minEPOCH3D Performance and Load Balancing on Cray XC30

Michael Bareford
Edinburgh Parallel Computer Centre
University of Edinburgh
Edinburgh, UK
Email: michael.bareford@epcc.ed.ac.uk

Abstract

EPOCH is a mature laser-plasma MPI simulation code that has a large international user base. The code uses particle-in-cell techniques to move Monte-Carlo sampled particles through a fixed grid. Thus, the core scheme is dominated by particle *pushes* and particle to field and field to particle interpolations. This report investigates the performance improvements that can be gained by changing how particles are stored in memory. Currently particles are stored in linked lists (LL), however, it is expected that storing particles within arrays — either as arrays of particle structures (AoS) or as separate arrays of particle properties (SoA) — will provide better performance. We also investigate the benefit of sorting the particles within the array. Following on from this work, we explore how the splitting of the particle push loop in order to exploit Intel vectorisation optimisations can give further reductions in runtime.

This report also covers a new scheme for balancing the workload within EPOCH. The default balancing splits the domain such that each MPI process is assigned an identically-sized section of grid space; this approach, although suitable for simulations that feature a uniform particle distribution, will not perform well should particle concentrations change markedly within the domain. For this reason, we have implemented and tested an alternative scheme that balances the number of particles per process rather than the sub-grid volume.

SoA with particle sorting or with Intel vectorisation runs faster than linked list, with the speed-up achieved falling between 1.03 and 1.44, depending on compiler and grid resolution. Typically, smaller speed-ups are seen for high grid resolutions. Although, SoA with particle balancing showed strong performance gains for the higher of the two grid resolutions tested: speed-up improved as the number of compute nodes was increased, e.g., $2.43\times$LL (4 nodes) and $3.58\times$LL (8 nodes).

Keywords

Cray XC30; MPI; PIC code, AoS, SoA, Intel Vectorisation, Hilbert space-filling curve

I. Introduction

This report covers a series of performance-improving enhancements that were made to a special *barebones* version of the EPOCH3D code, and subsequently tested on the ARCHER supercomputer [1], a Cray XC platform. The release version of EPOCH3D [2] is capable of simulating a variety of physical processes, such as QED physics and particle ionisation. It was decided to strip these features from EPOCH3D v4.4.1 and thereby create a new version, called minEPOCH, that could be used to test which enhancements provided the most benefit. Furthermore, a particular test case was built into minEPOCH in order to provide a common benchmark. The test case features two species of uniformly-distributed particles, initialised with opposing directions of motion. One species of around one million particles is initialised with a leftward (negative x) momentum and an identical number of particles, forming the second group, are given a rightward momentum. Note, each particle in fact represents a large number of real particles (electrons in this case) and so a spline function is used that gives the actual particle density over the volume occupied by the *simulation* particle. The simulation domain was a simple cube of size 5×10^5 consisting of $64 \times 64 \times 64$ cells — periodic boundary conditions prevented particle loss.

The minEPOCH code itself is held within a GitLab repository [3] hosted at the University of Warwick, where the master branch, denoted by `LL`, is the original minEPOCH that stores particles within linked lists. Within this repository, there are separate branches for each type of enhancement, which are named as follows, `AoS` (Array of Structures), `SoA` (Structure of Arrays), `AoSoA`, `SoA_vec` and `SoA_sfc`. The last two are additions to the `SoA` branch: `SoA_vec` uses a vectorised particle push loop and `SoA_sfc` incorporates particle load balancing through the use of a Hilbert space-filling curve. If you wish to create an account on the University of Warwick GitLab Server, please contact Keith Bennett on `k.bennett@warwick.ac.uk`.

Unless stated otherwise, all tests were conducted on a single ARCHER compute node containing 24 CPUs or cores. The simulation time was set to $0.07\,\mathrm{s}$, allowing the `LL` branch of the minEPOCH code to achieve a runtime of approximately 20 minutes over 4890 time steps. Each test was repeated three times; a speed-up value was then calculated by taking the average run time and comparing it with the average achieved by the `LL` branch. Lastly, all tests, unless otherwise stated, ran with Cray-compiled (v8.3.7) executables of the minEPOCH branches.

II. Particle Data Structures

The linked list branch of minEPOCH defines the particle structure as follows.

Listing 1. Linked List Code Definition

```
TYPE particle
  TYPE(particle), POINTER :: next, prev
  REAL, DIMENSION(3) :: pos, mom
  REAL :: mass, weight, charge
END TYPE particle
```

A particle is described by five properties, position, momentum, mass, weight and charge, with the weight being the number of actual particles represented by the simulation particle.

The two pointer attributes (`next` and `prev`) can be dropped when using an array-based data structure, as is the case for the AoS branch.

Listing 2. AoS Code Definition

```
TYPE vector
  REAL :: x, y, z
END TYPE vector

TYPE particle
  TYPE(vector) :: pos, mom
  REAL :: mass, weight, charge
  INTEGER :: cell
END TYPE particle

TYPE particle_data
  INTEGER :: size, count
  TYPE(particle), DIMENSION(:), POINTER :: part_list
END TYPE particle_data
```

The new `cell` attribute within the `particle` structure is an index value that indicates the grid cell location of the particle: thus, for a 3D grid, `cell` = $i + (j-1)n_x + (k-1)n_x n_y$. The main reason for this attribute is to allow the sorting of particles according to spatial proximity. The array itself is pointed to by the `part_list` attribute of the `particle_data` structure. When the particle array is created at runtime it is possible to specify a surplus capacity. Two global constants were added for this purpose: `partlist_size_multiplier` is multiplied with the requested particle count in order to calculate the actual array size, and `partlist_size_min` defines the minimum size for all particle arrays. Each MPI process maintains such an array for each particle species, updating those arrays whenever particles enter or leave the sub-grid assigned to the process. Particle removal can be performed by either zeroing the cell attribute of the element within the array that held the particle, or by shifting down by one all the particles stored after the array position of the departed particle. A global boolean flag, `c_part_shift`, is set to true if the second method is desired. Particle addition involves either finding the first particle element with `cell` = 0, which may exist at any array index if `c_part_shift` is false; otherwise, the new particle is simply inserted at an index one greater than the current particle count. The particle array can be dynamically extended should the particle count be equal to the array size.

The features described in the previous paragraph also apply to the `SoA` branch, where each particle property is a separate array held within a single data structure.

Listing 3. SoA Code Definition

```
TYPE particle_data
  INTEGER :: size, count
  TYPE(vector), DIMENSION(:), POINTER :: pos, mom
  REAL, DIMENSION(:), POINTER :: mass, weight, charge
  INTEGER, DIMENSION(:), POINTER :: cell
END TYPE particle_data
```

The sorting of particle arrays (by grid cell index) can be controlled via the setting of a global constant called `sort_partlist_stride`, where the term *stride* refers to the number of time steps that are required to elapse before the particle arrays are resorted. The particles are never sorted if the stride is set to zero. A third data structure, AoSoA, can be implemented such that it is capable of sorting the particles at every time step.

Listing 4. AoSoA Code Definition

```
TYPE particle_species
  TYPE(particle_list), DIMENSION(:), POINTER :: part_list
  ...
END TYPE particle_species

TYPE particle_list
  TYPE(particle_data), POINTER :: part_data
END TYPE particle_list

TYPE particle_data
  INTEGER :: size, count
  TYPE(vector), DIMENSION(:), POINTER :: pos, mom
  REAL, DIMENSION(:), POINTER :: mass, weight, charge
  LOGICAL, DIMENSION(:), POINTER :: live
END TYPE particle_data
```

For each particle species, an array of pointers to SoAs is maintained, where the index to the `part_list` array now plays the same role as the `cell` attribute from before. The size of the `part_list` array is fixed: there is one element for each cell in the sub-grid and the element order implicitly sorts the particles. A `live` attribute is now required in what used to be the SoA data type (`particle_data`) since if `c_part_shift` is false we need to indicate which particles have departed.

III. Linked Lists vs Array Structures

Figure 1 shows that `SoA` gives the best performance. The relative speed-up compared to linked list is $1.22\times$LL, which is slightly greater than that achieved by using `AoS` ($1.19\times$LL). Turning off particle shifting improved the performance further:

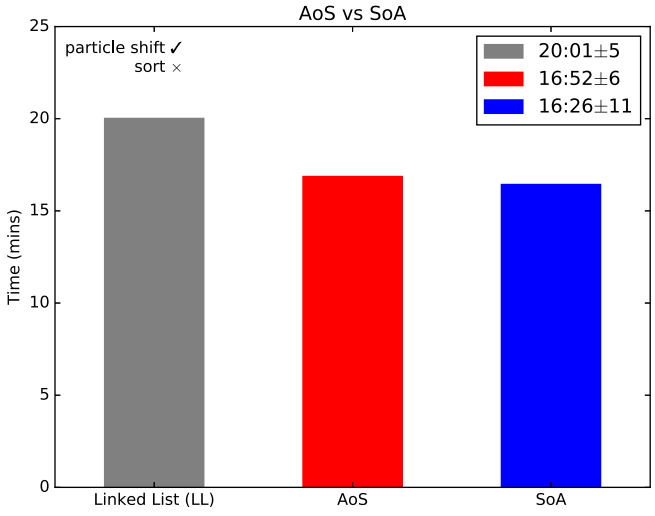

Figure 1. Runtimes for three branches of the Cray-compiled minEPOCH application, `LL`, `AoS` and `SoA`. Each runtime shown is the average of three runs and is displayed in the legend according to the following format, $mm:ss \pm ss$. Particle shifting was turned on and particle sorting was turned off.

speed-ups were increased to $1.22\times$LL for `AoS` and to $1.27\times$LL for `SoA`.

As mentioned previously, it is also possible to sort the particles such that adjacent particle array elements are either located within the same grid cell or within neighbouring cells. Reordering the particles in this way leads to further improvement:

the fastest runtimes were achieved by applying the quick sort algorithm to all particle arrays every ten time steps (i.e., $\mathrm{sort\_partlist\_stride} = 10$). Figure 2 compares the speed-up due to particle reordering across four different data structures.

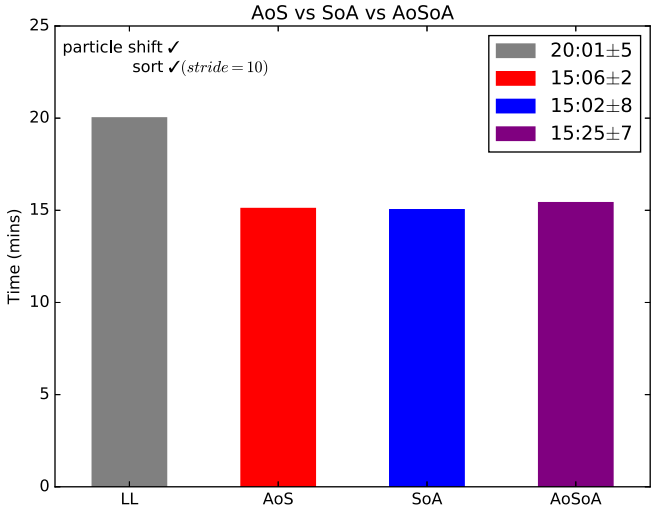

Figure 2. Runtimes for four branches of the Cray-compiled minEPOCH application, `LL`, `AoS`, `SoA` and `AoSoA`. Particle shifting was turned on and particle sorting was also turned on for `AoS` and `SoA`: a quick sort was applied over all particle arrays every ten time steps.

Note, a sorting algorithm is not used for `AoSoA`, since the data structure itself ensures the particles are always sorted. The difference in performance between `AoS` and `SoA` is now so slight (`SoA` is 4 seconds faster) that both data structures have the same speed-up compared to the linked list branch, $1.33\times$LL. `AoSoA` yielded a speed-up of $1.3\times$LL. When particle reordering is applied, turning on particle shifting *worsens* performance: the speed-up drops to 1.24 for `SoA` and 1.25 for `AoSoA`, and for `AoS`, the speed-up disappears altogether, i.e., `AoS` runtimes are slower than that achieved when using linked list.

The ordered particle test runs (with particle shifting turned off) were repeated for higher node counts of four and eight. The simulation time was increased such that the runtime for the linked list branch was maintained at around twenty minutes regardless of node count (e.g., $0.25\,\mathrm{s}$ for four nodes and $0.48\,\mathrm{s}$ for eight). The relative performance of the different branches was unchanged although the speed-ups did increase: for example, SoA achieved $1.45 \times$ LL on four nodes and 1.4 on eight.

IV. Cray vs Intel

The runtimes generated by the Cray-compiled (v8.3.7) minEPOCH executables were compared with the runtimes achieved by the Intel compiler (v15.0.2.164), see Figure 3. For Cray-compiled code, moving to `SoA` with particle sorting gives a

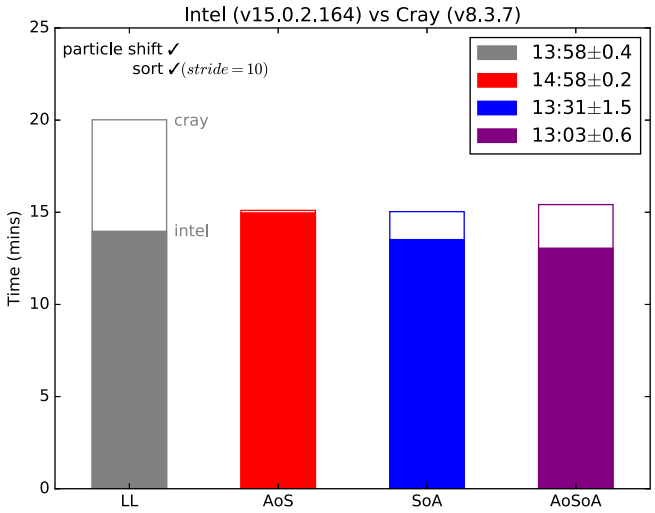

Figure 3. Runtimes for four branches of the Intel-compiled minEPOCH application, `LL`, `AoS`, `SoA` and `AoSoA`. Particle shifting was turned on and particle sorting was turned on for `AoS` and `SoA`. The longer Cray runtimes are indicated by the white bar segments.

$1.33\times$LL speed-up; however, for Intel-compiled code, this speed-up is a less impressive $1.03\times$LL (AoSoA has a slightly higher speed-up of $1.07\times$LL). Furthermore, the improved performance gained by simply moving to an Intel compiler also applies to the linked list branch, which is now *faster* than `AoS`.

These results are sensitive to the number of cells used to divide the grid (i.e., grid resolution): for example, using a $50\times50\times50$ grid with a simulation time of $0.15\,\mathrm{s}$, the Intel-compiled code now shows a speed-up of $1.17\times$LL for sorted `SoA` (`AoS` and `AoSoA` exhibit smaller speed-ups of $1.13\times$LL).

V. Intel Vectorisation

Continuing with the Intel compiler, it is possible to alter the structure of the code within the particle push loop in order to maximise the opportunities for vectorisation. We have incorporated the code restructuring discussed by Bird 2015 [4], which involves first splitting the particle push loop into three. The first loop calculates the particle positions at the half time step, after which, the particles are sorted according to the global cell index using a bin sort. The second loop updates particle momenta and calculates the particle positions at the full time step. (At this point, Bird et al. re-sort the particles, but this time, according to the cell index occupied at the *next* half time step - we found that the addition of this step worsened performance, hence it will not be discussed in the main text). Finally, the third loop calculates the currents - this is actually done within a three-deep nested loop, one for each dimension, see the following code fragment.

Listing 5. Current Calculation

```
DO iz = zmin, zmax
  DO iy = ymin, ymax
!DIR$ VECTOR ALIGNED
!DIR$ SIMD PRIVATE(...)
    DO ix = xmin, xmax
      jx(...) += jxh(...)
      jy(...) += jyh(...)
      jz(...) += jzh(...)
    ENDO
  ENDDO
ENDDO
```

In addition, it was necessary to change the set of compile options (e.g., `-g -traceback -heap-arrays 64 -O2 -xHost -qopt-report5 -align array64byte`) such that vectorisation directives were activated and also reported. The benefit of vectorisation depends on where the directives are placed within the current-calculating loops: in the code fragment shown above, the directives are placed immediately before the start of the x loop.

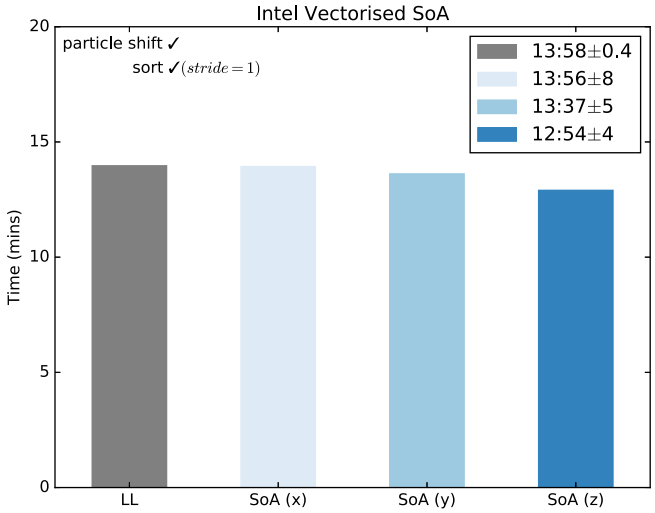

Figure 4. Runtimes for the `SoA_vec` branch of the minEPOCH application distinguished by the placing of vectorisation directives. Particle shifting was turned on and particle sorting was applied within the first loop of each particle push.

Figure 4 shows that placing the directives before the z loop gave the best speed-up ($1.08\times$LL) out of the three possibilities. Again, the result improves for a reduced grid resolution of 50^3, speed-up rises to $1.25\times$LL. Neither of these results is as high as the one reported by Bird et al. [4] which was $1.55\times$LL, albeit for a different code base called miniEPOCH*2D*.

VI. Particle Load Balancing

All variants of minEPOCH discussed so far divide the domain such that each MPI process (or rank) handles a similar-sized portion of the global grid. Instead, the workload could be divided such that each rank handles approximately the same number of particles. Hence, the MPI processes may be assigned grid spaces that vary in volume and are not necessarily cuboid. In order to divide the grid in this manner, we require a curve that visits every grid cell: furthermore, any two adjacent points on the curve must always refer to neighbouring grid cells. The Hilbert space-filling curve fulfils these requirements, it allows us to represent a multi-dimensional space as a one-dimensional function (Figure 5). Plotting the space-filling curve

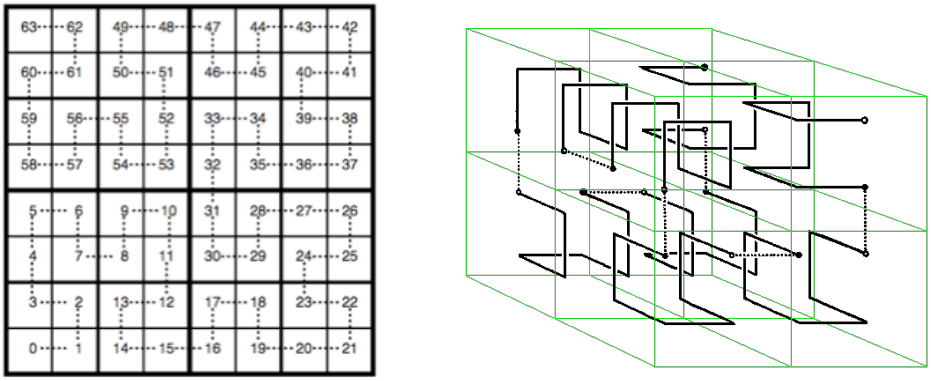

Figure 5. Illustrations of Hilbert space-filling curves for 2d (left) and 3d (right) spaces. The left image is credited to Massimo Cafaro, University of Salento, and the right image is credited to William Gilbert, University of Waterloo.

can be done recursively and the sequence of points can then be recorded in the following data structure.

Listing 6. Space Filling Curve Code Definition

```
TYPE global_sfc
  INTEGER, DIMENSION(ncell) :: icell
  INTEGER, DIMENSION(ncell) :: rank
  INTEGER :: ilocal, ncell_local
END TYPE
```

Every process holds an identical copy of the full space filling curve: the `icell` array maps each element (or point) of the curve to a specific grid cell and the `rank` array records the sections of curve assigned to each MPI process, which changes every time the particle load is rebalanced. The curve segment managed by the local process is defined by the `ilocal` and `ncell_local` fields.

A traversal of the curve allows the particles to be shared evenly, since both the total number of particles and the number of MPI processes are known. The former is divided by the latter to get the number of particles per rank. This value can then compared with the number of particles counted over the first section of the space-filling curve. Finding a match demarcates the sub-grid to be handled by rank zero. Repeating this process for all ranks in turn divides the workload according to particle number.

Figure 6 shows a situation where each rank manages a sub-grid of equal area. This regularity permits a straightforward

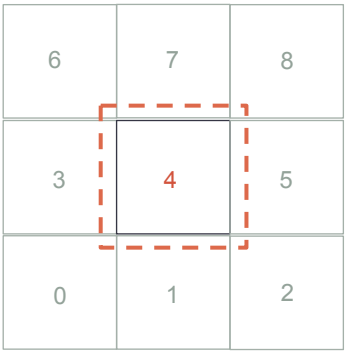

Figure 6. A square sub-grid surrounded by regularly-spaced neighbouring sub-grids. The numbers indicate the ranks of the MPI processes.

scheme for dealing with particles that have crossed sub-grid boundaries. Each MPI process conducts a round of calls to `MPI_SENDRECV` between it and its neighbours: for example, when rank 4 sends particles to rank 5 (its right neighbour), it also receives from its left neighbour (rank 3). In other words, send and receives always involve neighbours from opposing sides. This arrangement breaks down when using the space-filling curve approach: if the particle density rises steeply as one moves across the sub-grid then the rank that manages that grid space will have more neighbours on one side than on the other.

A solution is to consider the fact that every sub-grid is associated with a halo of cells that is known to be under the control of the neighbouring ranks. In fact there are two types of halo or boundary cell that fall within the view of a particular rank: an `external` boundary cell is one that has been assigned to a neighbouring rank, whereas an `internal` boundary cell is controlled by the local rank but falls within the external boundary of a neighbour. These terms are illustrated by Figure 6: the two data structures for external and internal boundary cells both point to an array of cell index lists, each element in the array corresponds to an individual neighbour.

Figure 7. The difference between external and internal boundary cells. Orange indicates the boundary cells for the rank 4 process, black the boundary cells for rank 5.

Thus, the communications between a rank and an arbitrary set of neighbours can proceed like so.

Listing 7. Boundary Cell Communication Loop

```
for each neighbour in sfc_neigh_ext
  MPI_IRECV field data for all external boundary cells

for each neighbour in sfc_neigh_int
  MPI_ISEND field data for all internal boundary cells
```

MPI_WAITALL

Although the example above is concerned with field updates, a similar algorithm can be used for transferring particles. In addition, we could also send first then receive, using either `MPI_ISEND`/`MPI_RECV` or `MPI_ISSEND`/`MPI_RECV`.

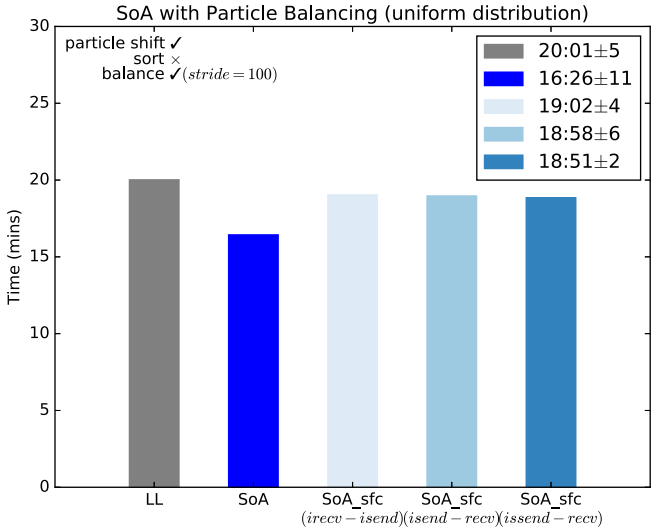

Figure 8. Runtimes for three branches of the Cray-compiled minEPOCH application, `LL`, `SoA` and `SoA_sfc`. The results for the latter branch are split into three, one for each communication scheme. Particle shifting was turned on and particle sorting was turned off. Note, particle balancing only applies to the `SoA_sfc` branches.

Figure 8 compares the times achieved for all three communication schemes. Using a non-blocking synchronous send (`MPI_ISSEND`) followed by a blocking receive (`MPI_RECV`) produces a speed-up of $1.06\times$LL, fractionally faster than using non-synchronous sends. Two constants have been added to the `SoA_sfc` branch for the purpose of specifying which of the

three communication schemes are to be used for particles (`partcomms_default`) and field data (`neighcomms_default`). All three `SoA_sfc` times (Figure 8) are slower than that given for `SoA`: this is because the minEPOCH test case describes a uniform particle distribution (~ 4 particles per cell). We must therefore change this test case to one more suitable for the `SoA_sfc` (`MPI_ISSEND` / `MPI_RECV`) branch.

The minEPOCH test case will still have two species of particles streaming in opposite directions, but this time, the particles will be concentrated at two density peaks, located at $(0.25, 0.5, 0.5)x_{\max}$ for the rightward-propagating species and at$(0.75, 0.5, 0.5)x_{\max}$ for the other species, where $x_{\max} = y_{\max} = z_{\max} = 5 \times 10^5$. The particle density over the entire grid is thus given by the following equations.

$$\rho = \rho_{\max}\left[e^{-(x_l+y_l+z_l)/\rho_0} \; + \; e^{-(x_r+y_r+z_r)/\rho_0}\right], \tag{1}$$

where

$$x_l(x) = \left(x - \frac{1}{4}x_{\max}\right)^2, \quad x_r(x) = \left(x - \frac{3}{4}x_{\max}\right)^2, \tag{2}$$

$$y_{l,r}(y) = \left(y - \frac{1}{2}y_{\max}\right)^2, \quad z_{l,r}(z) = \left(z - \frac{1}{2}z_{\max}\right)^2. \tag{3}$$

The constants, $\rho_{\max}$ and ρ_0 are set such there are $\sim$1.1 million particles in total, and the simulation time is now set to $0.074\,\mathrm{s}$ so as to maintain an execution time of twenty minutes for the linked list branch.

With a non-uniform particle distribution, we now see that `SoA_sfc` is faster than `SoA` and `LL` (Figure 9). The speed-up for `SoA_sfc` is $1.1\times$LL. This modified test case was repeated for node counts of four ($t_{\mathrm{sim}} = 0.165\,\mathrm{s}$) and eight nodes ($0.19\,\mathrm{s}$) — the associated simulation times that keep the `LL` runtime at approximately 20 minutes are given in brackets. At larger node counts the performance gain produced by `SoA_sfc` is more impressive, see Figure 10; the speed-ups are

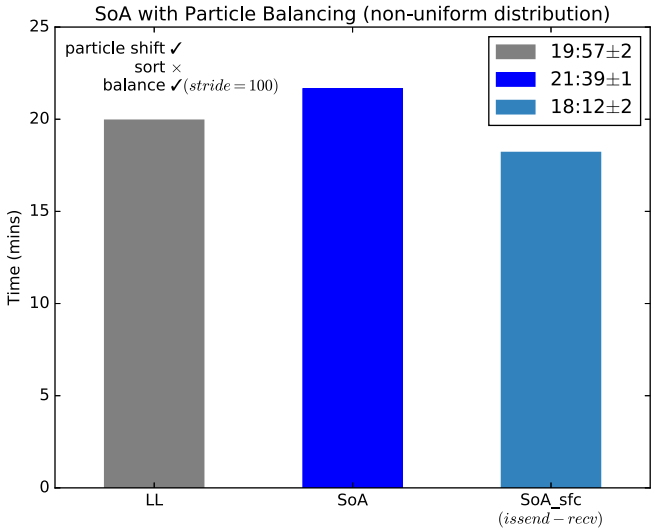

Figure 9. Runtimes for three branches of the Cray-compiled minEPOCH application, `LL`, `SoA` and `SoA_sfc`.

$2.43\times$LL (4 nodes) and $3.58\times$LL (8 nodes).

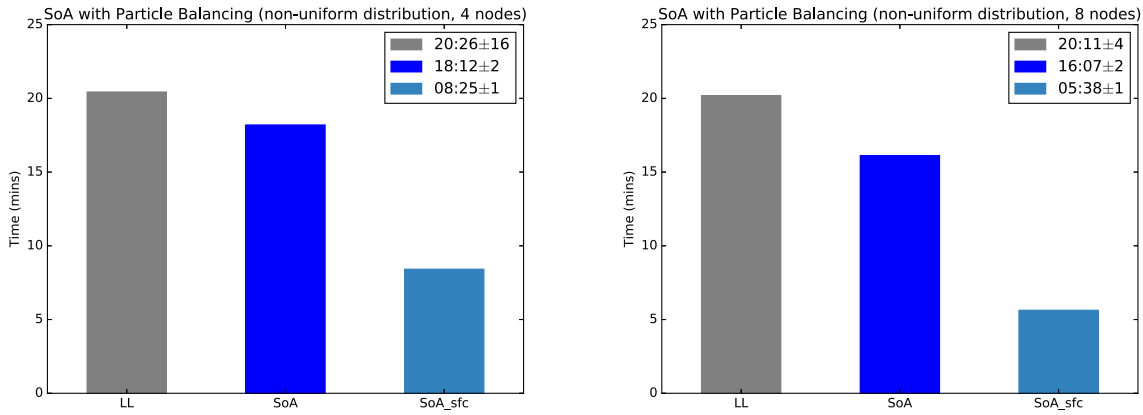

Figure 10. A repeat of Figure 9 for four nodes (left) and for eight nodes (right).

VII. Conclusions and Further Work

`SoA` with particle sorting or with Intel vectorisation runs faster than linked list, although, the precise speed-up depends on compiler and grid resolution (Table I). However, SoA with particle balancing (`SoA_sfc`) shows strong performance gains

minEPOCH3D SoA	50×50×50 (0.15 s, 8190 ts)	64×64×64 (0.07 s, 4890 ts)
Sorted (Cray)	1.44	1.33
Sorted (Intel)	1.17	1.03
Vectorised (Intel)	1.25	1.08

Table I
SoA SPEED-UPS COMPARED TO LINKED LIST (×LL) FOR THE CRAY AND INTEL COMPILER AT TWO GRID RESOLUTIONS (TS = TIME STEP).

at $64\times64\times64$ (compared to linked list) for a non-uniform particle distribution: speed-up improves as node count increases, e.g., $2.43\times$LL (4 nodes), $3.58\times$LL (8 nodes).

As regards further work, the `SoA_sfc` branch should be retested with particle sorting turned on; this should improve performance once more. There are also some aspects of the particle balancing that could be enhanced: for example, at present, only periodic boundary conditions have been implemented for the field arrays. Secondly, the fact that every MPI process has a copy of the entire space-filling curve, limits the grid size to 512^3 for an ARCHER node of 24 cores and 64 GB of RAM.

The ability to represent larger grids would require the use of MPI IO to write the `global_sfc` data structure to file. The current `SoA_sfc` implementation has each MPI process performing the same recursive algorithm to plot the entire space-filling curve (see the `plot_sfc` subroutine in `balance_sfc.F90`). The MPI processes then independently identify those sections of the curve under their control. A subsequent `MPI_AllReduce` results in all processes having a record of the initial rank ownership for the full curve. If MPI IO is to be used, just one rank should be responsible for plotting the Hilbert curve, writing the coordinates to a file as it does so. This rank should also be able to specify the initial rank ownership, assuming that the grid space has been evenly partitioned before any particle balancing has been applied. So far, all IO relating to the Hilbert curve is a once only cost, incurred before the first time step. During the simulation, a set of file IO operations, from all processes, will be needed to re-specify the rank ownership whenever the workload is re-balanced; some parallelism should still be possible if the process IO is done a block at a time, and if the options for Lustre file striping [5] have been set appropriately.

Finally, the minEPOCH code uses three-dimensional arrays to record how the electromagnetic field varies over the sub-grid volume, but the segment of space-filling curve that encloses a sufficient number of particles for a single MPI process is unlikely to map to a sub-grid that has a regular shape. Instead, the sub-grid for a single process is represented as the smallest regular shape that encloses the section of space-filling curve, see red rectangle in Figure 11 (right). This enclosing shape is expanded to include the boundary cells (Figure 12). Then a mask is added that records which cells are actually controlled

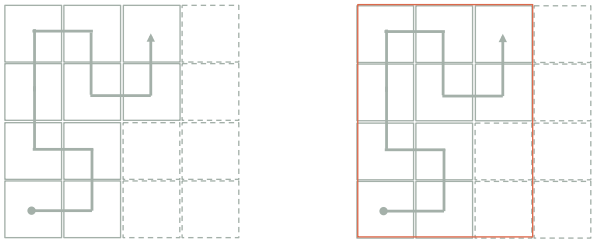

Figure 11. A space-filling curve defines a 2D volume that is not rectangular (left), however, the associated field arrays do encompass a rectangular area (right).

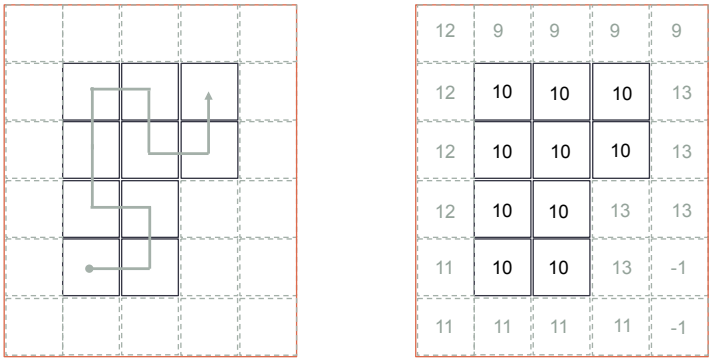

Figure 12. The sub-grid field arrays are enlarged to include the boundary cells (left) and a mask is applied that indicates which ranks control which cells (right).

by the MPI process (e.g., rank 10) and which processes are in charge of the boundary cells. The value of -1 is reserved for those cells that fall outside the true boundary. This method is convenient but creates some wastage: on average, $\sim$22% of the sub-grid field array elements are not used (i.e., have been assigned a -1 value) for the uniform particle distribution test case. This figure falls to $\sim$10% for the non-uniform distribution. One way to eliminate negative mask cell values would be to divide an irregular-shaped sub-grid into a set of differently sized sub-sub-grid volumes (Figure 13, left): alas, each of these smaller sub-grid volumes would need to maintain a collection of boundary cells in order to be processed by the particle push routine. Hence, overall, more cells would be required compared to the simpler implementation.

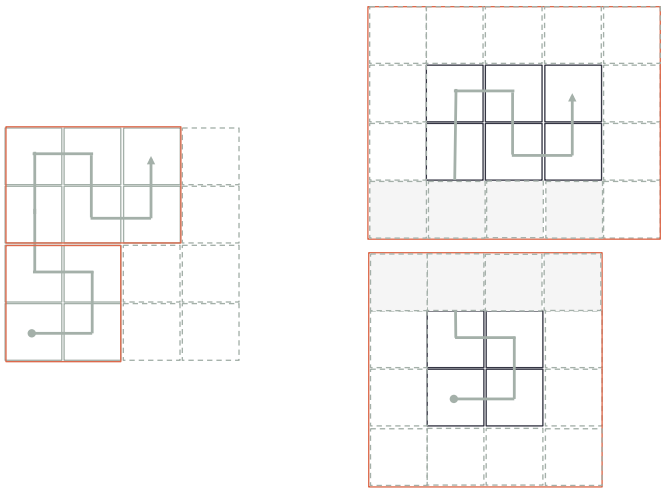

Figure 13. The irregular sub-grid is split into two regularly-sized smaller sub-grids (left), which are then enlarged to include the necessary boundary cells (right). Duplicated boundary cells are shaded grey.

ACKNOWLEDGMENT

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

REFERENCES

[1] EPCC. (2016) Archer user guide. Web site. [Online]. Available: http://www.archer.ac.uk/documentation/user-guide/

[2] T. D. Arber, K. Bennett, C. S. Brady, A. Lawrence-Douglas, M. G. Ramsay, N. J. Sircombe, P. Gillies, R. G. Evans, H. Schmitz, A. R. Bell, and C. P. Rodgers, "Contemporary particle-in-cell approach to laser-plasma modelling," *Plasma Physics and Controlled Fusion*, vol. 57, no. 11, pp. 1–26, November 2015.

[3] M. R. Bareford. (2016) minepoch. GitLab repository. [Online]. Available: https://cfsa-pmw.warwick.ac.uk

[4] R. F. Bird, P. Gillies, M. R. Bareford, J. A. Herdman, and J. A. Jarvis, "Mini-app driven optimisation of inertial confinement fusion codes," in *2015 IEEE International Conference on Cluster Computing*, September 2015. [Online]. Available: http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=7307681

[5] M. R. Bareford. (2016) Lustre and i/o tuning. You Tube. [Online]. Available: https://www.youtube.com/watch?v=xzMxCHmtJGo&feature=youtu.be