ARCHER2-eCSE01-11 Technical Report

Multi-Layered MPI parallelisation for the R-matrix with time-dependence code

M Plummer[1], AC Brown[2], HW van der Hart[2] and GSJ Armstrong[2,3]

[1]*Theroetical and Computational Physics Group, Scientific Computing Department, STFC Daresbury Laboratory, Sci-Tech Daresbury, Cheshire WA4 4AD, United Kingdom*
[2]*Centre for Theoretical Atomic. Molecular and Optical Physics, School of Mathematics and Physics, Queen's University Belfast, Belfast B17 1NN, United Kingdom*
[3]*Now at: Quantemol Ltd, 320 Angel, 320 City Road, London, EC1V 2NZ, United Kingdom*

Email contacts: Andrew.Brown@qub.ac.uk, h.vanderhart@qub.ac.uk (RMT theory and applications), Martin.Plummer@stfc.ac.uk (technical queries about the project)

Abstract

The R-matrix with Time Dependence code package RMT is a numerical solver for the time-dependent Schrödinger equation. As such it provides a means for describing the time-domain behaviour of atomic and molecular systems driven by external fields. RMT is at the forefront of research in attosecond/ultrafast/strong-field physics wherein it is the only code capable of describing general systems driven by arbitrarily polarised laser pulses, with a full account of the multi-electron correlation. RMT employs the R-matrix division-of-space principle. The physical space occupied by the electronic wavefunction is divided into two distinct regions. In the inner region, centred on the nuclear centre of mass, multielectron effects are described in full. In the outer region, an ionised electron is sufficiently isolated from the residual ion that electron exchange with the residual electrons can be neglected. In the small inner region, a basis set expansion is used to give an accurate description of the multielectron correlation, while in the large outer region the one-electron wavefunction is described on a finite-difference grid. This eCSE project has added new levels to the separate existing MPI parallelization schemes in the two regions. The inner region parallelization was restricted to at least 1 task per component symmetry: it now allows several symmetries per task, so that the number of inner region tasks can be much smaller than before in appropriate calculations. The outer region parallelization allowed for 1 task per radial sector of grid points. It now allows for several tasks per sector, dividing up the wavefunction component 'channels' among them. An existing option for OpenMP parallelization over channels handled by a task is retained. Performance improvements on ARCHER2 are illustrated, and a systematic approach for choosing the correct balance of inner- and outer-region tasks for large and costly calculations is given.

Keywords: ARCHER2, eCSE, RMT code, attosecond physics, R-matrix, time-dependent Schrödinger equation, multi-layered MPI parallelization, parallelization enhancement, optimum HPC performance, synchronization of distinct parallelization schemes

(1) Introduction

The field of attosecond physics (1 attosecond = 10^{-18} s) has unlocked new realms of investigation, beyond what were previously thought to be impenetrable limits: recent developments in laser technology have, for the first time, made it possible to observe the motion of electrons in real time, as well as control this motion with high precision [1–5]. Attosecond physics is a natural successor to femtochemistry [6], and several of its overarching aims remain in chemical and even biological systems. Processes as fundamental as vision and photosynthesis are driven by light-mediated charge transfer in large molecules [7], while photovoltaic cells harness photoabsorption in semiconductors [8]. A key, long-term goal of ultrafast physics must be to understand these processes in more than qualitative terms. To reach that goal we must first properly understand light-matter interaction at the atomic level.

The rapid advance of laser technology has facilitated myriad new measurement techniques able to resolve time-dependent electronic dynamics in complex systems. Our community is investigating these phenomena using the world-leading R-matrix with time-dependence code (RMT) [9]. RMT solves the time-dependent Schrödinger equation for general multielectron atoms, ions and molecules interacting with laser light. As such it can be used to model ionization (single-photon. multiphoton and strong-field), recollision (high-harmonic generation, strong-field rescattering, [10]) and, more generally, absorption or scattering processes with a full account of the multielectron correlation effects in a time-dependent manner. Calculations can be performed for targets interacting with ultrashort, intense laser pulses of arbitrary polarization. Calculations for atoms can optionally include the Breit-Pauli correction terms for the description of relativistic (in particular, spin-orbit) effects [11].

This eCSE project introduces new parallelism at the basic level to RMT to enable new leading-edge calculations by our community, and to greatly improve the efficiency of ongoing current application calculations. The recent functional extensions to RMT can increase the size of calculations by orders of magnitude, and the already complex existing parallel structure of the code must be enhanced to deliver current scientific goals efficiently and pave the way for more ambitious projects.

(1.1) Overall Plan of Work

The RMT code and its applications are described in detail by Brown et al (Computer Physics Communications **250** (2020) 107062, reference 11, hereafter called RMT-CPC). We will assume familiarity with (or access to) this paper, which the current report supplements, in particular the listings of parameters and routines in RMT-CPC sections 3 and 9 respectively. The RMT repository [12] is hosted on GitLab, with an expanding number of continuous integration features, a test suite, and strict rules for descriptions of code updates. The code is supported by detailed comment-based documentation using doxygen,

The first phase of the project was to introduce a new level of parallelization into the RMT ‘inner region’. The RMT code divides the problem into two distinct regions: an inner region (IR), in which all electrons are close to the nucleus, and an outer region (OR), in which one electron (two in the case of a current extension to treat double ionization) has moved away from the nucleus. In the small IR a basis set expansion is used to give an accurate description of the multielectron correlation, while in the large OR the one-electron wavefunction is described on a finite-difference (FD) grid. In the RMT-CPC implementation a separate efficient parallel scheme is employed in each region, but there are two fundamental limits on how well these schemes scale which we are seeking to eradicate in this project, as described in sections 2 and 3 respectively. During the eCSE project, the inner region work was carried out first, followed by the introduction a new level of parallelization into the outer region in the second phase of the project. The code also contains comprehensive new doxygen commenting, which can be used as a more detailed description of localized code changes.

The rest of this report concentrates on describing the code changes introduced in the project and on performance tests, both from the repository test suite and of a size more typical for ambitious calculations. The report is intended to be a hands-on guide for RMT users, to enable them to make the most of the new coding and parallelization schemes to perform computationally efficient simulations. There is no further description of the physics modelled by RMT (summarized in CPC-RMT and its references) or the underlying R-matrix theory [9].

The coding was tested using both the Cray and gnu compilers on ARCHER2. Timing results are for the Cray compiler using –O3 and the default OFI MPI libraries (tests with UCX libraries did not show any particular significant improvement). The main results were obtained before the change in default CPU frequency from 2.25 GHz to 2.0 GHz. Subsequent testing showed that Cray performance was degraded using 2.0 GHz and the --cpu-freq=2250000 flag was used to obtain 'double-checking' results. An energy use comparison would be a useful follow-up project.

(2) Inner Region Parallelisation

In the IR, the problem requires repeated multiplication of the large Hamiltonian matrix with the wavefunction vector. The field-free Hamiltonian is of block-diagonal form, with each block corresponding to a symmetry of the system, and the laser field introduces field-dependent (dipole) interaction blocks which mix the symmetries. **The original implementation requires at least one MPI task for each symmetry block** (called LML blocks after quantum-mechanical angular momentum quantum numbers L and M_L). For some calculations— with a *small number* of *very large* blocks— we employ multiple MPI tasks per block. However, for calculations of atoms driven by long-wavelength, arbitrarily polarised light, we have the inverse problem: an enormous number (>10,000) of reasonably small blocks. Given that if for these calculations, the bottleneck is the OR, the IR cores (or 'pes': processing elements) can actually spend the bulk of the execution time sitting idle: e.g. up to 50% in a recent test, amounting to a wastage of 60,000 core hours *per calculation*. This is an extreme example, but it indicates the pressing need for this development. **Thus, the first goal is to refactor the inner-region parallel scheme to allow the assignment of more than one symmetry block to a single MPI task.** This is illustrated in figure 1 for linearly polarized light (with symmetry blocks labelled 'H' and dipole interaction blocks labelled D). Figures 1-3 of RMT-CPC show examples of more complex Hamiltonian structures (e.g. for non-linear polarised light).

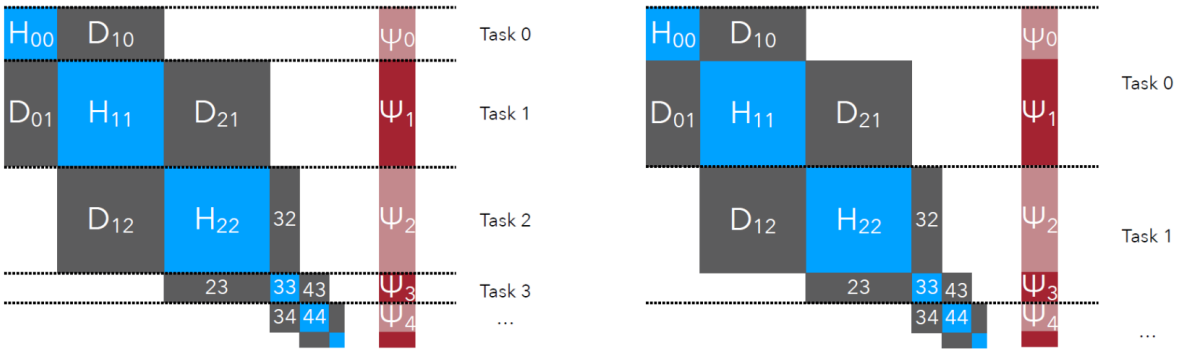

Figure 1: Schematic illustrating the inner region MPI task division of wavefunction symmetry components in the original scheme (with1 task per block) and new scheme.

The main kernel of the code's operation requires components of the wavefunction vector for a particular symmetry to be shared via MPI with every other symmetry to which it is coupled. Each core (symmetry) holds a list of symmetry indices to which it is coupled, and for each index in the list it posts a non-blocking send or receive as appropriate. Each core then waits for all of its receives to complete before proceeding to operate on the received data.

Lb_m_comm and **lb_comm**

The inner region MPI tasks are divided into communicators **lb_comm** (handling a particular LML block) with each communicator's 'master' forming a communicator **lb_m_comm**. The original scheme required each LML block to be in a distinct **lb_comm**. The new parallelization scheme allows for multiple blocks per MPI task. While this does not involve introducing new communicators, we now have a regime choice in which **lb_comm** has (default) size 1 and several LML blocks can be attached to each member of **lb_m_comm**. New arrays and shared data have been introduced within the **lb_m_comm** communicator to allow the use of multiple LML blocks in **lb_m_comm**. These are built upon the existing structure.

A parameter **my_num_LML_blocks** is generated in module **mpi_layer_lblocks**, in a new routine **setup_Lblock_communicator_size_pn** which is called from the existing routine s**etup_Lblock_communicator_size** if **no_of_inner_region_pes** < **no_of_LML_blocks**. The user does not have to input anything to choose the distribution of LML blocks between tasks. The actual choice of the new distribution is made using an algorithm set out in the routine.

- For linear (z-direction) field polarization in the atomic case, the algorithm aims for as even as possible load balancing of numerical work on each task during the main propagation.
- For non-linear (or xy-plane) polarization and the molecular case (and other more complicated cases), the algorithm also aims for an even distribution of communications between tasks during the propagation.

A parameter **num_av** is defined in the latter case as **NINT(REAL(no_of_LML_blocks) / REAL(no_of_inner_region_pes))** which sets a limit (in practice, **num_av** + 1) on the number of LML blocks that may be included as local to a task. In the former linear polarized case this limit is not used. The distribution in both cases uses a summed function of the number of elements in the LML blocks to model the associated workload (further details are included in the code/doxygen comments), though restricted by the limit in the latter case. This is because of the much more complicated block-to-block potential coupling in the latter cases (figures 1-3 and accompanying text in RMT-CPC). Initial performance testing of the new parallelization structure showed this approach is needed, with the communications providing the bottleneck in the latter case.

The routine provides diagnostic output from the distribution algorithm and includes limits (which can be adjusted or overridden) on deviations from load-balancing, halting the run for communicator sizes and input number of tasks in the inner region deemed inappropriate for a particular calculation. If the first pass through the algorithm does not assign all LML blocks to tasks, or all tasks to blocks, the procedure is repeated with the load-balancing function adjusted. The model average workload is formed from a total model workload divided by a parameter **number_to_divide_by** originally set to **no_of_inner_region_pes** and subsequently decreased or increased by 1 respectively on each pass, with checks to avoid oscillations which exit the loop with an assignment of remaining blocks/tasks). In case of an exit outside the internal acceptable limits, the program stops and the user has the choice to change the number of MPI tasks or override the limits via a check on **number_to_divide_by**.

```
if (inner_region_rank == 0) print *, 'number_to_divide_by set to:', number_to_divide_by
!  if (number_to_divide_by > (no_of_inner_region_pes + (no_of_inner_region_pes / 5))) then
! number_to_divide_by criterion relaxed here to allow more flexible inner region pe transformations:
```

```
! this means that more preliminary tests can (should) be carried out.
    if (number_to_divide_by > (no_of_inner_region_pes + (no_of_inner_region_pes / 1))) then
< run stops>
```

and similarly

```
if (inner_region_rank == 0) print *, 'number_to_divide_by set to:', number_to_divide_by
if (number_to_divide_by < (no_of_inner_region_pes - (no_of_inner_region_pes / 3))) &
    call assert (.false., 'number_to_divide_by is too small, think again')  !  .false. : run stops
cycle
```

The **setup_Lblock_communicator_size_pn** (and **setup_Lblock_communicator_size**) routines are self-contained, so that the choice of distribution of LML blocks among tasks (and vice versa) can now be decided using the methods in the routines or alternatively by other methods without disturbing the main communication structures. One method being worked on following the eCSE project is to allow a combination of many-task-per-LML-block and many-LML-blocks-per-task using a Huffman tree approach (see, for example, https://en.wikipedia.org/wiki/Huffman_coding and https://www.programiz.com/dsa/huffman-coding) for full automation of the distribution choice: this will be reported separately.

Once the distribution is decided, the information for efficient propagation of the main wavefunction calculation is contained in existing parameters and arrays, with new meaning, and some additional arrays. For example, **my_LML_block_id** now refers to the first LML block handled by a task in **lb_m_comm** and is used in conjunction with **my_num_LML_blocks**. Arrays **my_LML_blocks(:)**, **master_for_LML_blocks(:)**, **LML_blocks_per_Lb_master(:)** contain additional relevant information and are used in modules **distribute_hd_blocks**, **distribute_wv_data**, **live_communications** and **inner_to_outer_interface,** and are set in routines **setup_colors_for_Lblock_comms** and **setup_colors_for_Lblock_comms_p** (a new routine) in **mpi_layer_lbocks**. Several other modules have minor modifications (variables becoming arrays) to allow for the fact that each **lb_m_comm** task handles several LML blocks. Details of the particular block holding the initial ground state and its (row, column) position in the now multi-block local storage arrays are used in **inner_to_outer_interface** and **wavefunction**.

To fit the existing parallelization, splitting the LML block over **lb_comm** tasks, without disruption of the main propagation code, two new parameters, **numrows_sum** and **numrows_blocks**, are introduced in module **distribute_hd_blocks2** to supplement and replace externally the parameter **numrows** and the use **of max_L_block_size**.

- With several LML blocks per task, **numrows_sum** is the sum of the numbers of rows of these LML blocks, rather than the number of rows of the partitioned LML block in the many-task **Lb_comm** case (the parameters **rowbeg** and **rowend** are similarly modified in the first case).
- **numrows_blocks** is defined as **max_L_block_size** in the original case and is equal to **numrows_sum** (which can be greater than **max_L_block_size**) in the new case.

The formulation is needed for arrays which the **lb_comm** communicator assumes have a fixed dimension **max_L_block_size** in MPI data exchanges.

The main routines (modules in brackets) that required more substantial modification (other modules have minor alterations for consistency of parameters) are:
get_psi_at_inner_fd_pts and **get_outer_initial_state_at_b** (**inner_to_outer_interface**), part of **initialize_wavefunction** (**wavefunction**),

scatter_dblocks_within_Lblock and **get_num_rows_for_local_dblocks**
(**distribute_hd_blocks2**),
send{recv}_surf_amps_to_lb_masters,
setup_and_distribute_dblocks (**distribute_hd_blocks),**
and particularly
scatter_ib_surfamps (**distribute_wv_data**)
parallel_matrix_vector_multiply and called routines (**live_communications**)

The changes to **parallel_matrix_vector_multiply** are overlaid onto the existing parallelism in which relevant parts of off-task wavefunction arrays are received from, and local parts are sent to, other **lb_m_comm** tasks. The received wavefunction components are matrix-multiplied by connecting dipole matrices and added to the local parts of the wavefunctions arrays. The **MPI_irecv**s and the corresponding **MPI_isend**s of local data to connected tasks are performed first, an **MPI_WaitAll** allows the data to be put in place, then the arrays are multiplied by the field coefficients and the matrix multiplications can take place. The modified routines now allow for a loop over local LML blocks in the gather/send section, preceded by checks to see whether the connected blocks are local or off-local and whether MPI is needed. Multiplication by the field coefficients is now part of the second loop over local LML blocks after the data has been transferred to ensure the transferred data are correctly in place, as the **MPI_WaitAll** must now be held back until the first loop over local LML blocks is complete. The transferred data arrays have an extra outer dimension over local LML blocks to avoid overwriting. This new code restructuring has been thoroughly tested and is rigorous.

Following the experience of the outer region new parallelization, there is scope for possible further reduction in the MPI calls in the case of more intricately linked Hamiltonian blocks (for molecules and elliptical polarization). Further checks can be introduced at start-up to see if a received non-local wavefunction LML component array section is required for several different local LML block interactions (‘up’, ‘down’ or ‘same’ categories), along with the option of bulk transfers of several wavefunction components in a single pair of MPI calls. The start-up checks would be equivalent to those performed in **test_outer_hamiltonian** (module **outer_hamiltonian**) but related to inner-region couplings. This refinement work (more complicated than the similar outer region set-up) is at an investigatory stage and will be reported separately. It will involve introducing expanded storage arrays for transferred data, which could be large, and the first loop with the **MPI_isend** and **MPI_irecv** commands would be performed once rather than up to three times depending on the complexity of the coupling, and would involve fewer calls with more data transferred in each call.

The changes to **scatter_ib_surfamps** involve modifications to an MPI subarray creation which allows the field-free input surface amplitudes (L-block dependent) to be copied and distributed among the RMT calculation LML blocks. An extra dimension was added to the subarray and an extra loop over local LML blocks was needed with careful targeting of the correct L block for each LML block**.**

get_psi_at_inner_fd_pts transforms inner-region matrix rows to outer region channels and requires new arrays of rows and channels per LML block and an additional loop inserted into the transformation loops.
get_outer_initial_state_at_b and its calling routine **initialize_wavefunction** choose the correct ground state block (and rows within it) from the correct task, and then share with the

first outer region (using new outer code as described below for outer region channel parallelization).
send{recv}_surf_amps_to_lb_masters require small 3-component arrays of details of each particular local LML block's position to be sent/received in advance of the main information. **scatter_dblocks_within_Lblock** requires book-keeping and correct distribution of rows of combined LML blocks, while **get_num_rows_for_local_dblocks** sets up the combined distribution of rows and **numrows_sum/numrows_blocks** parameters.

We may note that the main program **tdse** and initialisation test program **GetProcInfo** have a minor modification with the integer parameter **my_post** now an array of size **my_post_dim_lim** (set to 100 in **initial_conditions** and checked against the actual dimension needed in **initialise**).

(2.1) Inner region serial optimization

The introduction of the new parallelization and ongoing testing highlighted a noticeable inefficiency in the existing serial code for some of the test cases. The main compute time is taken up with either matrix-vector multiplications (BLAS **zgemv**), or matrix-matrix multiplications (BLAS **zgemm**) for multiple solutions, **numsols**, calculated simultaneously. The latter option is not always used or needed in application runs (and is a relatively new feature). However, it is possible for complex number arithmetic to rewrite the matrix-vector multiplication as a matrix-matrix operation for real numbers. This significantly reduces the number of floating point operations in RMT for particular (the majority of) multiplications for which the matrix (the dipole interaction matrices **loc_dblock_<u,d,s>** and the boundary surface amplitude matrix, now called **re_surf_amps**) is purely real or imaginary. It is also often more efficient in general as the hard-coded implementation of **zgemm**/**dgemm** can be more efficient than that for **zgemv**/**dgemv** (thanks to caching of intermediate data [13]). There is also a significant memory-saving advantage and, for smaller calculations, an additional caching advantage, as the copying of dipole arrays into complex arrays is no longer needed. The penalty is the copying of complex vector arrays into real arrays (real parts followed by imaginary parts) with a stride of the size of the array. The routines are in module **live_communications** (in particular **parallel_matrix_vector_multiply**, which now uses **dgemm** exclusively rather than a choice of **zgemv** and **zgemm**) with new array definitions backtraced to **distribute_hd_blocks** and **distribute_hd_blocks2** where the dipole and surface blocks are allocated. A similar change was made in module **outer_to_inner_interface** in which the real static inner region surface amplitudes are combined with the complex dynamic wavefunction derivatives: the derivatives are reordered in a temporary real array and **dgemm** is used, whereas previously the amplitudes were put into complex arrays and **zgemv**/**zgemm** was used.

(2.2) Sample test results

The RMT suite on gitlab includes a set of test cases for both atomic and molecular systems, with the atomic tests divided into 'small' and 'large' tests. The large tests vary in size but are not large on the scale of ARCHER2. These tests were mainly carried out to check the accuracy of the new scheme: they are not necessarily cases in which the new paradigm results in faster execution time, as work previously split between several tasks is now performed by one task. We first show results for a selection of four tests using various numbers of inner region tasks in the first column (the OR task number is held fixed for each test). As well as the test suite

samples, we also show IR timing results from a larger case for multiphoton ionization of Xe. Timings are either for the whole test simulation or a given number of iterations as indicated. Further details of timings from different parts of the inner region process are available from timing routines included in the package. The downside of combining several LML blocks on one MPI task is that the execution time on that task increases. However, in these tests, the superior caching (and reduced number of computations) in the new scheme results improves the execution time substantially for Ne+ and Ar_jK. The Ar_jK test with 83 IR tasks increases execution time with respect to 169 IR tasks, but the overall cost in CUs is lower. As will be shown in section 3, test Ar_LS is dominated by the OR time, as is test Ne for 120 IR tasks (with IR time dominating for ~40 IR tasks).

Ne+: 10 LML (L) blocks, 72 outer pes, full time in seconds (guide)

Inner region tasks	Execution time (seconds)	
	Original scheme	New Scheme
30	116	115
10	199	107
3	635	259

Ne: 71 LML (L) blocks, 72 outer pes, full time in seconds (guide)

Inner region tasks	Execution time (seconds)	
	Original scheme	New Scheme
120	102	101
41	171	166
40	-	180

Ar_LS: 196 LML blocks (405 channels), 164 outer pes, full time in seconds (guide)

Inner region tasks	Execution time (seconds)	
	Original scheme	New Scheme
196	941	941
97	944	944
64	969	965

Ar_jK: 169 LML blocks (1344 channels), 143 outer pes, time in seconds for 1460 timesteps (guide)

Inner region tasks	Execution time (seconds)	
	Original scheme	New Scheme
169	239	239
83	648	**326**
43	1179	651

The sample results for the Xe case concentrate on the inner region timings. In the table below, original results, a tenfold reduction in the number of inner region tasks results in a tenfold increase in inner region iteration time. The increase with the new sheme results is a factor of ~5, thanks to the more efficient and memory saving (thus allowing better cache use) new serial coding. The overall time for the iteration is dominated by the inner region, even with 1 LML block per task. ‘BOUND’ and ‘ARNO’ refer to boundary interactions (with the OR) and the IR enacting the Arnoldi update of the IR wavefunction respectively. This case is investigated more extensively in section 5.

Xe: 3619 LML blocks (16278 channels), 24*16(tasks)*8(threads) = 3072 outer pes, inner region time in seconds per iteration

IR tasks	Time (seconds)					
	Original Scheme			New Scheme		
	ITER	BOUND	ARNO	ITER	BOUND	ARNO
3712	0.21	0.15	0.05	0.163	0.08	0.02
320	2.0	1.5	0.35	0.88	0.69	0.13

We follow the tables with two figures showing the savings in CUs that the inner region parallelisation can provide. These are large test cases for multiphoton ionization of F- and a second Xe case (with a more complex description of the target atom). We show the results as CUs per iteration against varying numbers of inner region tasks. We will return to these cases following the outer region parallelisation. Note that the figures show CUs used per 100 time steps of the simulation. This determines the overall efficiency of the run after setup. The number of outer-region tasks is fixed for each system as (before the work of section 3) this fixes the size of the outer region grid. We may see that for F-, decreasing the number of inner region nodes from 20 to 3 improves performance by a factor of ~2.1, while for the more complex Xe case, decreasing the number of inner region nodes from 68 to 20 gives a performance improvement factor of ~1.6. A reduction in the number of nodes required gives a worthwhile performance benefit at the cost of a longer wall-clock time.

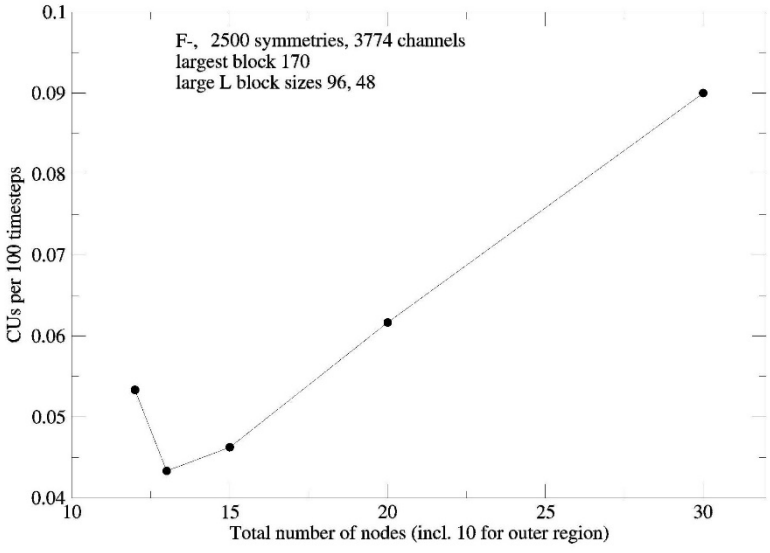

Figure 2: ARCHER2 computation cost for various IR node counts and 10 OR node counts, F-

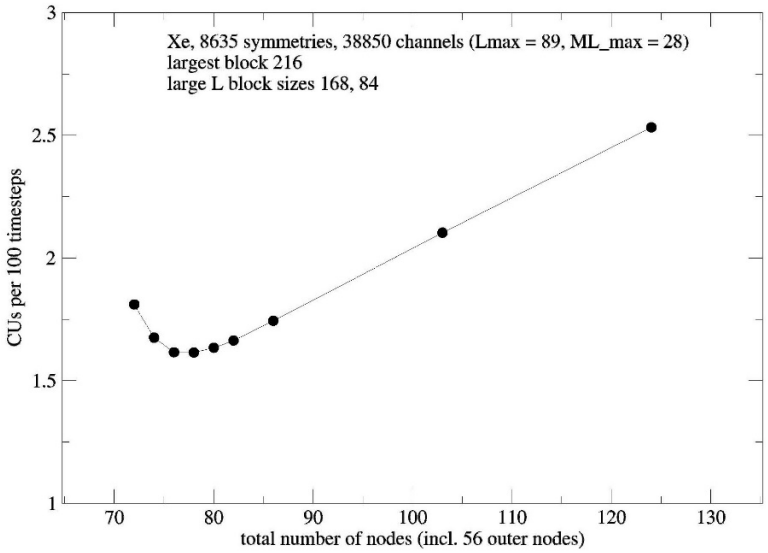

Figure 3: ARCHER2 computation cost for various IR node counts and 56 OR node counts, Xe (8635 LML blocks).

(3) Outer Region (OR) Parallelisation

In the OR the physical space is mapped onto a finite difference (FD) grid, and each MPI task handles a sub-region of grid points. The description of the electronic wavefunction involves a number of coupled electron emission channels, and most of the work in the OR involves loops over these channels. The RMT-CPC code includes OpenMP parallelisation over channels, but this is limited especially for the largest calculations where the number of channels can exceed 40,000. Thus, the second eCSE project goal is **to implement an additional layer of MPI parallelism in the outer region, so that each sub-region may be handled in parallel by multiple MPI tasks.**

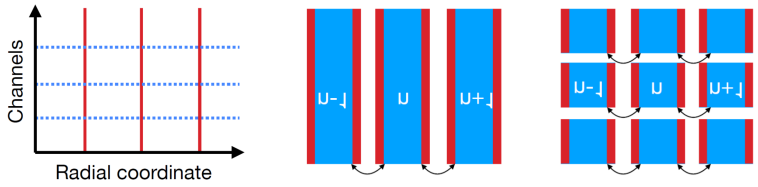

Figure 4: Schematic of the OR MPI division with 1 and 3 MPI tasks per sector.

Again, this new parallelisation affects a great range of modules and routines in RMT. The new parameter **number_of_pes_per_sector** (default 1) is set in input file input.conf, and checks are made on read-in that **no_of_outer_pes** has **number_of_pes_per_sector** as a factor. The size of the outer grid in sectors (referred to in the codes as outer blocks) is now given by **no_of_outer_pes** / **number_of_pes_per_sector**. The actual size of the outer region is worked out from the number of sectors, the number of points per sector and the grid spacing as described in RMT-CPC. The new communicators, which are set in **init_communications_module_part_two** (module **mpi_communications**), are **mpi_comm_block**, **mpi_comm_inter_block** and **mpi_comm_0_outer_block**. The first task in each **mpi_comm_block** (with **my_block_group_pe_id** = 0) acts as the block master (used

in **mpi_comm_inter_block**. **mpi_comm_0_outer_block** brings together the inner region master (the first IR MPI task) with all the tasks in the first outer block and is used for inner/outer region communication. When **number_of_pes_per_sector** = 1, **mpi_comm_block** has 1 member and **mpi_comm_inter_block** is the same as the OR **mpi_comm_region.**

mpi_comm_inter_block is defined for all tasks in **mpi_comm_block** and is used in **get_fresh_remote_bndries** (**mpi_communications**) to exchange local channel data between sectors (**get_block_pe_translation_arrays** is modified to allow this as detailed in comments). Parameters **first_block_group_id** and **last_block_group_id** are introduced (along with other new parameters) in module **communications_parameters**.

The division of channels between tasks in **mpi_comm_block** is made in routine **init_my_channels** (**initialise**). If the remainder from dividing **number_channels** by **number_of_pes_per_sector** is **rem**, the first **rem** tasks in **mpi_comm_block** have one more channel than the remaining tasks. Parameters **my_num_channels**, **my_channel_id_1st** and **my_channel_id_last** are set, along with arrays **counts_per_pe(:)**, **disp_for_pe(:)** and **pe_for_channel(:)** which are used in MPI communications. The bulk of the minor changes across large numbers of modules/routines is to replace array descriptors such as **channel_id_1st:channel_id_last** with **my_channel_id_1st:my_channel_id_last** and **number_channels** with **my_number_channels**/**my_num_channels**, and modifications to sums over global variables (populations, dipole expected values) when these are needed. There is also some rewriting in **checkpoint.f90**, **wavefunction.f90** and **io_routines.f90** to gather or scatter outer wavefunction data and write/read it (by the blocks master) in ordered chunks obtained from or sent to each task in **mpi_comm_block**. The wavefunction files have data in a modified order when **number_of_pes_per_sector** /= 1, to avoid accumulation of the wavefunction into full-size arrays, thus restarts and runs over several jobs should use the same value.

The modules affected (to a greater or lesser extent) include:

checkpoint
communication_parameters
distribute_wv_data
global_linear_algebra
inner_propagators
inner_to_outer_interface
io_routines
kernel
local_ham_matrix
mpi_communications
outer_hamiltonian
outer_hamiltonian_atrlessthanb
outer_to_inner_interface
propagators
wavefunction
tdse (main program)

The major set-up (and the main communication during iterations) for the data transfer within **mpi_comm_block** is in **outer_hamiltonian.f90** The routine **test_outer_hamiltonian**

originally worked out and set up the required channel couplings for the various potential terms in the multiplication of the wavefunction vector by the Hamiltonian matrix at each step. It now also identifies the coupled channels as being on the same or a separate MPI task. Having worked out these couplings, arrays are set up listing the coupled tasks and the channels required to be transferred. It is assumed that if channel 'a' is coupled to channel 'b', then channel 'b' is coupled to channel 'a' and thus a series of **MPI_SendRecv** calls occur in the same order for each task in the communicator as the channels are cycled through in ascending order. The set up goes through each part of the Hamiltonian potential ('we', 'wp', 'wd') in turn checking for new channels to be accumulated in the wavefunction arrays, For 'wp' potentials, a list of channels whose wavefunction derivative needs to be sent is also made.

Arrays to hold actual wavefunction data, refreshed at each iteration, are set up to contain all the channels required:
psi_from_neighbours(:), psi_to_neighbours(:),
deriv_psi_from_neighbours(:,:), deriv_psi_to_neighbours(:,:)
The arrays are allocated in **test_outer_hamiltonian** and are used to transfer data within the communicator. The first dimension of **deriv_psi_from_neighbours** is (**x_last**–**x_1st**+1), the second is the maximum value of the number of channels a task needs to receive. The dimensions are collapsed in **psi_from_neighbours** as this is also used in module **outer_hamiltonian_atrlessthanb** and dimension size is set according to which is the greater of (**x_last**–**x_1st**+1) and (2***nfdm**–**half_fd_order**).

The arrays are used in new routines **prep_w_comm_block_ham_x_vector_outer** and **prep_deriv_w_comm_block_ham_x_vector_outer** which are called at the start of the main iteration loop in **ham_x_vector_outer** (once for each solution isol = 1, **numsols**). These perform all **mpi_comm_block** transfers required for subsequent calls to the various routines '**incr_**<**…**>' called from **apply_long_range_potential_matrices** further in the loop. The **incr_**<…> routines split the loop over connected channels into two loops, for those purely on the local task and those collected from other tasks into the **psi_from_neighbours** (and **deriv_psi_from_neighbours**) arrays.

The 'channel size' part of the dimension of **psi_to_neighbours** depends on a logical parameter **sendrecv_yes**, currently hardwired to **.true.** as a module parameter. In this setting **MPI_SendRecvs** are used to transfer data between tasks and **psi_to_neighbours** can be reset before each call. A second option uses **MPI_ISend** and **MPI_IRecv**, with an **MPI_WaitAll** call at the end of the loop over the linked tasks. In this case **psi_to_neighbours** must range over all the channels that need to be send to other pes as it cannot be overwritten before the **MPI_WaitAll** command, so it is more memory intensive (the **sendrecv_yes** flag is checked at allocation). As long as the above rule for potential coupling is satisfied, this option should not be needed, but can be tested for efficiency.

Note: while introducing the new parallelization, it was noticed that an internal legacy variable **mpi_comms_method_desired** in module **communications_parameters** was set (hardwired) in the code to a parameter **use_sends_recvs** rather than **use_mpi_collective**. This affected various communications already present between regions and in the outer region (mainly in module **mpi_communications**). The parameter was reset to use MPI collectives.

(3.1) MPI verses OpenMP

The OpenMP commands already present in the code are unaffected apart from one case. This means that MPI parallelisation and OpenMP parallelisation can be combined flexibly to fit the configuration of the hardware. Examples of this are given below. The MPI parallelisation may be limited by available memory on nodes, OpenMP parallelisation by node substructure: on ARCHER2 OpenMP threads are usually limited to a maximum of 8 for efficient use.

The one OpenMP loop affected by the MPI change is the main iteration loop in **ham_x_vector_outer**. The original code uses a collapsed OpenMP directive to parallelise over the solutions loop (1:**numsols**) and the immediately following loop over channels. In the new code, due to the need to call the **prep_**<…> routines, the OpenMP parallelisation is over **my_num_channels** and the loop over solutions above it is serial. This means that for runs in which **numsols** > 1 pure OpenMP runs may be slightly less efficient as the numsols loop is embarrassingly parallel with no communications. This will be investigated further (outside the scope of the eCSE): the test results show that for large cases invoking parallelisation over channels the efficiency gains outweigh this loss. Other cases where loops are collapsed for OpenMP parallelism are unchanged.

(3.2) Sample test results

We show results for some of the RMT test suite cases examined in section 2. The particular cases are not expected to show large performance advantages. The coding was validated for accuracy against all the test cases. For the Ne+ example, with 10 inner region pes the outer region time dominates and **number_of_pes_per_sector** = 2 substantially reduces wall clock time. The inner region time subsequently dominates. In the neon case, the behaviour is similar for 120 inner pes: reducing these to 41 and keeping **number_of_pes_per_sector** = 1 gives better performance than 120 IR pes and **number_of_pes_per_sector** = 2, In the 120 case, the actual outer region time per iteration scales perfectly. For Ar_LS, the outer region time dominates for 63 inner pes, and scaling is near perfect, giving an overall performance improvement. For Ar_jK, the numbers show that using 83 inner pes with **number_of_pes_per_sector** = 1 costs 0.16 CUs while 196 inner pes with **number_of_pes_per_sector** = 2 costs 0.1 CUs.

Ne+: 10 LML (L) blocks, 10 inner pes, 72 * n outer pes, full time in seconds (guide)

Number of PEs per sector (n)	Time (Seconds)
1	114
2	76
3	73*

* outer time per iteration: 0.005, inner time: 0.013

Ne: 71 LML (L) blocks, 72 * n outer pes, full time in seconds (guide)

Number of Pes per sector (n)	Time (seconds)	Number of Pes per sector (n)	Time (seconds)	Time breakdown (per iteration)	
41 Inner PEs		**120 Inner PEs**		Inner (s)	Outer (s)
1	189	1	108	0.018	0.014
2	182	2	85	0.014	0.007
		3	86	0.014	0.005

Ar_LS: 196 LML blocks (405 channels), 63 inner pes, 82 * n outer pes, full time in seconds (guide)

Number of PEs per sector (n)	Time (Seconds)
1	1009
2	530
3	383

Ar_jK: 169 LML blocks (1344 channels), 143 * n outer pes, time in seconds for 1460 timesteps (guide)

Number of Pes per sector (n)	Time (seconds)	Number of Pes per sector (n)	Time (seconds)
83 Inner PEs		**169 Inner Pes**	
1	319	1	246
2	312	2	105
3	320	3	101*

* outer time per iteration 0.044, inner time 0.06

We may now look at some figures for larger cases before examining the combined inner and outer parallelisation in detail. For the F- case, decreasing the number of inner region tasks from 20 to 5 nodes halves the CU cost as outer region time dominates. Doubling the number of outer region tasks nearly halves the execution time and slightly reduces the overall charged time. In the 'smaller' Xe case, the original outer region run used 8 OpenMP threads per MPI task (with 1 thread per task in the inner region). With 29 inner region blocks, the timing is slightly improved by replacing the OpenMP with **number_of_pes_per_sector** = 8. Using fewer inner region blocks (5), has a CU cost. We will see that an intermediate number of inner region tasks can maintain the CU efficiency of the 53 node case.

F-: 2500 symmetries, 3774 channels (times for 100 cycles)

Nodes	IR Pes	OR PEs	OR PEs per sector	Wall time	Charged time
30	2560	1280	1	14.7	441 * 128
15	640	1280	1	14.7	221 * 128
25	640	2560	2	8.4	**210 * 128**

Xe: 3619 LML blocks, 16278 channels, (times for 100 cycles)

Nodes	OR nodes	OR tpn*	IR PEs	n**	Threads	Wall time	CUs/100cyc
29	24	16	640	1	8	40.7	0.33
29	24	128	640	8	1	**40.7**	**0.33**
53	24	16	3712	1	8	17.3	0.25
53	24	128	3712	8	1	14.5	0.21

[outer time per cycle (s): 0.17 for n=1, 8 threads; 0.14 for n=8, 1 thread]

* tpn = tasks per node

** n = number of OR PEs per sector

The larger Xe test case needs the OpenMP parallelism to make full use of ARCHER2's 128 cores per node as its memory requirements mean there can be a maximum of 4 (rather than 8) MPI tasks per CCD (compute complex die) in the outer region. As with the smaller case, OpenMP is used in the outer region (up to 4 threads here) but not the inner region.

Xe: 8635 LML blocks, 38850 channels, (times for 100 cycles)

Nodes	OR nodes	OR tpn	IR nodes	n	Threads	Wall time	CUs/100cyc
124	56	32	68	1	4	75	2.6
124	56	64	68	2	2	70	2.4
180	112	64	68	4	2	39	2.0
80	56	32	24	1	4	75	1.7
80	56	64	24	2	2	71	1.6
136	112	64	24	4	2	61	2.3

In this case the channel parallelization improves the 68 inner nodes test, but the flexible inner region parallelization gives the best benefit. A wider range of combinations of inner and outer core numbers was tried for the three larger test cases. Generally the all-MPI outer region outperforms the 1-sector task plus OpenMP approach to a small extent, though the large tests all ask for a single solution (numsols = 1).

(4) Combined 'best' parallelisation

Here we present charts of 'CUs per 100 time steps' for the three larger test cases using a range of choices of task combinations. We show the optimum performance across various total node counts, followed by a breakdown of points showing the different combinations. A direct algorithm for 'instant' detailed matching of inner to outer node use is very much case-dependent, but we may provide a straightforward guide for preliminary performance tests before a very large simulation is run, (see the next section) to minimize computational costs.

(i) F-

The optimum choice of inner and outer division of node use gives a consistent performance across a range of nodes (the numbers from the previous F- figure are included for convenience):

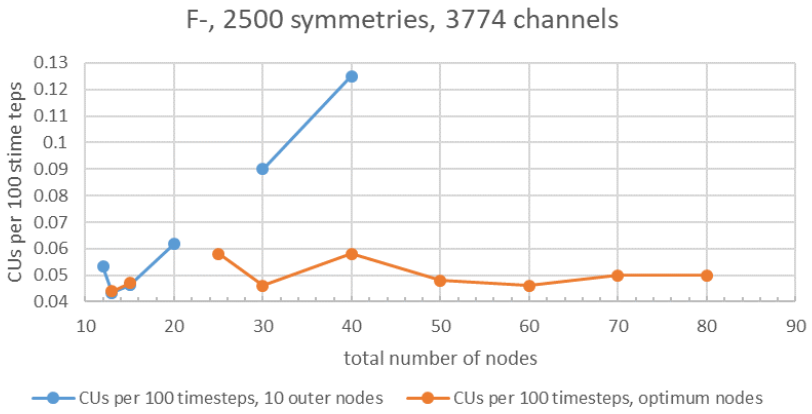

Figure 5: ARCHER2 optimal computation cost for various node counts, F-.

The original code test requires 10 outer nodes and 20 inner nodes. The optimum combinations give a fairly consistent performance of between 0.04 and 0.05 CUs per time step up to a total of 80 nodes, meaning that the same calculation can be performed in a fraction of the time for the same cost. The best performance is given for 3 inner and 10 outer nodes, but if wall-clock time is to be prioritised than the 80-node combination (30 inner plus 50 outer: number_of_pes_per_sector = 5) runs 5 times as fast. Note that at 30 nodes we get a factor of 2 speed-up form the original code (with a change from 20+10 to 10+20 inner+outer nodes).

The breakdown of performance is as follows:

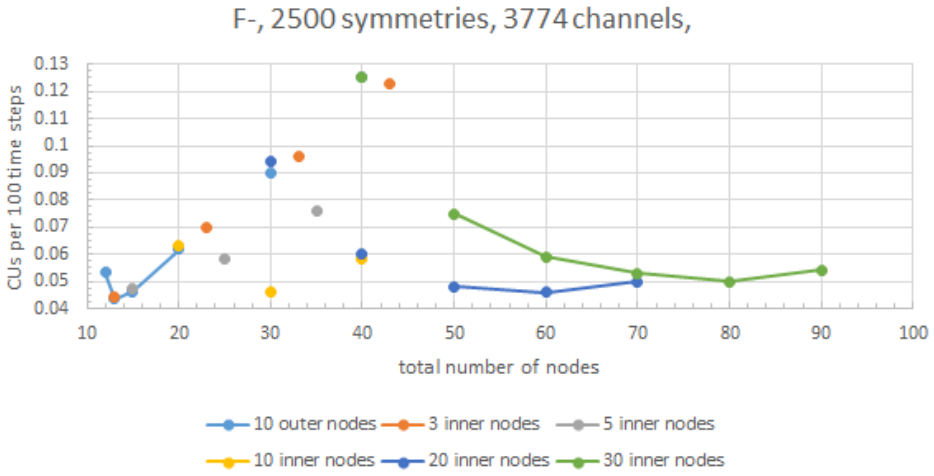

Figure 6: ARCHER2 computation cost breakdown for various node counts, F-

The general pattern is that using the outer channel parallelization improves performance as long as outer time is dominant, then adds to the cost. The light blue points represent the earlier values (the two values for 30 nodes show two calculations and give an idea of timing variation on ARCHER2.

(ii) Xe (smaller case)

The optimum choice of node distribution again gives slightly different behaviour to the F- case:

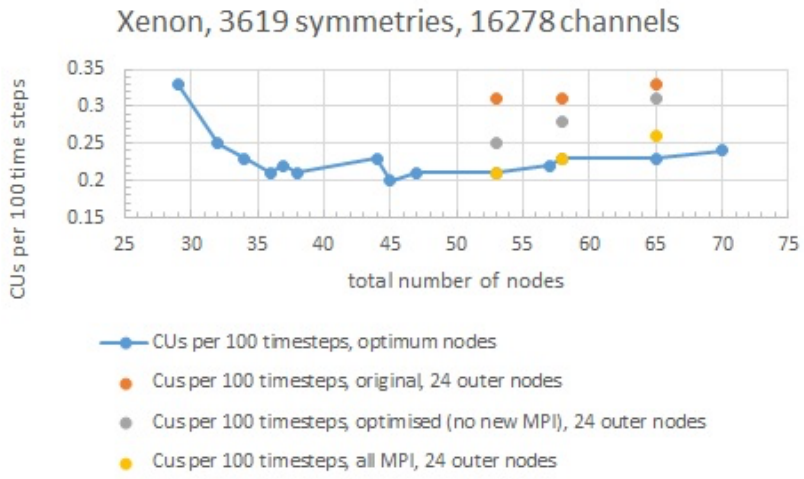

Figure 7: ARCHER2 optimal computation cost for various node counts, Xe (3619).

The case requires a minimum of 384 MPI tasks in the outer region to define the outer grid and these use 1 CCD (8 cores) each for memory reasons, thus the outer region node count is generally 24, except for tests with **number_of_pes_per_sector** = 12 which have 36 outer nodes. The original case used 29 inner region nodes (the smallest number with more than 3619 cores). The points above the optimal results (with 24 outer nodes and increasing numbers of inner nodes) illustrate the successive effects of the inner region serial optimization and the all-MPI outer region (**number_of_pes_per_sector** = 8). As more inner region nodes are added, the serial optimization, which at lower core counts brings more arrays fully into L3 cache, has a smaller effect as the original complex arrays are smaller. Compared to the original code, CU use for this case has been reduced by 1/3 for the main simulation (with some start-up overhead only noticeable for short runs).

The breakdown of the optimal node distributions is summarized as:

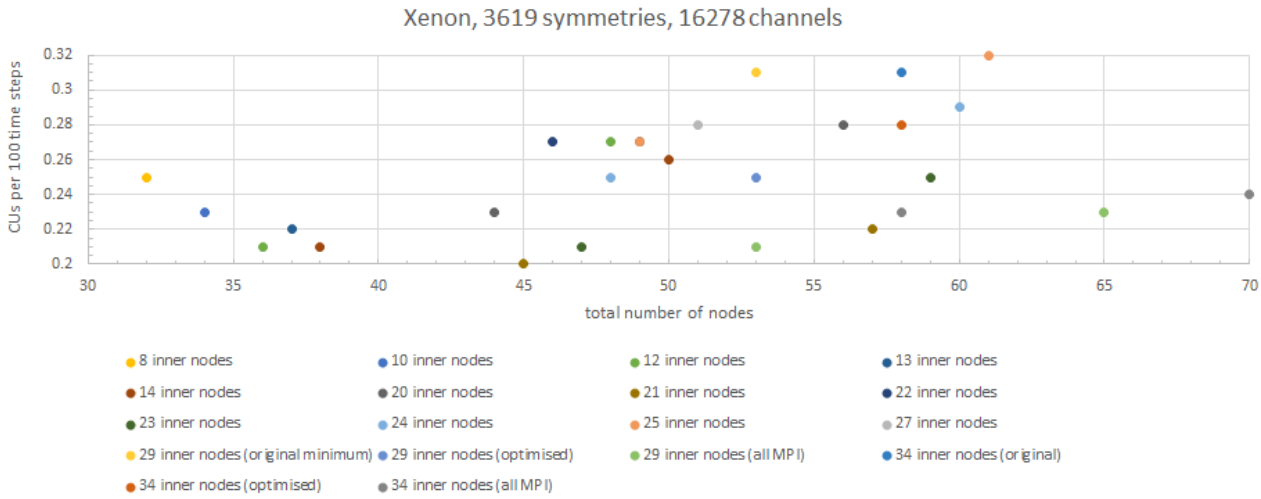

Figure 8: ARCHER2 computation cost breakdown for various node counts, Xe (3619).

While this data may look confusing to begin with, the main rule to take away is that with the outer region MPI parallelization in place, the performance is reasonably constant for 36 (12 inner) nodes through to 47 (23 inner nodes) and 53 (29 inner) nodes. It is still reasonable for 55 nodes (21 inner nodes plus 36 outer nodes) and acceptable for fast wall clock times up to 70 nodes. The particular high performance point here is for 21 inner nodes and 24 outer nodes. The gap between 38 and 44 nodes is due to the algorithm used to distribute the inner region LML blocks among tasks producing load-balance warnings at these values. Some runs with the load-balancing restrictions partially lifted will allow sample results at these values, such as for 44 nodes with performance reduced, however the new Huffman tree coding and planned reduction in MPI calls (see section 2) should produce better results. Similarly, as the number of inner tasks becomes closer (from below) to the number of LML blocks (48-52 nodes) performance with the current algorithm is degraded.

In this particular case the MPI channel parallelization has allowed a wider range of node choices for good performance. For the range of values tested, the complicated communications overheads stop there being a clear division between inner region dominance and outer region dominance.

(iii) Xe (larger case)

In this case, the optimized inner region code still produces the most efficient choice of node division. The improvement around 76 nodes relative to the previous figure is due to a mixture of some serial optimization, the choice of **use_mpi_collective**, and replacing 4-thread OpenMP in the outer region with **number_of_pes_per_sector** = 2 and using 2 threads. This optimization is helpful but relatively small around the optimized inner region values. The inner region serial optimization is less effective at the larger node counts where outer region work dominates (again the small differences between the yellow and grey results show the general timing variability of ARCHER2. The outer region parallelization allows reasonable performance at larger node counts if fast wall-clock time is needed. The 'sweet spot' at 110 nodes has 26 inner nodes and **number_of_pes_per_sector** = 3 (2 threads). Compared to the original code test, the CUs per 100 cycles are altered by a factor ~0.6 from ~2.6 to ~1.6.

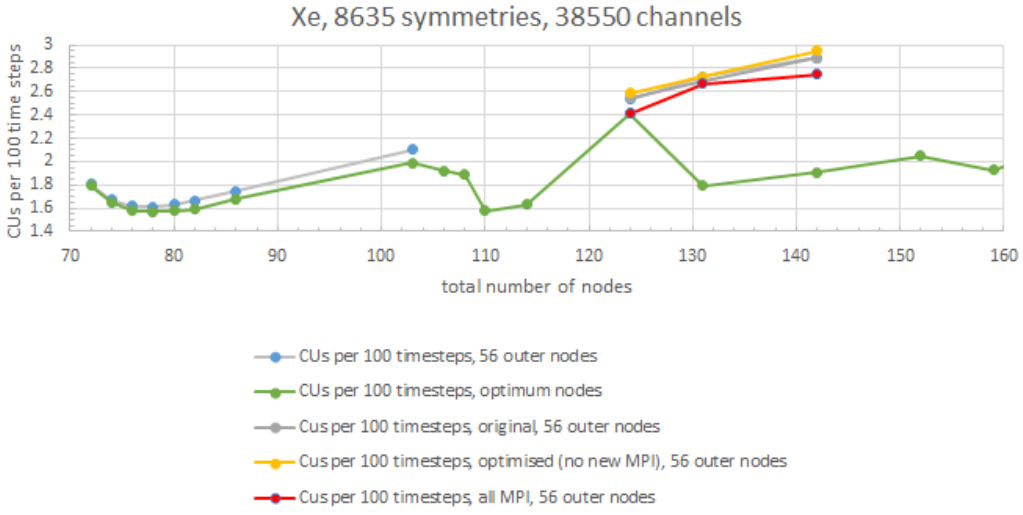

Figure 9: ARCHER2 optimal computation cost for various node counts, Xe (8635).

The performance of up to ~2 CUs per 100 cycles continues to 240 nodes as may be seen by the breakdown below.

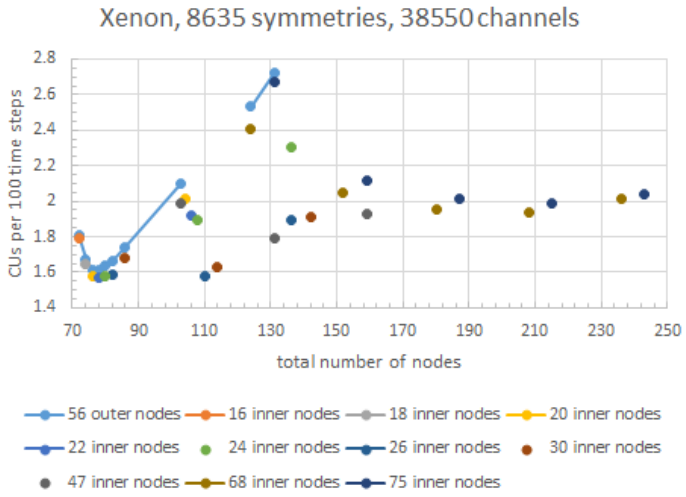

Figure 10: ARCHER2 computation cost breakdown for various node counts, Xe (8635).

A similar pattern to the F- case may be seen as more MPI tasks are used for outer region channel parallelization. 26 inner nodes with 56 outer nodes is close to the 78 node (22+56) 'best' performance, with 26 inner nodes and 84 outer nodes equally good, then 26 + 112 nodes losing the good performance. At higher inner region node counts such as both 68 and 75 inner nodes, the performance improves as **number_of_pes_per_sector** increases from 2 to 4, stays the same at 5 and starts to degrade slightly at 6.

Sample timings, as opposed to CU use, for the F- and large Xe cases for 'optimum', 'fast' and 'intermediate' node choices, can be found in the accompanying eCSE final report.

(5) Summary

In the large-scale benchmarks, the correct combination of inner and outer region parallelization has reduced the cost of a calculation by 1/3, and in one optimal case (F-), by ½. The new flexibility also allows large calculations to be performed using a wide range of core counts at the same cost, so that users can choose whether to submit a longer running job requiring fewer nodes, or a larger job for faster turnover if ARCHER2 is not too busy. This flexibility also extends the range of hardware on which RMT may be run, from local parallel machines to Tier2 clusters, to ARCHER2 and even modern peta/exa-scale machines for large enough calculations.

Some immediate follow up activities include:

- Introduce the Huffman Tree approach to inner-region task allocation. This should be more efficient than the current distribution, and can allow mixing of the several tasks per LML block approach with the several LML blocks per task approach (and is already localized in module **mpi_layer_lblocks**).
- Revisit the inner-region inner parallel set-up so that a single set of MPI_isend/irecv calls can produce all the off-node information required for the local LML blocks ready for the **parallel_matrix_vector_multiply** operations (as is now done in the outer region). This can re-introduce larger arrays than may be wished for in some cases but should in principle further reduce the amount of inner-region communications during a time step and thus further speed up the code.

A single automated algorithm for the optimal distribution of tasks is not yet practical as the range of systems studied, from atoms to molecules, from linear to elliptical and circular polarization, with a full range of descriptions of 'target' electronic structure, is too large (any such algorithm will need further adjustment after the follow-up optimization above). However there is a relatively straightforward set of preliminary operations for large-scale simulations which we can recommend. To enable this, a commented line in the main control program **tdse** may be used for the preliminary tests:

```
!  alternative for detailed algorithm testing:
!       DO timeindex = start_of_chkpt_timeindex, start_of_chkpt_timeindex + 1000
      DO timeindex = start_of_chkpt_timeindex, end_of_chkpt_timeindex
```

where the '1000' steps may be modified/reduced as required for the tests. The RMT standard output gives time taken for every 20 steps and may be examined to get the appropriate timing

information. This approach means the calculation input is not affected and the test simulation just stops when wanted.

- Having decided on the number of outer region subregions needed, begin with number_of_pes_per_sector = 1. Again decide on an appropriate initial number of inner region pes, either starting with one task per LML block, or (for simulations that have been run previously) a working larger number.
    - If needed, use a machine appropriate number of OpenMP tasks if memory per node is a problem
- Run short tests which stop after a few hundred time steps (set in tdse.f90, the main program, or in input.conf) to see whether the inner or outer region is dominating: the **timings_desired** flag set true will produce data in (in particular) files **timing_inner.**<job_id> and **timing_outer0.**<job_id>
    - After timings have stabilized after start-up (and avoiding occasional points with longer timings due to I/O backup), the first column (**iteration**) gives roughly the working time in the inner/outer region, while the last column (**eshare**) gives roughly the idle wait time in this region, for a time step.
- If the outer region seems to be dominating, back this up by doubling the number of outer region tasks and setting **number_of_pes_per_sector** = 2 (keep the number of OpenMP threads per task to the minimum required by memory). If time is halved, the outer region is dominating. Run with more outer pes and **number_of_pes_per_sector** increased by one until the CU cost increases (ie when performance does not increase linearly).
- For both this set of outer pes and the original set with **number_of_pes_per_sector** = 1 (ideally), reduce the number of inner region pes until the CU cost is minimized. Compare this to the previous cost figure.
    - If the new figure is substantially lower, use this number of inner tasks as a baseline and try increasing **number_of_pes_per_sector**. [On ARCHER2 this will be complicated by factoring in 128 (or 64 or 32 plus OpenMP) to make optimum use of each node. The RMT code already allows for separate MPI task/OpenMP thread combinations in the inner and outer regions. Example scripts are included in the test suite directory.]
    - If there is no significant difference, or if reducing the number of inner-region pes makes performance worse, try starting from larger numbers of inner region pes (more than the number of LML blocks) and once the outer region time dominates, increase **number_of_pes_per_sector** until no further CU cost saving can be made. Compare with the minimized inner region figure.
- Choose the combination that suits your workflow for long simulations the best (ie minimize CU cost or minimize run time for a given cost.

These tests follow a straightforward flow and can save significantly on the cost/run time for the large simulation. Users with more interest in physics than computational cost may find this process frustrating at first, but the preliminary work requires very short runs and can pay large dividends. Alternatively, we may think of these performance enhancements as providing reasonably good performance, even if the optimisation procedure is not followed. This is important for non-expert users- several 'gotchas' which would have killed calculations previously (e.g. not allocating sufficiently many inner-region processors) have been lifted.

Further work could enable even more user-friendly features, such as automatically assigning IR/OR PEs based on the available resource.

Acknowledgements

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

References

[1] Nat. Phys., 3, 381 (2007)
[2] Phys. Today, 64, 36 (2011)
[3] Chem. Phys., 414, 1 (2013)
[4] Nat. Photon., 8, 195 (2014)
[5] Nat. Photon., 8, 205 (2014)
[6] J. Phys. Chem. A, 104, 5660 (2000)
[7] Angewandte Chemie International Edition in English, 25, 971 (1986)
[8] J. Photochem. Photobio. C: Photochem. Rev., 4, 145 (2003)
[9] Opt. Lett., 41, 709 (2016)
[10] see, for example, Phys. Rev. Lett. 117, 093201 (2016)
[11] Comput. Phys.Commun., 250, 107062 (2020)
[12] https://gitlab.com/Uk-amor/RMT/rmt
[13] recently discussed in: ICS '21: Proceedings of the ACM International Conference on Supercomputing (2021), 127, https://doi.org/10.1145/3447818.3460364