call fmpi_win_create(comm, sendbuf)
      call fmpi_win_fence(comm)

    do joverlap = 1, comm%noverlaps
      if ispin == 1 then
        call mpi_type_create_hvector(comm(joverlap))
        call mpi_type_commit(comm(joverlap))

      if rank == comm%mpidest(joverlap) then
        call mpi_type_size(comm(joverlap))
        call mpi_type_get_extent(comm(joverlap))
        call mpi_get(recvbuf, comm(joverlap))

    end do
  end do
```

Listing 1: pseudocode abstraction of core one-sided communication logic in subroutine start_onesided_communications in /src/modules/communications.f90.

The observed one-sided communication pattern is as follows. For a given ispin value each rank obtains the data it needs by iterating over values of joverlap. The joverlap loop has as many iterations as there are ranks in the global communicator, i.e. comm%noverlaps is equal to the total number of ranks. The condition rank==comm%mpidest(joverlap), which checks whether the executing rank is the destination of data and hence should issue a call to mpi_get, is satisfied for every iteration. The target rank specified as part of the mpi_get call, i.e. the rank that serves as the source of the data to get - not shown explicitly in Listing 1 but identified as mpisource in the source code - is equal to joverlap-1. In other words, on each rank the joverlap loop iterates over all ranks, getting data from one particular rank (mpisource) in turn - including from itself - during successive iterations. By the end of the joverlap loop each rank has obtained data (or at least issued an mpi_get call to do so) from each other rank (as well as from itself), effectively implementing an All-to-All communication pattern.

Although the observed communication pattern naturally suggests the use of one or more collective two-sided communications, an approach that adhered closer to the existing code structure and logic is to implement a substituted send/receive messaging pair as a direct alternative to each mpi_get. Retaining the existing code structure means that point-to-point message pairs should be non-blocking (mpi_isend/mpi_irecv) to avoid deadlock. The main parameters needed as arguments by mpi_irecv map directly onto a subset of the mpi_get parameters and their values passed as arguments in the existing mpi_get call on the same rank, as shown in Table 1.

mpi_irecv parameter	**relevant mpi_get parameter**	**BigDFT value to pass to mpi_irecv (based on mpi_get argument on same rank)**
`buffer`	`origin_address`	`recvbuf(ispin_shift+istdest)`
`count`	`origin_count`	`nsize`
`datatype`	`origin_datatype`	`mpi_double_precision`
`source`	`target_rank`	`mpisource`

Table 1: determination of BigDFT arguments to pass to mpi_irecv based on arguments to mpi_get on same rank

Table 2 shows where the mpi_irecv arguments listed in Table 1 originate, highlighting how they depend on comm and joverlap.

BigDFT variable used to form mpi_irecv argument	**value**
`recvbuf`	passed to subroutine
`ispin_shift`	`= (ispin-1)*comm%nrecvbuf`
`istdest`	`= comm%comarr(4,joverlap)`
`nsize`	set by call to `mpi_type_size(comm(joverlap))`
`mpisource`	`= comm%comarr(1,joverlap)`

Table 2: origin of argument values for mpi_irecv, showing their dependence on comm and joverlap

When it comes to mpi_isend, some of the parameters needed as arguments also relate to mpi_get parameters as shown in Table 3. However unlike for mpi_irecv *it is the values of these mpi_get parameters used as arguments on the rank posting the matching mpi_irecv call that should be passed to a given mpi_isend*.

mpi_isend parameter	**relevant mpi_get parameter**	**BigDFT value for mpi_isend (based on mpi_get argument on matching mpi_irecv rank)**

`buffer`	`target_displacement`	`sendbuf(target_displacement)`
		`= sendbuf(int(isend_shift+istsource-1))`

`count`	`target_count`	`1`
`datatype`	`target_datatype`	`comm%mpi_datatypes(joverlap)`

Table 3: determination of BigDFT arguments to pass to mpi_isend based on arguments to mpi_get on rank posting the matching mpi_irecv call

Table 4 shows where the mpi_isend arguments listed in Table 3 originate, highlighting how they depend on comm and joverlap.

BigDFT variable used to form mpi_isend argument	**value**
`sendbuf`	passed to subroutine
`isend_shift`	`= (ispin-1)*npotarr(mpisource)`
`npotarr`	passed to subroutine
`mpisource`	`= comm%comarr(1,joverlap)`

Table 4: origin of argument values for mpi_send, showing their dependence on comm and joverlap

As indicated, in order to formulate a correct mpi_isend the sending rank needs to know the values of a number of additional parameters that in the pre-existing code are known only by the receiving rank - communications metadata. As each rank will ultimately need to send to every other rank to replicate the result of the one-sided communication pattern, each rank needs to know what the values of these required parameters are on all ranks. An mpi_gather call for each required parameter is a natural way to provide this. In principle this could be implemented following the somewhat clumsy logic shown in Listing 2.

```
subroutine start_onesided_communication(comm, sendbuf, recvbuf, other_args)

  type(p2pComms) comm

  do ispin = 1,comm%nspin
    if ispin == 1 then
      call fmpi_win_create(comm, sendbuf)
      call fmpi_win_fence(comm)

! Run joverlap loop once to gather comms values required to form mpi_isends
    do joverlap = 1, comm%noverlaps
      if ispin == 1 then
        call mpi_type_create_hvector(comm(joverlap))
        call mpi_type_commit(comm(joverlap))

      if rank == comm%mpidest(joverlap) then
        call mpi_type_size(comm(joverlap))
        call mpi_type_get_extent(comm(joverlap))
        if “onesided” then
          call mpi_get(recvbuf, comm(joverlap))
        else if “twosided” then
          call mpi_gather to gather required comms values from all ranks

    end do

! Run joverlap loop again to perform two-sided communications
    do joverlap = 1, comm%noverlaps
      if ispin == 1 then
        call mpi_type_create_hvector(comm(joverlap))
        call mpi_type_commit(comm(joverlap))

      if rank == comm%mpidest(joverlap) then
        call mpi_type_size(comm(joverlap))
        call mpi_type_get_extent(comm(joverlap))
        if “onesided” then
          call mpi_get(recvbuf, comm(joverlap))
        else if “twosided” then
          call mpi_irecv(from mpisource)
          do j = 1, comm%noverlaps
            mpidest = j-1
            call mpi_isend(to mpidest)                ! send to all ranks
          end do

    end do
    if "twosided" then
      call mpi_wait on both mpi_irecv and mpi_isend requests

  end do
```

Listing 2: pseudocode abstraction of alternative two-sided communication logic in subroutine start_onesided_communications in /src/modules/communications.f90.

A more elegant solution is to use collectives to distribute not only the required communications metadata but also the potential data itself, as this would preclude the need to run the joverlap loop twice and could in fact be used to avoid the joverlap loop altogether.

For all variations of attempted approaches - point-to-point versus collectives - however, code complexity including indirection associated with the apparent generality of the existing one-sided implementation proved to be a confounding factor in attempting to formulate correctly specified matching mpi_isend calls and, by extension, collectives with similar effect. This is reflected by the use of the comm object held by each rank and the inability - all else being equal - to make any simplifying assumptions about the rank-specificity of any communication metadata embedded within it, as well as the complexities associated with specification of a custom MPI hvector on each rank for each joverlap value.

As an example of the rapidly expanding scope of communications metadata that would need to be distributed prior to successful two-sided communication of potential data in this approach and difficulties associated with this, consider the identification of the send buffer for mpi_isend listed in Table 3. This identification was motivated by the facts that the base address of the MPI window created by each rank is sendbuf(1) and that target_displacement in mpi_get specifies the offset position within that buffer for a given target rank. Hence the send buffer for the mpi_isend originating from that same rank should be sendbuf(target_displacement), as listed in Table 3. However closer consideration shows that the datatype for the target buffer (source of data) in mpi_get is comm%mpi_datatypes(joverlap), which consists of an mpi_hvector type specific to both the get-posting rank and the particular value of joverlap. In addition, mpi_get specifies two different data types and counts, namely count nsize for datatype mpi_double_precision on the origin (receiving) side, versus count 1 of the abovementioned custom MPI hvector on the target (sending) side. Attempts to implement and debug communication of prerequisite values associated with the MPI hvector datatype such as its size ran into subtleties regarding the difference between mpi_type_get_extent and mpi_type_get_true_extent.

Attempts to develop a fully functioning two-sided prototype were not successful. However development prototypes exploring implementation of two-sided MPI communication of the potential can be found at https://github.com/aproeme/bigdft-suite (branch "twosided").

A better approach for implementing two-sided communication than pursued in this project is likely to be to forgo the attempt at tightly integrated in-situ addition of two-sided communication as an alternative within the start_onesided_communication subroutine and based on minimal changes to the code therein. Instead, a more productive approach may be to revisit at a higher level the definition of the p2pComms type object and simplify both its structure by eliminating replication of communication metadata where possible and how it is used to implement one-sided communication. This might make a close integration of two-sided MPI into the then one-sided implementation more feasible. Alternatively, the two-sided approach could then be developed separately and more naturally without being tied as closely to the one-sided implementation.

Objective 3: Improve OpenMP Performance

A systematic parallel scaling analysis of overall application runtime was performed on ARCHER2 for a number of benchmarks of varying system sizes (number of atoms) spanning the linear scaling (LS) and cubic scaling algorithms that are of primary interest, including isolation of the initialisation stage that dominates fragment calculations for the former and periodic boundary conditions for the latter. Although calculations for larger systems are also run in production, the number of atoms in these benchmarks were chosen to allow investigation of scaling trends and bottlenecks using convenient amounts of computational resources, and with an understanding of the underlying algorithm informing us that conclusions about the limits on performance at this scale should generalise to the limits on performance of larger systems on larger node counts.

The main goal of this analysis was as a first step to provide a context to better execute and draw conclusions from targeted profiling - performed using CrayPAT - aimed at identifying OpenMP threading efficiency and potential improvements thereon. In addition, the scaling results themselves should serve to inform better utilisation of ARCHER2 by users of the code.

In all cases parallel efficiency was computed relative to single node performance for pure MPI - 128 ranks and 1 thread per rank - even though this typically does not reflect the smallest possible scale at which a given problem could be run as this is the most relevant reference point on ARCHER2 given enforced exclusive node allocation. Although absolute efficiencies will vary when computed with reference to smaller scale calculations, conclusions regarding the underlying trends in performance remain valid. Results for three identical independent runs were averaged for every configuration explored, and variability determined to be well below 1% of the mean in many cases and never above 5%. Benchmarks used are available at https://github.com/aproeme/bigdft-benchmarks. For the sake of reproducibility and potential further analysis results of all experiments is available at https://github.com/aproeme/bigdft-benchmarking-profiling/ and scripts used to submit jobs, extract timings and compute statistics can be found at
https://github.com/aproeme/benchmarking (latest commit at time of writing - 8117de8).

What follows below is a summary of observed trends in performance as well as insights from profiling into contributing factors, including the effects of OpenMP threading and MPI parallelism, and observations on the efficiencies thereof and potential improvements identified.

Linear scaling

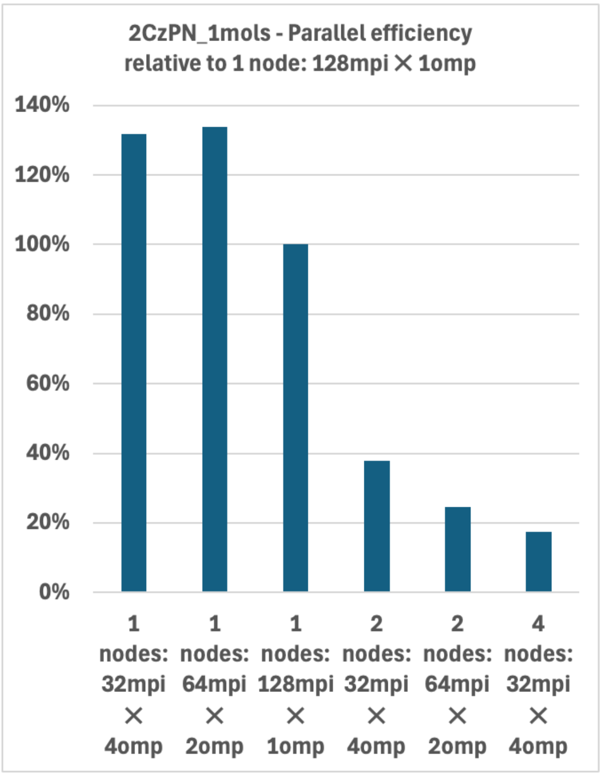


Figure 3: Parallel efficiency scaling of overall application walltime for **2CzPN_1mols** benchmark with linear scaling (LS) algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 83s.

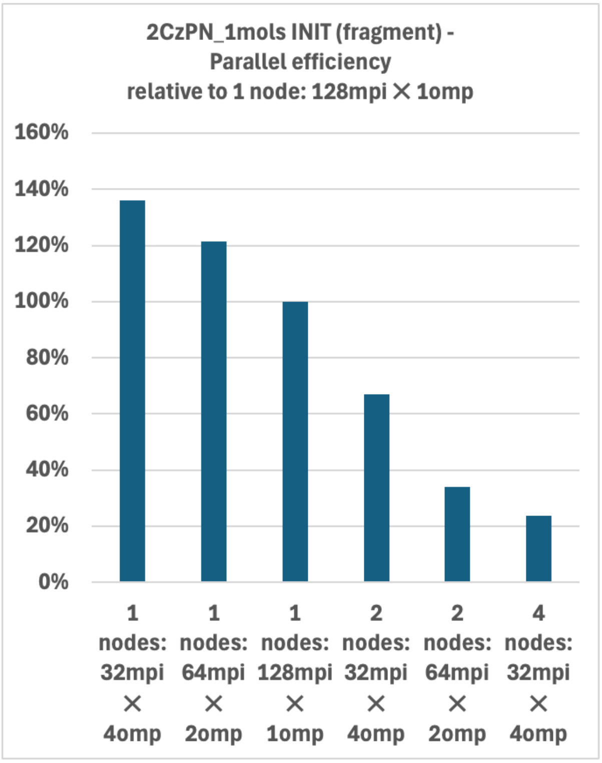


Figure 4: Parallel efficiency scaling of initialisation (INIT) walltime for **2CzPN_1mols** benchmark for linear scaling (LS) algorithm, representative of performance for fragment calculations. Walltime for 1 node 128mpi × 1omp reference configuration: 7.5s.

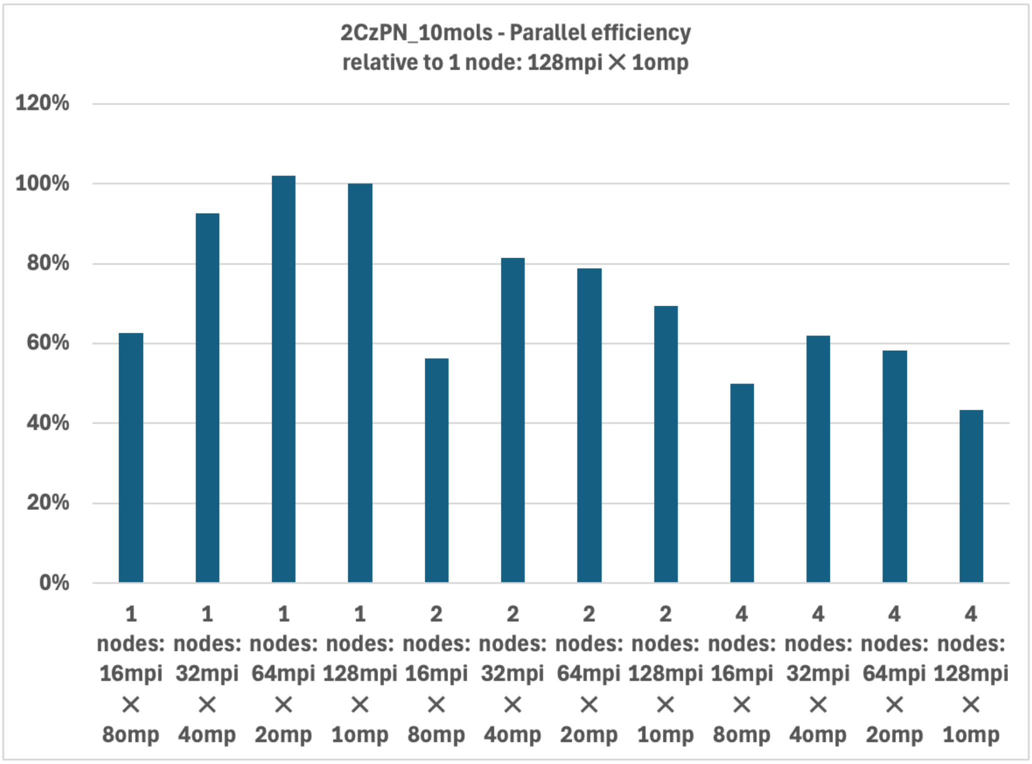


Figure 5: Parallel efficiency scaling of overall application walltime for **2CzPN_10mols** benchmark with linear scaling (LS) algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 1560s.

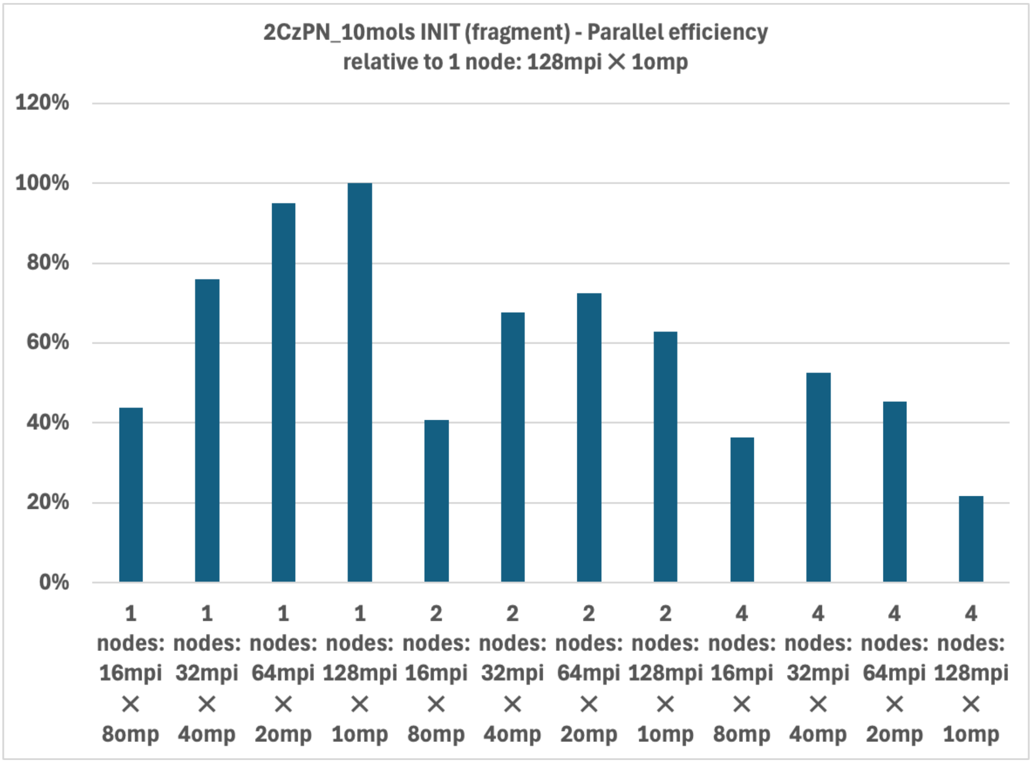


Figure 6: Parallel efficiency scaling of initialisation (INIT) walltime for **2CzPN_10mols** benchmark for linear scaling (LS) algorithm, representative of performance for fragment calculations. Walltime for 1 node 128mpi × 1omp reference configuration: 88.3s.

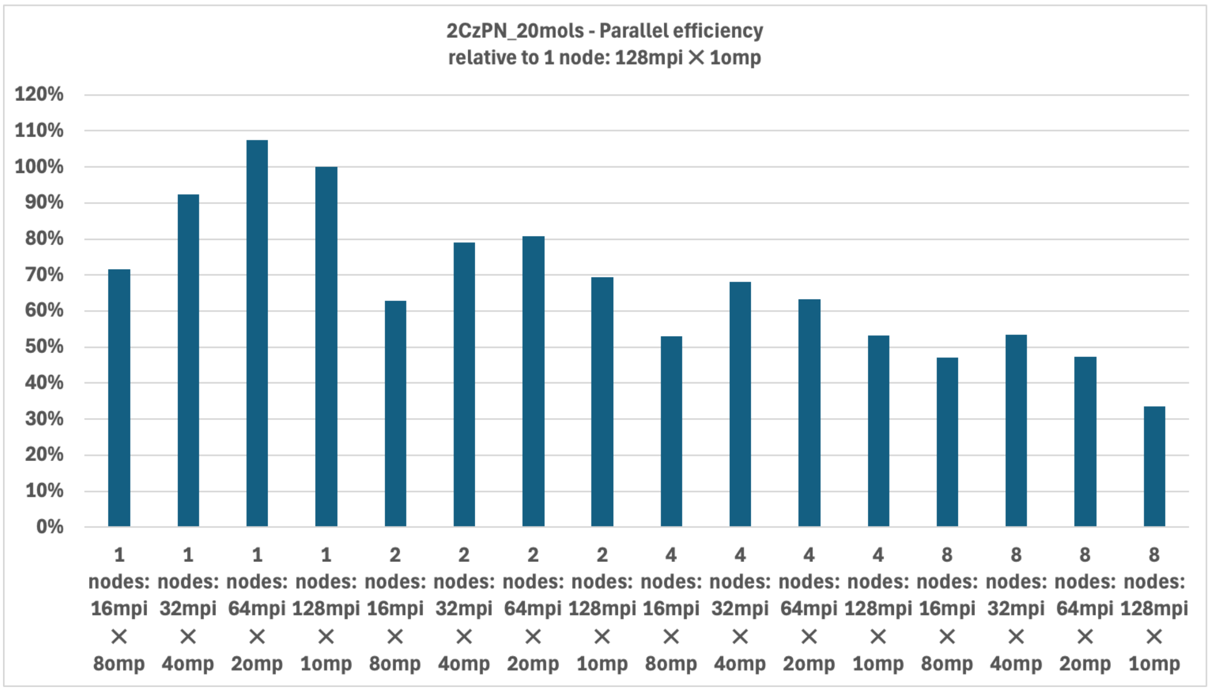


Figure 7: Parallel efficiency scaling of overall application walltime for **2CzPN_20mols** benchmark with linear scaling (LS) algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 3810s.

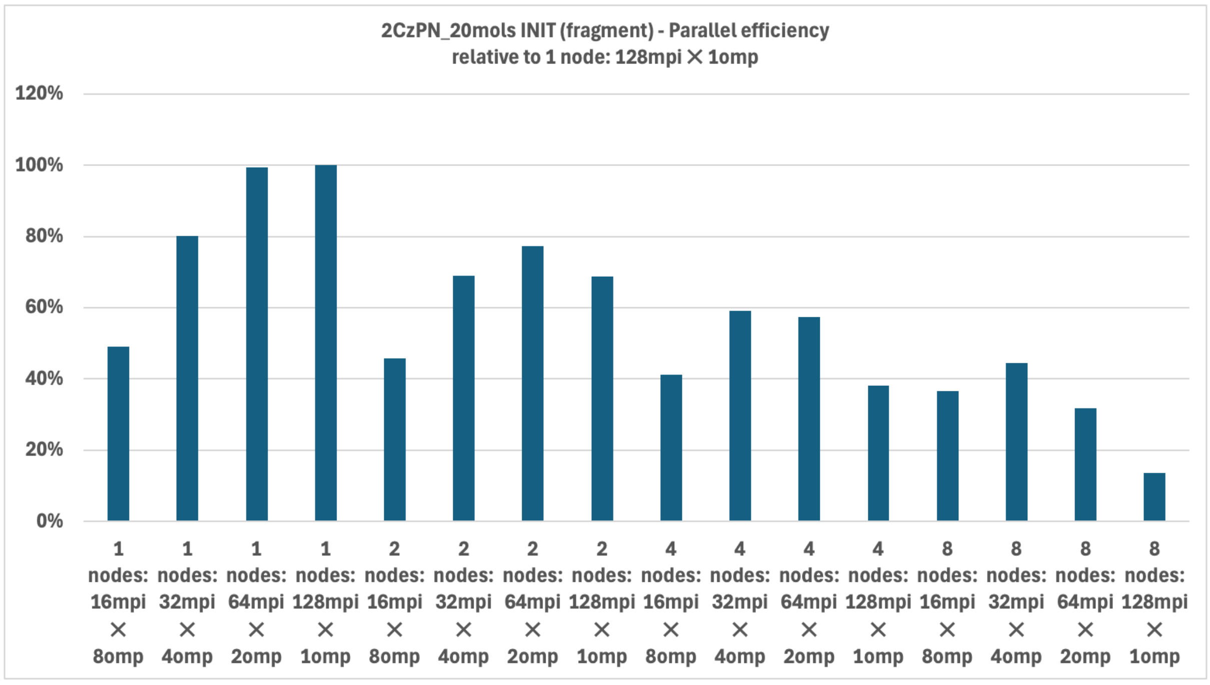


Figure 8: Parallel efficiency scaling of initialisation (INIT) walltime for **2CzPN_20mols** benchmark for linear scaling (LS) algorithm, representative of performance for fragment calculations. Walltime for 1 node 128mpi × 1omp reference configuration: 239s.

Globally speaking increasing system size - e.g. from 2CzPN_1mols (54 atoms) to 2CzPN_10mols (540 atoms) to 2CzPN_20mols (1080 atoms) - extends the scale at which reasonable parallel efficiency can be maintained or, equivalently, shows greater parallel efficiency at any given corecount, not surprising as the total amount of computational work per rank and thread grows relative to parallelisation overheads.

It is clear that pure MPI execution - 128 ranks per node × 1 thread per rank - is almost never the most performant use of a given number of nodes (cores) and that using multiple threads per rank is essential for obtaining best performance on any given number of nodes. Additionally, in attempting to retain parallel efficiency when scaling out to run on more nodes the optimal number of threads per rank increases for larger node counts. In other words, a strategy of deciding on an optimally performing hybrid execution configuration based on single-node experiments will not yield optimal performance at larger scale.

The parallel scaling behaviour of fragment calculations, which are approximated in our analysis by focusing purely on the initialisation (INIT) time of the same linear scaling benchmark runs, follows a roughly similar trend.

Sampling-based profiling was performed using CrayPAT to gain insight into why this is, i.e. into how the net observed performance trends result from the interplay of MPI and OpenMP parallelisation, and whether - and if so, how - the use of OpenMP in BigDFT could be improved to further support efficient parallel scaling. It should be noted that due to memory constraints faced when profiling it was only possible to profile runs with 1 or 2 threads per rank. Nonetheless this yielded the following key insights into global scaling behaviour:

- In single-node runs the percentage of time spent in MPI calls is significant (20%-45%) but still generally less than time spent within BigDFT itself ("USER" time in profiling terms), and smaller when fewer ranks and more threads per rank are used. MPI_ALLREDUCE is by far the most prominent MPI call, with MPI_BARRIER also significant and MPI_WAIT and MPI_WIN_CREATE also appearing but less significantly.
- As the number of nodes and ranks increases, the percentage of time spent in MPI routines grows substantially and becomes the dominant contributor to execution time, i.e. exceeding the time spent in BigDFT proper. At larger scales, MPI_GET and MPI_WIN_FENCE often become the dominant MPI operations, sometimes exhibiting high imbalance. MPI_ALLREDUCE also remains significant.

In other words, globally speaking overall performance is primarily limited by MPI communication, particularly one-sided operations and collectives, as the scale (number of nodes and ranks) increases. Load imbalance across ranks and nodes also contributes significantly to the bottlenecks at larger scales. The reason why multithreading is able to alleviate this bottleneck somewhat at a given node count is because it can reduce the total number of ranks and associated communication overheads without introducing equal or greater combined overheads and inefficiencies through OpenMP-parallelisation. Suggestions to run with reordered rank placement as suggested by CrayPAT for a number of runs might help improve work balance across ranks. Optimizing the use of MPI collective and one-sided communications should help (as might replacing one-sided communications altogether).

We investigated whether the existing OpenMP implementation in BigDFT could be improved and indeed whether OpenMP might be added where not yet present in order to better alleviate the MPI bottleneck. Of the time spent in BigDFT itself the subroutine sparsemm_new (defined in the sparsematrix module, part of the CheSS library) was clearly dominant across benchmarks and node counts, followed by sumrho_for_TMBs (defined in the rhopotential module). The subroutine sequential_acces_matrix_fast2 (also in the sparsematrix module) also appeared, but less significantly. These subroutines are all already parallelised using OpenMP. Of these, sparsemm_new and sequential_acces_matrix_fast2 sometimes show significant imbalance *between ranks*. However imbalance percentages *between threads* in an OpenMP team *within a rank* for the main OpenMP-parallelised subroutines are often low. Hence this likely reflects imbalance across MPI ranks, possibly as a result of unequal work distribution, rather than poor OpenMP parallelization within a rank. Functions related to OpenMP runtime overheads like gomp_team_barrier_wait_end or GOMP_parallel appear occasionally, indicating some overhead, though usually small.

Further work could be done to verify the distribution of work for OpenMP parallel regions across ranks and the degree of imbalance between threads within a rank, especially for larger numbers of threads per rank as this could not be done using CrayPAT as a result of previously mentioned memory constraints. If intrateam thread imbalance were to be found to grow significantly for higher threadcounts it would be worth investigating scheduling as starting point.

Cubic scaling

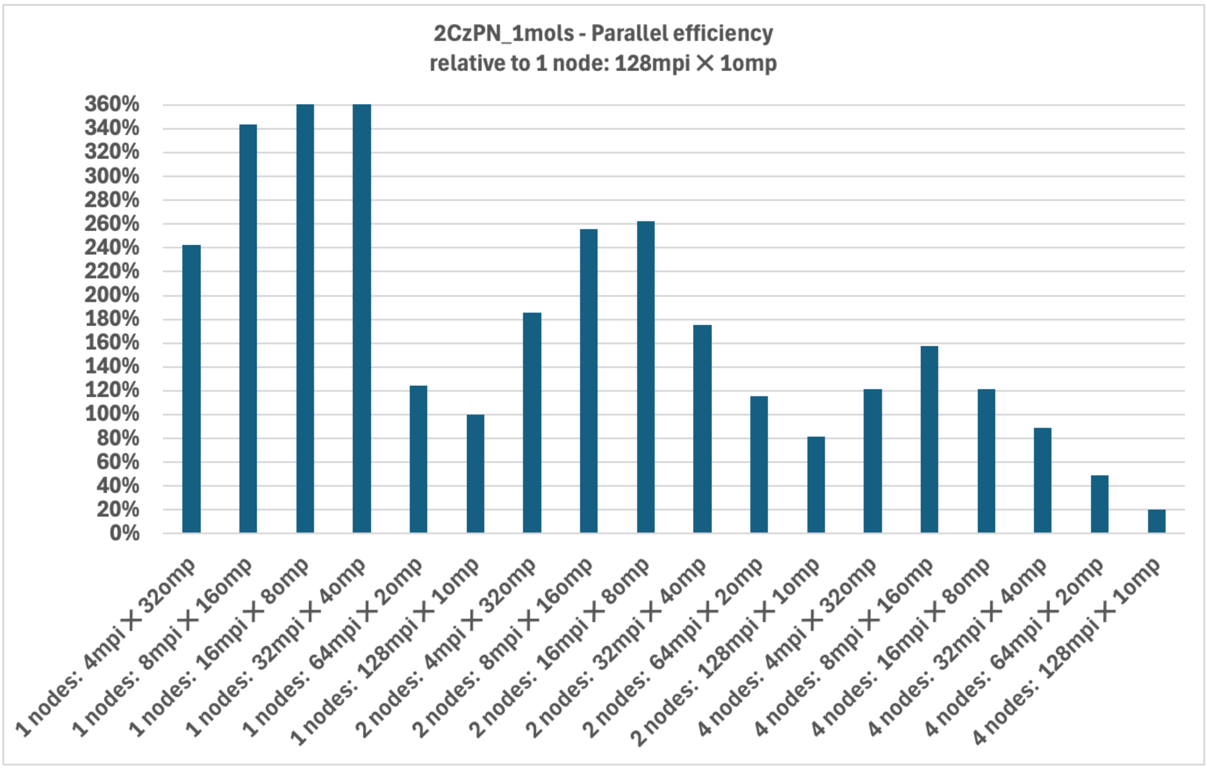


Figure 9: Parallel efficiency scaling of overall application walltime for **2CzPN_1mols** benchmark with cubic scaling algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 63s.

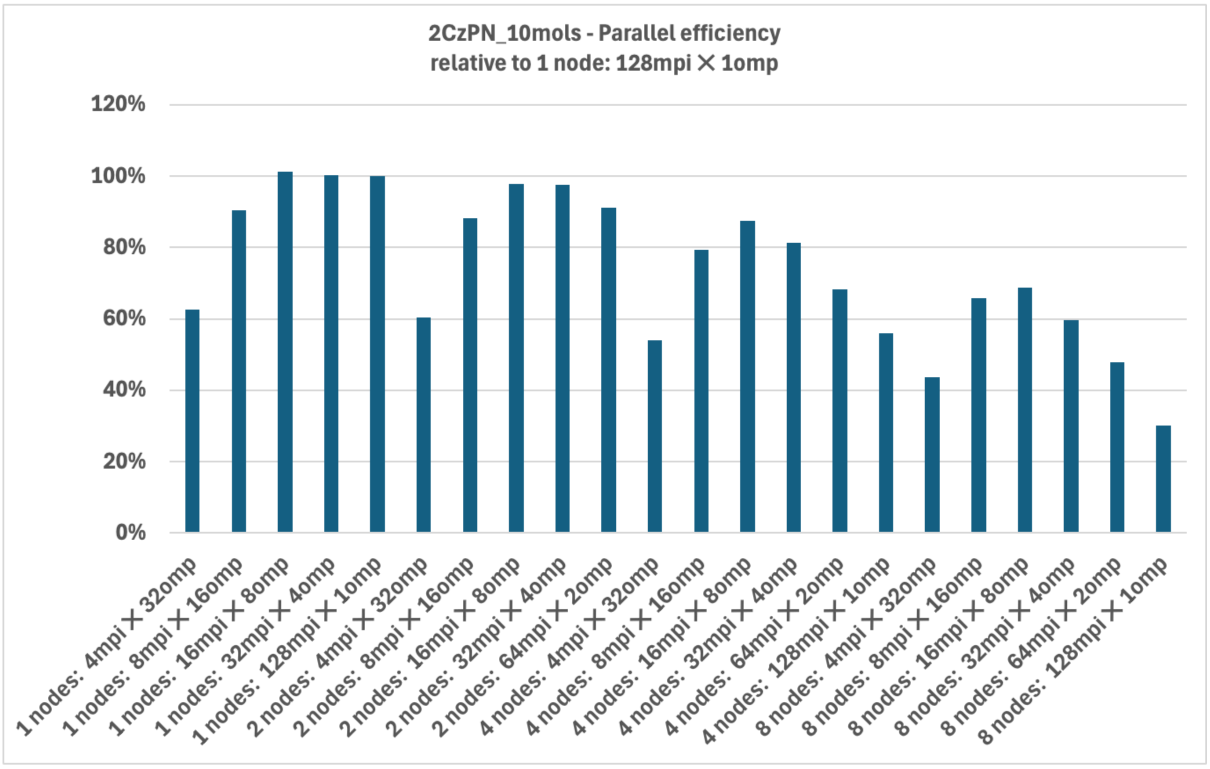


Figure 10: Parallel efficiency scaling of overall application walltime for **2CzPN_10mols** benchmark with cubic scaling algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 933.5s.

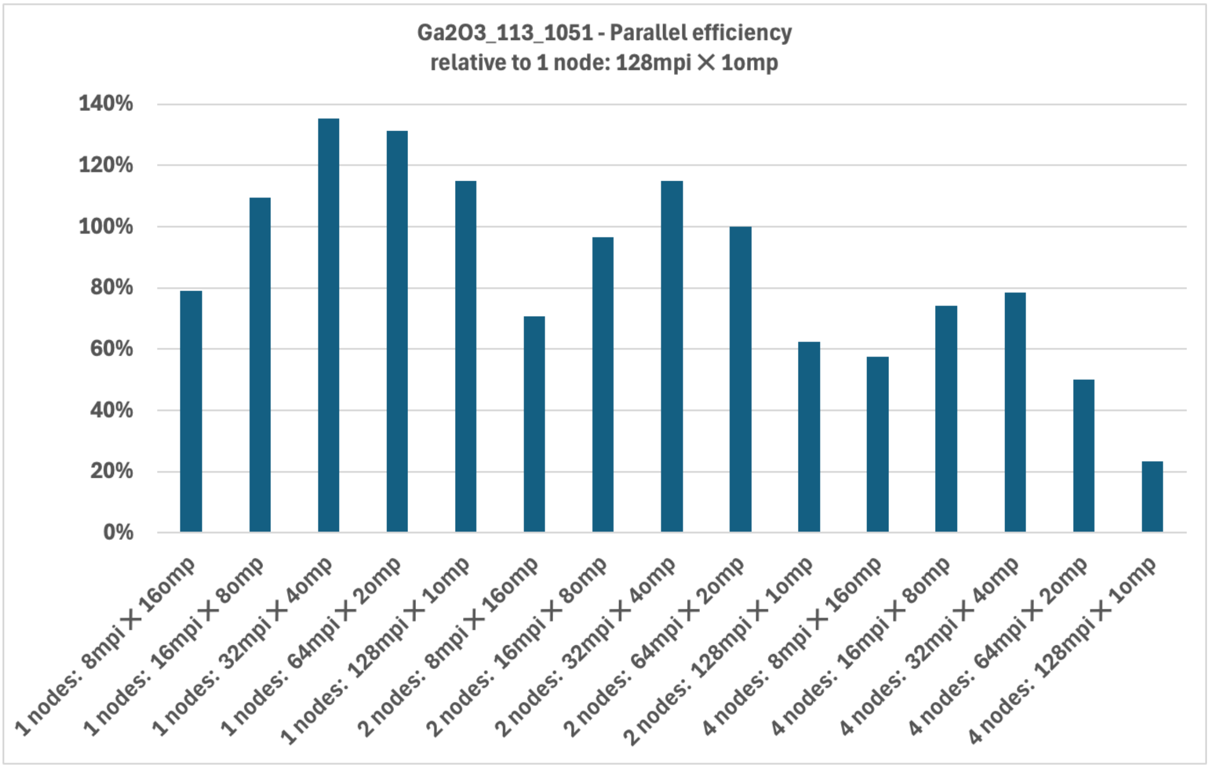


Figure 11: Parallel efficiency scaling of overall application walltime for **Ga2O3_113_1051** benchmark (periodic boundary conditions) with cubic scaling algorithm. Walltime for 1 node 128mpi × 1omp reference configuration: 80s.

Scaling trends for the cubic scaling algorithm were found to be globally similar to those for the linear scaling algorithm, including for the Ga2O3_113_1051 benchmark (160 atoms), which uses periodic boundary conditions, but with benchmarks typically benefiting from a larger number of threads per rank for optimal performance compared to the linear scaling algorithm. For a relatively small system such as the 2CzPN_1mols benchmark (54 atoms) this effect is dramatic, with an optimal thread count of 8 or 16 threads per rank giving between 3× and 8× times faster performance than pure MPI execution on the same number of cores.

Profiling showed that as for the linear scaling algorithm total time spent in MPI is significant and grows to equal or exceed time spent in BigDFT code as the total number of ranks increases to and beyond the limit of good parallel efficiency, albeit somewhat less drastically than for the linear scaling algorithm. Again MPI_ALLREDUCE is consistently a top time consumer, showing increasing cost and imbalance as the total number of ranks increases. Other calls like MPI_GET, MPI_WIN_FENCE, and MPI_WAIT also become significant at the largest scale, exhibiting high imbalance. As for linear scaling, addressing these imbalances, e.g. by using custom rank ordering as suggested by CrayPAT for some multi-node runs, might help improve scalability. Optimizing the use of MPI collective and one-sided communications should help (as might replacing one-sided communications altogether).

With regards to BigDFT itself, for 2CzPN_* benchmarks the subroutines convolkinetic and convolkinetict dominated consistently, and comb_rot_grow_* and comb_rot_shrink_* (for lower rank counts) subroutine variants also appeared as secondary top contributors. For the Ga2O3_113_1051 benchmark with periodic boundary conditions the dominant subroutine is convolut_kinetic_per_sdc, with syn_rot_per, convrot_n_per, acc_from_tensprod, and apply_w appearing as secondary top contributors. These subroutines, many of which are part of BigDFT suite's Orbital Manipulations library, are all already OpenMP parallel. For all of these, the general observed trend is that imbalance across ranks grows with increasing rank count, but with usually relatively low imbalance between threads in a team on a given rank. As for linear scaling this likely reflects imbalance across MPI ranks as a result of unequal work distribution rather than poor OpenMP parallelization within a rank. Further work could be done to verify the distribution of work for OpenMP parallel regions across ranks and the degree of imbalance between threads within a rank for larger numbers of threads per rank. If intrateam thread imbalance were to be found to grow significantly for higher thread counts it would be worth investigating scheduling as a starting point.

One difference compared to profiling results for the linear scaling algorithm is that there are not insignificant contributions from calls to BLAS (between 1% and 7% of total time), namely mkl_blas_def_dgemm_kernel_zen, mkl_blas_def_xdgemv, mkl_blas_avx512_strmm_sm, and from the LAPACK subroutine DAXPY (2% - 5%).