ARCHER2-eCSE07-6: Improving parallel performances of the semi-implicit Particle-In-Cell code ECsim

Dr Elisabetta Boella (Lancaster University), Prof Giovanni Lapenta (KU Leuven, Belgium), Prof Maria Elena Innocenti (Ruhr-University Bochum, Germany), Alexei Borissov (EPCC, University of Edinburgh)

May 13, 2025

1 OpenMP Parallelism

A large majority of the work performed in ECsim is contained within a few loops over the particles. Specifically, for a simulation on a 256^2 2D grid with 4 particle species each with 40 particles per cell in each direction (approx 4×10^8 particles on 1 ARCHER2 node, of the non-IO related code `ComputeMoments` and `ParticleMover` take up the majority of the runtime at 63.5% and 11.6% respecively. Prior to this eCSE work, OpenMP directives were introduced around loops over particles within these functions, however race conditions caused by updates of grid cells when looping over the particles could cause inaccurate results. To address these issues investigations of atomics and OpenMP reductions were performed to see whether correct solutions could be obtained without any reduction of performance.

When using the atomic approach atomic pragmas were added prior to updates of grid quantities, similar to:

```
\#pragma omp atomic
Jxhs[is][X - i][Y - j][Z - k] += temp;
```

in `addJxh`. This is called within an OpenMP parallel loop over all the particles. Unfortunately a significant reduction in performance was observed (see Table 1). Although there is some increase in execution time for the initialisation and IO (as exepcted given fewer processes would be performing those operations), the largest increase was seen when computing the moments where there were the same number of ranks performing the operations. The reason for this increase is associated with the increased overhead of updating the grid arrays due to the atomic operations. This change was consistent for two and four OpenMP threads per process.

Another approach that resolves race conditions is to use reductions. OpenMP provides reductions on individual variables, but not for class members. To address this reductions were manually implemented by creating additional arrays that would accumulate values for each OpenMP thread before an additional function call that sums them together.

```
int thread = omp\_get\_thread\_num();
Jxhs\_thread\_local[is][X - i][Y - j][Z - k][thread] += weight[i][j][k];
void EMfields3D::addJxh\_reduction() \{
  int thread = omp\_get\_thread\_num();
  #pragma omp reduction(+:Jxhs)
```

	Baseline	2 OpenMP threads		4 OpenMP threads	
		Atomics	Manual reductions	Atomics	Manual reductions
Initialisation	1.43177	2.19742	2.22682	4.20231	4.13472
Moment gathering	64.8253	123.223	127.070	156.927	132.552
Field calculation	0.46465	0.57558	0.57695	0.83052	0.82099
Particle mover	10.7864	13.7574	10.9444	20.0290	11.7676
IO	17.4255	46.9061	24.7954	48.7488	28.7466
Total	98.0754	193.096	169.657	237.249	183.673

Table 1: Timing of baseline, atomic and manual reduction OpenMP runs, each with 2 and 4 threads per rank. All times in seconds.

```
  (for-loops over is, i, j, k)
    Jxhs[is][i][j][k] += Jxhs\_thread\_local[is][i][j][k][thread];
}
```

This results in slightly better runtime than the atomics implementation, but it is still significantly slower than the baseline runs. Increasing the number of threads to 4 yielded slower results in for both atomic and manual reduction approaches, however the difference between 2 and 4 threads was smaller for the manual reductions compared to the atomics. This suggests a lot of the overheads were to do with setting up the OpenMP parallel regions.

A number of other approaches were tried, for example implementing user defined OpenMP reductions to handle classes, however they gave either incorrect results or were impossible to compile. It was decided that using OpenMP, at least on CPUs, was not a viable approach. Restructuring the code to not use classes which would allow the use of built-in reductions might be a possible way forward, however it would be beyond the scope of this work to do so in order to get the same, or perhaps slightly better performance compared with the baseline.

2 Improving Parallel Performance

This workpackage was focused on improving the scaling performance of ECsim by investigating MPI bottlenecks and improving them to reduce communication costs when running at large scales. This work was performed after improving the memory access pattern for the mass matrix (described in section 4) as that gave a significant performance improvement and would significantly alter scaling performance. An initial profile was performed using MAP using a 2128^2 grid with 64 subdomains in the x-direction, and 32 in the y-direction. There were 2.85×10^8 particles in the simulation, and it was run on 2048 cores for 40 timesteps with IO turned off. MAP reported a total runtime of about 133 seconds of which 40.4% was spent on MPI communications. Of this 40.4%, more than 30% was spent communicating data between face subdomain faces either within `communicateInterp`, `communicateCenterBC` or `communicateNodeBC` in the solver. In all cases communication is point-to-point using `MPI_Send_recv_replace`. When running on a large number of nodes the data that needs to be communicated comes from fewer grid cells per process resulting in relatively small data transfers and a high overhead for initiating those communications. In this case average bandwidth is only 1.42MB/s per rank, with a peak of 105MB/s.

There are a couple of approaches that could improve performance. One could be to

move away from MPI point-to-point communication in favour of remote memory access (RMA). This approach has been shown to reduce communication time by $5 - 10\%$ [2], however it would require a significant refactoring of the code that was out of the scope of the time allocated for MPI improvements for this project. A simpler approach would be to try to combine as many of the transfers as possible into single messages. This could be done by using `MPI_Pack/MPI_Unpack`. The existing implementation is detailed below in pseudocode and focusing only on the x direction for simplicity

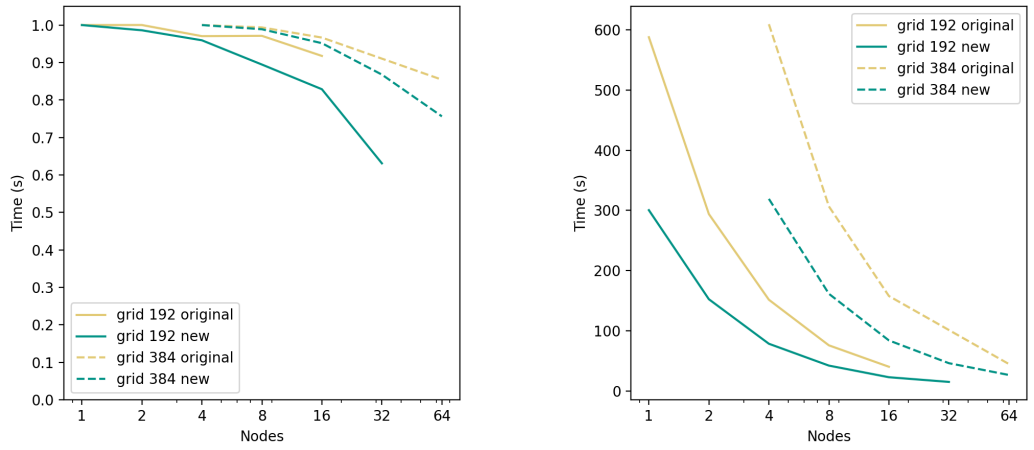
```
// Assembles data from vector into Xright, etc
makeCenterFace(vector, Xright, Xleft, Yright, Yleft, Zright, Zleft);
// Halo swap in X-direction
communicateGhostFace(getXright_neighbor(), getXleft_neighbor(), Xright, Xleft);
// Pack data into vector
parseFace(vector, Xright, Xleft, Yright, Yleft, Zright, Zleft);
// Pack data to communicate X edges. These get swapped in Z direction
makeNodeEdgeX(Yleft, Yright, YrightZrightEdge, YleftZleftEdge,
              YleftZrightEdge, YrightZleftEdge);
// Halo swap in Z
communicateGhostFace(getZright_neighbor(), getZleft_neighbor(),
                     YleftZrightEdge, YleftZleftEdge);
communicateGhostFace(getZright_neighbor(), getZleft_neighbor(),
                     YrightZrightEdge, YrightZleftEdge);
// Pack data back into vector
parseEdgeX(vector, YrightZrightEdge, YleftZleftEdge, YleftZrightEdge, YrightZleftEdge);
// Assemble data to exchange corners
makeNodeCorner(...)
// Communicate only in the X direction (no analogues in Y or Z, names of
// arrays containing data ommitted for brevity)
communicateGhostFace(getXright_neighbor(), vct->getXleft_neighbor(), ... );
communicateGhostFace(getXright_neighbor(), vct->getXleft_neighbor(), ... );
communicateGhostFace(getXright_neighbor(), vct->getXleft_neighbor(), ... );
communicateGhostFace(getXright_neighbor(), vct->getXleft_neighbor(), ... );
// Pack data back into vector
parseCorner(vector, ...)
```

It turns out the communication of the edges depends on having communicated the faces first (and likewise the corners depend on the edges) so it is not possible (at least in the current communication setup) to collapse this all into three sets of messages, one in each of the x, y, and z directions. However by packing the messages when communicating the edges and the corners, cuts the number of point-to-point calls from 13 to 7.

To further increase the amount of data transferred per message instead of communicating each of the fields individually, they can be communicated simultaneously. This is achieved by allocating larger arrays that are being communicated and copying the data in with appropriate offsets. This is key as it can further decrease the number of point-to-point calls by a factor corresponding to the number of fields that can be communicated simultaneously. Doing so in as many places as possible resulted in a significant reduction, as reported by MAP from 40.4% of the compute time being spent in MPI to 27.4%, with an overall reduction in runtime, again as reported by MAP, from 133s to 114s. Unfortunately for unknown reasons the actual runtime when running without a profiler was unchanged,

and there was insufficient time available on the project to pursue further investigation of this discrepancy. Further investigation, perhaps with other profilers, would be recommended to investigate the reason for this lack of improvement. A general restructuring of the communication pattern is also suggested as even with the implemented improvements communication is still dominated by small messages. Something such as implementing MPI-RMA for the halo exchanges might be worthwhile.

Strong scaling tests were performed for both before and after all the eCSE changes described in this report (see Figure 1. IO was turned off, and two problem sizes with grids of 192^2 and 384^2 were performed with 9.66×10^9 particles. Note that the 384^2 problem starts with 4 nodes as that is the minimum number required to fit in memory. All simulations were performed for 60 timesteps. The timings after the changes start out slightly less than two times faster than before, although the distance between the lines decreases slightly and the efficiency after the changes is not as high (although still above 70% for 64 nodes). The reason for this is that most of the gain in performance is due to improving memory accesses, and the MPI improvements as mentioned above do not manifest in improved performance outside profiling runs. This means communication takes up a greater proportion of the execution time especially at larger core counts resulting in worse scaling performance.



(a) Parallel efficiency

(b) Simulation time

Figure 1: Efficiency and time to solution for strong scaling runs of 384^2 and 192^2 grids with 9.66×10^9 particles. Gold lines indicate results before any of the changes introduced in this eCSE, green lines indicate results after changes. Solid lines are the smaller problem size while dashed are the larger.

3 IO

The final workpackage involved refactoring the IO to move away from the no longer supported H5Hut library. It was decided to follow the OpenPMD standard [3], designed to store mesh and particle data. An OpenPMD-conformant IO may be implemented by hand, or using the OpenPMD-api library [4]. OpenPMD-api can use both ADIOS2 and HDF5 backends, which allows reuse of post-processing scripts with minimal changes if using HDF5. The use of ADIOS2 is recommended for performance reasons as there are

more tuning options available, which will be discussed below.

The implementation is reasonably straightforward. OpenPMD provides a hierarchy where the data is contained within a `series` which consists of `iterations` (although one can also have one series per iteration if one wants to write a file per timestep). Each iteration contains a `Mesh` structure and a `Particle Species` structure for each particle species. The particle species contains a `Species Record` for each physical quantity (e.g. position or momentum). Finally the spatial components of meshes or species records are contained in `Record Components`.

To emulate this hierarchy the top-level writing function contains calls to functions responsible for writing particles and the E and B fields. Each of the fields is written one component at a time (i.e. E_x, E_y, E_z, etc) by a `write_fields` function. Likewise, particle data is written on a component by component basis including their positions, velocities, id, etc. values common to all the particles within a species (e.g. charge, or charge to mass ratio) are written as single scalars. Prior to initiating the writing, offsets are computed into the writing arrays so that the data is placed in the correct order by rank. These are done separately for the mesh and particles and stored in the `openpmd_io` class. This is done based purely on the number of ranks, mesh size and particle distribution and can be reused for restarts. Restart functionality has also been implemented using a derived class from `openpmd_io` with a little bit of additional/differing functionality to account for needing to write a different file, read data, and set the relevant values for the fields and particles so that the code can restart the simulation.

OpenPMD-api provides a number of parameters that can be tweeked to obtain improved performance, however these are mostly applicable to the ADIOS2 backend. To compare performance of the existing HDF5 IO implementation to the new openPMD one we ran a simulation with 50 snapshots written on a 192^2 grid with 40 particles per cell and 4 species on 1 node. In this case the number ADIOS2 aggregators and subfiles was set to the number of tasks. These can be set with the environment variables `OPENPMD_ADIOS2_BP5_NumAgg` and `OPENPMD_ADIOS2_BP5_NumSubFiles`. In practice for a large scale run it is recommended to reduce the number of aggregators to one per node. We didn't have time to test what the changeover scale would be. Compared to the previous HDF5 IO implementation we obtained a speedup of about 25% with the total IO time going from 75s to 56s.

4 General improvements

Like most PIC codes, ECsim is severely memory bound. The largest proportion of the time is taken in `GatherMoments`, which computes grid values based on the particle distribution. This involves incrementing arrays storing those quantities for each particle in the simulation. Of these the most time consuming is updating the mass matrix, which comes out to be about 50% of the runtime for a 16 node run for a 768^2 grid with 9.66×10^9 particles. This is a relatively high proportion of the time as the number of particles is quite large, and the actual time spent on updating the mass matrix will vary based on problem size, however this is always the dominant part of the calculation for reasonably sized problems. This update happens in `addMass`, and involves updating 9 4D arrays with 9 precomputed values. To improve performance These 9 arrays were assembled into a single 5D array with the updates running over the last index. Corresponding accessor functions were also provided in order to access the correct subarrays throughout the code. This reduced the amount of time spent updating the mass matrix to 20% of the runtime. Along with a few similar optimisations elsewhere in the code, the total time for this run

was reduced from 105s to 52s. Other places where memory accesses may be improved include setting particle communication buffers to default values prior to populating them. This comprises about 17% of the time in the improved version of the code for this run.

Acknowledgements: *This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk)[1].*

References

[1] George Beckett, Josephine Beech-Brandt, Kieran Leach, Zöe Payne, Alan Simpson, Lorna Smith, Andy Turner, and Anne Whiting. Archer2 service description, December 2024.

[2] Nick Brown, Michael Bareford, and Michèle Weiland. Leveraging mpi rma to optimize halo-swapping communications in monc on cray machines. *Concurrency and Computation: Practice and Experience*, 31(16), September 2018.

[3] Axel Huebl, Rémi Lehe, Jean-Luc Vay, David P. Grote, Ivo Sbalzarini, Stephan Kuschel, David Sagan, Frédéric Pérez, Fabian Koller, and Michael Bussmann. openPMD 1.1.0: Base paths for mesh- and particle- only files and updated attributes, February 2018.

[4] Axel Huebl, Franz Poeschel, Fabian Koller, and Junmin Gu. C++ & python api for scientific i/o with openpmd, November 2021.