Implementing distributed-memory in the 2D-3V drift-kinetic edge code

M. R. Hardman[1,2], J. Omotani[3], and M. Barnes[2]

[1] Tokamak Energy Ltd, 173 Brook Drive, Milton Park, Abingdon, OX14 4SD, United Kingdom

[2] Rudolf Peierls Centre for Theoretical Physics, University of Oxford, Clarendon Laboratory, Parks Road, Oxford OX1 3PU, United Kingdom

[3] Culham Centre for Fusion Energy, Culham Science Centre, Abingdon, Oxon, OX14 3DB, United Kingdom

E-mail: `michael.hardman@tokamakenergy.co.uk`

1. Introduction

The aim of this report is to describe the progress towards implementing the drift-kinetic model described in earlier ExCALIBUR reports [1, 2, 3, 4, 5, 6, 7, 8, 9]. The main outstanding challenge is the successful testing of the 2D-3V model in the presence of a wall boundary when there is variation in the radial direction. To keep this report brief, we do not reintroduce the model, but instead focus on describing the code development that has taken place to support the testing of the model implementation. The source code described in this document can be found here `https://github.com/mabarnes/moment_kinetics`.

In the previous report on the 2D-3V model, we were able to demonstrate using the method of manufactured solutions that the implementation of the wall boundary condition was likely correct by showing that the error in the numerical solutions became small as the number of velocity elements $N_{element}$ increased in cases where there was no radial variation and the radial electric field $E_r = 0$ everywhere in the simulation domain. Since the E_r does appear in the wall boundary condition, this demonstration is not sufficient to conclude that the entire model is implemented correctly. Indeed, when $E_r \neq 0$, we were not able to demonstrate convergence with increasing $N_{element}$.

The reasons for this lack of convergence in the 2D case remain unclear. One possibility was that the numerical resolution available for the tests was inadequate to demonstrate convergence. In the 1D-3V case, errors of order 10^{-8} were only reached for $N_{element} \approx 16$, whereas in the 2D-3V case we are only able to scale to $N_{element} = 4$ using only a single node on our preferred HPC resource. To perform larger tests, we implemented distributed-memory MPI alongside the shared-memory MPI that was implemented into the code by J. Omotani. This feature now allows to code to utilise thousands of cores on a HPC resource, and allows much larger problem sizes to be considered. Both these MPI features use the `Julia` MPI library [10].

2. Implementing distributed-memory MPI

To understand the description of the implementation, it is important to know that the ion species are evolved with a $(v_\parallel, v_\perp, z, r)$ grid whereas the neutral species are evolved with a $(v_z, v_r, v_\zeta, z, r)$ grid. There are N_{si} species and N_{sn} species. (Strictly, only $N_{si} = N_{sn} = 1$ is supported because of the initilisation options).

In the shared-memory MPI implementation arrays of dimension $(N_{v_\parallel}, N_{v_\perp}, N_z, N_r, N_{si})$ and $(N_{v_z}, N_{v_r}, N_{v_\zeta}, N_z, N_r, N_{sn})$ are parallelised across all cores in a given node (or shared-memory region, if smaller than a node). The parallelisation operates with each core looping over a subset of array elements, with all-to-all communicates making the results of the calculations consistent at checkpoints in the calculation. The all-to-all communications are assumed to be relatively inexpensive because they occur only within a shared-memory region, rather than across nodes.

Naturally, a shared-memory MPI code can only scale to the size permitted by the memory on a single shared-memory region. To extend beyond this limit, we implemented distributed-memory MPI for the (z, r) spatial coordinates. We limited the distributed memory to the spatial coordinates for the following reasons: (i) velocity integration can be neatly carried out without significant communication overhead in the shared-memory formalism, (ii) since we can run simple 1D-3V problems adequately in a single shared-memory region, extension to 2D-3V problems seems to require extra memory only because we need a larger spatial problem size, (iii) all particle species share common spatial coordinates (z, r) allowing the use of the same domain decomposition for all distribution functions, fields, and fluid moments, (iv) parallelisation of the spatial coordinates can be carried out with minimal inter-node communication as described shortly.

All coordinates are assumed to be discretised with a Chebyshev spectral-element method. Within each element the derivative is computed using a Chebyshev transform (via an FFT carried out by FFTW). At element boundaries we must enforce continuity by taking the average or by copying the value from one element to another (depending on the upwind direction). In this framework we can obtain an efficient parallelisation if we parallelise over elements: then the computation of the FFT is always performed using local data and we must only communicate boundary points through the MPI interface. This necessitates cyclic communication between cores holding individual elements of arrays of size $N_{v_\parallel} \times N_{v_\perp} \times N_r \times N_{si}$ and $N_{v_z} \times N_{v_r} \times N_{v_\zeta} \times N_r \times N_{sn}$ (for the z derivatives) and $N_{v_\parallel} \times N_{v_\perp} \times N_z \times N_{si}$ and $N_{v_z} \times N_{v_r} \times N_{v_\zeta} \times N_z \times N_{sn}$ (for the r derivatives). These arrays are not as large as the entire distribution function, so we have avoided the most expensive of operations – the all-to-all redistribute.

3. The domain decomposition

We now provide some details on the decomposition of the (z, r) domain. As an example, in figure 1 we show a domain where which we have broken up into different shared-

memory regions. There are `nblocks=6` shared-memory domains, the z axis is broken into `z_nchunks=3` chunks, and the r axis is broken into `r_nchunks=2` chunks. There are `nrank_per_zr_block=block_size=4` cores per shared-memory block.

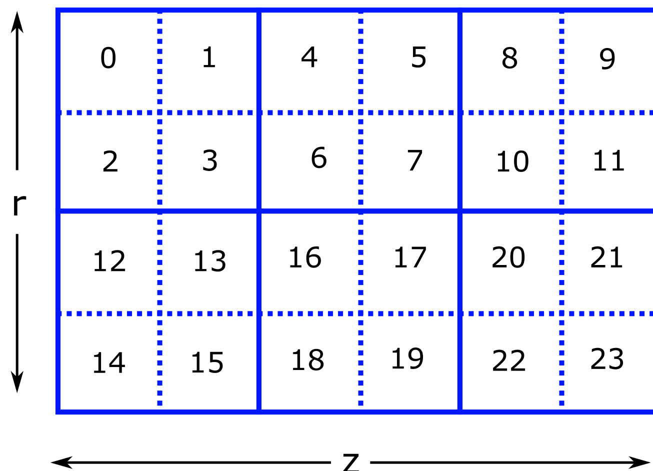

Figure 1: A (z, r) domain where each square represents a core. The domain is split up into `nblocks=6` separate shared-memory regions, with the `irank` in the global MPI communicator `MPI_WORLD` indicated for each core. There are `nrank_per_zr_block=block_size=4` cores per shared-memory block.

It is important to note how the domain decomposition is related to the number of elements in each spatial coordinate: we cannot have more processes than the number of elements allow. In practice, we specify the variables `r_nelement_global`, `r_nelement_local`, `z_nelement_global`, and `z_nelement_local` that determine the total number of elements in each grid and the number locally in each chuck of the r or z domains. We also specify the total number of ranks `nproc` supplied to the program via the following command.

```
mpirun -n nproc julia --project run_moment_kinetics.jl runs/*.toml
```

Then, we calculate the number of chunks and teh number of blocks in the following way.

```
# get information about how the grid is divided up
# number of sections ‘chunks’ of the x grid
r_nchunks = floor(mk_int,r_nelement_global/r_nelement_local)
# number of sections ‘chunks’ of the z grid
z_nchunks = floor(mk_int,z_nelement_global/z_nelement_local)
# get the number of shared-memorz blocks in the z r decomposition
nblocks = r_nchunks*z_nchunks
# get the number of ranks per block
nrank_per_zr_block = floor(mk_int,nrank_global/nblocks)
```

Finally, we must make sure that `nproc` is an integer multiple of `nblocks`.

To use the (z, r) domain we must introduce extra labels upon which we can use the standard MPI functions. We introduce an integer `iblock` which indexes the blocks, and an integer `irank_block` which indexes the ranks within a block. We show these integers in our example in figure 2.

```
# assign information regarding shared-memory blocks
# block index -- which block is this process in
iblock = floor(mk_int,irank_global/nrank_per_zr_block)
# rank index within a block
irank_block = mod(irank_global,nrank_per_zr_block)
```

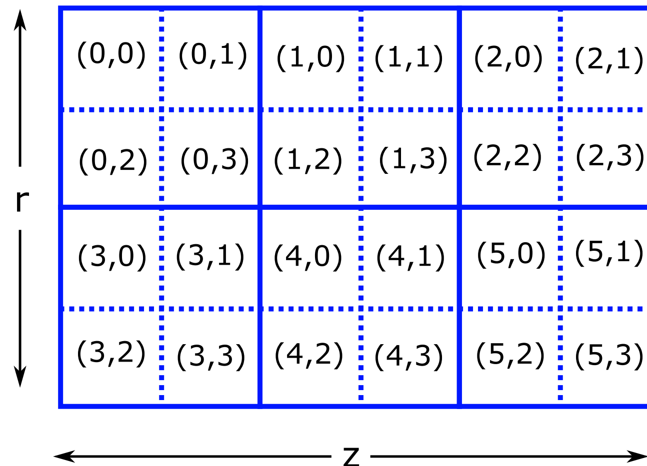

Figure 2: The (z, r) domain from figure 1 where each square represents a core. The cores are labelled with `(iblock,irank_block)`.

In order to do the spatial derivatives we need to set up communications across shared-memory blocks. To do this we must map our `iblock` integers to a 2D grid that can hold the lead processes in the (z, r) domain. We define the integers

```
z_igroup = floor(mk_int,iblock/z_nchunks) # iblock(irank) - > z_igroup
z_irank =  mod(iblock,z_nchunks) # iblock(irank) -> z_irank
# iblock = z_igroup * z_nchunks + z_irank_sub
```

which determine which group of z points each lead process is in, and which rank each process has within a group. We do our z spatial derivatives within a single `z_igroup` grouping, whereas we do our r spatial derivatives within a single `z_irank` grouping. The indices in our example are given in figure 3. Note that only lead processes are involved in the inter-block communication.

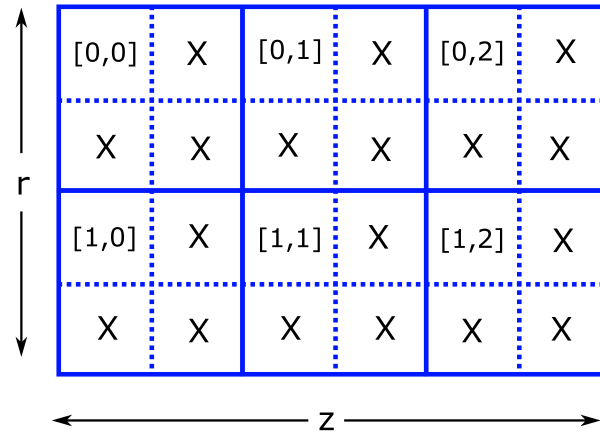

Figure 3: The (z, r) domain from figure 1 where each square represents a core. The lead process on each block is labelled with `(z_igroup,z_irank)`. Processes that are not lead processes are labelled with 'X'.

To carry out the communication, we must use the MPI function library to make use of the labels that we have defined. An invaluable function is `Comm_split`. It is conventional to split the 'communicator' `MPI_WORLD` that contains all the `nrank` processes into useful subsets. This is carried out with the following command

```
MPI.Comm_split(comm,color,key)
# comm -> communicator to be split
# color -> label of group of processes
# key -> label of process in group
```

We assign `color` and `key` appropriately in each case using the integers that we have defined above. The domain decomposition is carried out in the source code in the file `https://github.com/mabarnes/moment_kinetics/blob/radial-vperp-standard-DKE-Julia-1.7.2-mpi/src/communication.jl`.

4. Application of the MPI formalism to larger problem sizes

To demonstrate that the distributed-memory MPI version of the code is working, we carry out MMS tests where we use the distributed-memory capability. The specific branch where inter-node MPI is implemented may be found at `https://github.com/mabarnes/moment_kinetics/tree/radial-vperp-standard-DKE-Julia-1.7.2-mpi`. We carry out a manufactured solutions test using the version of the code in commit cc0f04226ef0650171dd459de2236906083a09c6 and the input file `2D-sound-wave_cheb_nel_r_16_z_16_vpa_16_vperp_16.toml` in appendix Appendix A. This input file forces the solution to be a time-independent sine wave in space and a Maxwellian in velocity space. We vary

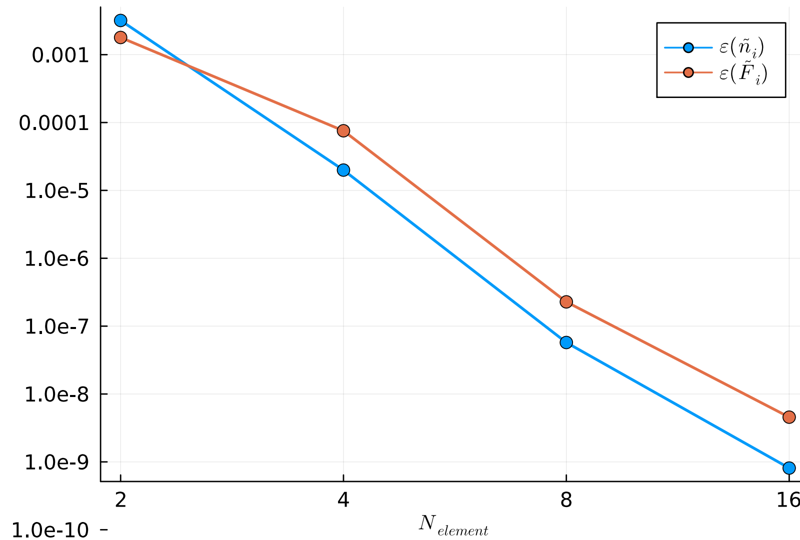

Figure 4: The measures of the numerical error on the ion density and distribution function in the collisionless sound wave MMS test.

`z_nelement=r_nelement=vpa_nelement=vperp_nelement=` $N_{element}$ from 4 to 16. We simulate to a time of $L_{\rm ref}/c_{\rm ref} = 1.0$, halving `dt` and doubling `nstep` as we double $N_{element}$. We do not include neutral species to avoid the significant extra memory requirements implied by a 3V velocity space. The largest simulation in the series required 32×48 cores on MARCONI to meet the memory requirements of the ion distribution function. The key output of the MMS test is the numerical error ϵ – this error should decrease with increasing resolution. The results of the test can be observed in figures 4 and 5. We see that the error decreases linearly on the log-log plots, revealing good convergence with resolution

To generate these figures from input file in the appendix, make the input files

```
2D-sound-wave_cheb_nel_r_2_z_2_vpa_2_vperp_2.toml
2D-sound-wave_cheb_nel_r_4_z_4_vpa_4_vperp_4.toml
2D-sound-wave_cheb_nel_r_8_z_8_vpa_8_vperp_8.toml
2D-sound-wave_cheb_nel_r_16_z_16_vpa_16_vperp_16.toml
```

by modifying `dt` and `z_nelement=r_nelement=vpa_nelement=vperp_nelement` appropriately. Then, run the following commands within the appropriate `mpirun` or `srun` commands where `input_file.toml` is a placeholder.

```
$ julia -O3 --project run_moment_kinetics.jl input_file.toml
```

To analyse the results of these simulations, run the script `run_MMS_test.jl` with the following command.

```
$ julia -O3 --project run_MMS_test.jl
```

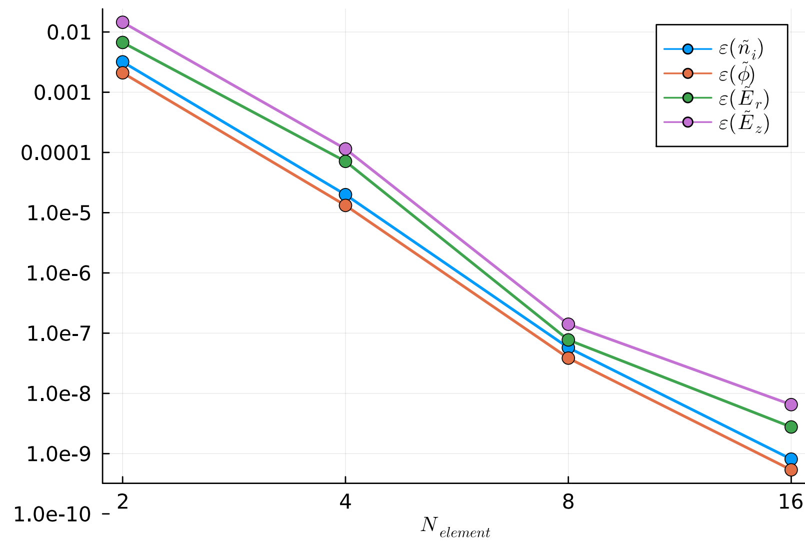

Figure 5: The measures of the numerical error in the ion density and the electric fields collisionless sound wave MMS test.

5. Dicussion and future outlook

Having performed a simple test of the upgraded software, we are now able to leverage the abilities of the code to test in more detail the 2D-3V implementation in the presence of wall boundary conditions. The development described here was vital because the problems with MMS tests and the wall boundary condition only arise where there radial variation in the problem. This places large memory requirements on any test. We will be able to determine if the errors observed previously are simply due to poor convergence of the velocity integrals and a lack of resolution.

Assuming that the problems with the wall boundary condition can be properly identified and fixed, the code is now in an excellent position to exploit a large fraction of a state of the art HPC resource. With relevant physics features implemented, the software promises to become a relevant tool for high-fidelity modelling of edge plasmas.

[1] Parra F I, Barnes M and Hardman M R 2021 *Excalibur/Neptune Report* 2047357–TN–03–01 M1.2
[2] Parra F I, Barnes M and Hardman M R 2021 *Excalibur/Neptune Report* 2047357–TN–05–01 M1.3
[3] Parra F I, Barnes M and Hardman M R 2021 *Excalibur/Neptune Report* 2047357–TN–07–01 M1.4
[4] Parra F I, Barnes M and Hardman M R 2021 *Excalibur/Neptune Report* 2047357–TN–09–01 M1.6
[5] Parra F I, Barnes M and Hardman M R 2021 *Excalibur/Neptune Report* 2047357–TN–11–01 M1.7
[6] Barnes M, Parra F I, Hardman M R and Omotani J 2021 *Excalibur/Neptune Report* 4:2047357–TN–01–02 M2.2
[7] Barnes M, Parra F I, Hardman M R and Omotani J 2021 *Excalibur/Neptune Report* 6:2047357–TN–01–02 M2.3
[8] Barnes M, Parra F I, Hardman M R and Omotani J 2021 *Excalibur/Neptune Report* 8:2047357–TN–02 M2.4
[9] Hardman M R, Omotani J, Barnes M and Parra F I 2022 *Excalibur/Neptune Report*
[10] Byrne S, Wilcox L C and Churavy V 2021 *JuliaCon Proceedings* 1(1), 68

Appendix A. `2D-sound-wave_cheb_nel_r_16_z_16_vpa_16_vperp_16.toml`

```
use_manufactured_solns_for_advance = true
n_ion_species = 1
n_neutral_species = 0
electron_physics = "boltzmann_electron_response"
run_name = "2D-sound-wave_cheb_nel_r_16_z_16_vpa_16_vperp_16"
evolve_moments_density = false
evolve_moments_parallel_flow = false
evolve_moments_parallel_pressure = false
evolve_moments_conservation = false
T_e = 1.0
Bzed = 0.5
Bmag = 1.0
rhostar = 1.0
initial_density1 = 0.5
initial_temperature1 = 1.0
initial_density2 = 0.5
initial_temperature2 = 1.0
z_IC_option1 = "sinusoid"
z_IC_density_amplitude1 = 0.001
z_IC_density_phase1 = 0.0
z_IC_upar_amplitude1 = 0.0
z_IC_upar_phase1 = 0.0
z_IC_temperature_amplitude1 = 0.0
z_IC_temperature_phase1 = 0.0
z_IC_option2 = "sinusoid"
z_IC_density_amplitude2 = 0.001
z_IC_density_phase2 = 0.0
z_IC_upar_amplitude2 = 0.0
z_IC_upar_phase2 = 0.0
z_IC_temperature_amplitude2 = 0.0
z_IC_temperature_phase2 = 0.0
charge_exchange_frequency = 0.0
ionization_frequency = 0.0
nstep = 8000
dt = 0.000125
nwrite = 800
use_semi_lagrange = false
n_rk_stages = 4
split_operators = false
z_ngrid = 9
```

```
z_nelement = 16
z_nelement_local = 1
z_bc = "periodic"
z_discretization = "chebyshev_pseudospectral"
r_ngrid = 9
r_nelement = 16
r_nelement_local = 1
r_bc = "periodic"
r_discretization = "chebyshev_pseudospectral"
vpa_ngrid = 9
vpa_nelement = 16
vpa_L = 12.0
vpa_bc = "periodic"
vpa_discretization = "chebyshev_pseudospectral"

vperp_ngrid = 9
vperp_nelement = 16
vperp_L = 6.0
vperp_bc = "periodic"
#vperp_discretization = "finite_difference"
vperp_discretization = "chebyshev_pseudospectral"

vz_ngrid = 9
vz_nelement = 16
vz_L = 12.0
vz_bc = "periodic"
vz_discretization = "chebyshev_pseudospectral"

vr_ngrid = 9
vr_nelement = 16
vr_L = 12.0
vr_bc = "periodic"
vr_discretization = "chebyshev_pseudospectral"

vzeta_ngrid = 9
vzeta_nelement = 16
vzeta_L = 12.0
vzeta_bc = "periodic"
vzeta_discretization = "chebyshev_pseudospectral"
```