minEPOCH3D Performance and Load Balancing on Cray XC30

Michael Bareford, ARCHER CSE Team
michael.bareford@epcc.ed.ac.uk

Outline

1. Introduction (ARCHER and minEPOCH3D)

2. Particle Data Structures

3. Particle Push Vectorisation

4. Load Balancing with respect to Particle Counts

5. Summary and Conclusions

Introducing ARCHER

Advanced Research Computing High End Resource

EPSRC

NERC SCIENCE OF THE ENVIRONMENT

CRAY THE SUPERCOMPUTER COMPANY

archer

www.archer.ac.uk

epcc

THE UNIVERSITY OF EDINBURGH

Introducing ARCHER

Cray XC30 MPP, 4920 Compute Nodes

Dual Intel Xeon processors (Ivy Bridge), 24 cores, 64 GB

Dragonfly topology

rank 1: intra-chassis, sixteen 4-node blades (Aries interconnect)
rank 2: intra-group (two cabinets per group)
rank 3: optical, inter-group (13 groups make up ARCHER)

Tests conducted on 2-cabinet *Test Development Server*

Private to EPCC, minimises resource contention.

ARCHER supports three programming environments

Cray (v8.3.7), Intel (v14.0.4) and gnu (v4.9.2) running on CLE v5.1 OS

minEPOCH3D

Based on EPOCH v4.4.1 and contains built-in two stream test case involving around 2 million evenly-distributed particles.

Domain is cube of side 5×10^5, partitioned over 50×50×50 cells and 2×3×4 cores (one ARCHER node). Periodic BCs.

Simulation runs for 0.15 s, one time step = 18.3 μs.

Total particle and field energies recorded every time step (8190).

Particles stored within linked list.

Two stream test case is two species of 500 000 particles (~4 particles per cell), electrons and positrons, travelling in opposing directions. Particle shape function is bspline3.

Linked List

```
TYPE(particle_species), POINTER :: species_list

TYPE particle_species
  TYPE(particle_species), POINTER :: next, prev
  TYPE(particle_list) :: attached_list
  ...
END TYPE particle_species

TYPE particle_list
  TYPE(particle) :: head
  ...
END TYPE particle_list

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

Array of Structures

```
TYPE particle_data
  INTEGER(i8) :: size, count
  TYPE(particle), DIMENSION(:), POINTER :: part_list
END TYPE particle_data

TYPE particle
  TYPE(vector) :: pos, mom
  REAL(num) :: mass, weight, charge
  LOGICAL :: live
END TYPE particle_list

TYPE vector
  REAL(num) :: x, y, z
END TYPE vector

REAL(num), PARAMETER :: partlist_size_multiplier = 1.0_num
INTEGER(i8), PARAMETER :: partlist_size_min = 100

LOGICAL, PARAMETER :: c_part_shift = .FALSE.
```

Introduced a live field, which indicates if a particular array position is occupied by a real particle. For example, if ith particle leaves subdomain, live(i) is set to false. However, live is only used when c_part_shift=false, otherwise particles to the right of escaped particle are simply shifted one space to the left.

Structure of Arrays

```
TYPE particle_data
  INTEGER(i8) :: size, count
  TYPE(vector), DIMENSION(:), POINTER :: pos, mom
  REAL(num), DIMENSION(:), POINTER :: mass, weight, charge
  LOGICAL, DIMENSION(:), POINTER :: live
END TYPE particle_data
```

`live` array indicates if particular array position is occupied by a particle, only used if `c_part_shift = .false.`

Performance (data structures)

Cray-compiled EPOCH Code

Particle Data Structure	Run Time Average (min:sec)
Linked List	17:28
Array of Structures	14:22
AoS (part shift off)	14:16
Structure of Arrays	14:26
SoA (part shift off)	14:23

~18% improvement for AoS or SoA

<1% increase in performance when particle shift turned off

Particle Reordering

Sort particles such that adjacent particles in lists are either located within same cell or within neighbouring cells.

Use a 1D index to reference each cell in a 3D section of domain.

$$\mathtt{icell} = i + (j-1)n_x + (k-1)n_x n_y$$

How frequently should particle arrays be sorted?

```
INTEGER, PARAMETER :: sort_partlist_stride = 10
```

Every 10 time steps, sort the `particle_data` arrays according to particle location, expressed as a 1D grid cell index.

Particle Reordering (Simple Sort)

```
! determine which cells are occupied and
! calculate 1D cell index for each particle
DO ip = 1, count
  ic = get_cell_index(part_data%pos(ip))
  cell_occupied(ic) = .TRUE.
  part_cell(ip) = ic
ENDDO

! a sorted list is constructed by iterating
! through the cells in order
DO ic = 1, ncell
  IF (cell_occupied(ic)) THEN
    DO ip = 1, count
      IF (ic == part_cell(ip)) THEN
        ! get particle at position ip in part_data
        ! append that particle to part_data_sorted
      ENDIF
    ENDDO
  ENDIF
ENDDO
```

Note, ncell for a given rank may change as a result of load balancing. This naive sorting algorithm is used to understand the performance benefit of sorting the particles compared to that introduced by the sorting algorithm itself, see next slide.

Particle Reordering (Quick Sort)

```
DO ip = 1, count
  ic = get_cell_index(part_data%pos(ip))
  part_cell(ip) = ic
  part_index(ip) = ip
ENDDO

! part_cell is sorted according to ic
! identical rearrangement applied to part_index
CALL qsort_partlist(part_cell, part_index, count2)

DO ip = 1, count
  ip2 = part_index(ip)
  IF (ip2 >= 1 .AND. ip2 <= count) THEN
    ! get particle at position ip2 in part_data
    ! append that particle to part_data_sorted
  ENDIF
ENDDO
```

AoS Performance (sorting techniques)

Cray-compiled EPOCH Code (part shift on)

Stride	Basic	Quick
1	>40:00	13:20
10	25:13	12:16
100	13:31	12:15
1000	13:02	12:53

Quick sort with stride of 10-100 seems to be optimal setting for quick sort.

~60% increase in runtime when particle shift turned off.

SoA Performance (sorting techniques)

Cray-compiled EPOCH Code (part shift on)

Stride	Basic	Quick
1	>40:00	12:57
10	20:16	12:07
100	13:16	12:07
1000	12:52	12:45

Quick sort always completes in a reasonable time and SoA is slightly faster than AoS.

Particle Reordering (AoSoA)

```
! INTEGER, PARAMETER :: sort_partlist_stride = 1

TYPE particle_species
  ! an element for every cell (one rank ~5000 cells)
  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(i8) :: size, count
  TYPE(vector), DIMENSION(:), POINTER :: pos, mom
  REAL(num), DIMENSION(:), POINTER :: mass, weight, charge
  LOGICAL, DIMENSION(:), POINTER :: live
END TYPE particle_data
```

Performance (Cray vs Intel)

Particle Data Structure	Cray	Intel
minEPOCH3D	17:28	12:42
AoS (f)	14:16	12:33
SoA (f)	14:22	12:19
AoS (qs, 100, t)	12:15	11:14
SoA (qs, 10, t)	12:07	10:53
AoSoA (t)	12:33	11:12

The names of the particle data structures have annotations (see brackets).

f and t indicate c_part_shift=false,true;
qs denotes quicksort; the number gives the sorting stride.

For AoSoA, having c_part_shift=false, increases run time by one minute.

Performance (Cray vs Intel)

For Cray-compiled code moving to AoS/SoA gives ~18% decrease in runtime, if particle sorting is also applied then improvement is roughly 30%.

However, for Intel-compiled code, performance improvement is not as impressive. Changing data structures only gives ~2%, although particle sorting increases this figure to 15%.

Sorting reduces the number of cache misses.

Particle Push Vectorisation

Incorporated Bob Bird's vectorisation of 2D particle push within minEPOCH3D SoA version.

Particle push loop is split in three.

Loop 1: calculate particle positions at half time step.

Sort particles according to global cell index – done using bin sort.

Loop 2: update momentum and calculate particle positions at full time step.

Optional, sort particles according to index of cell occupied at next half time step.

Loop 3: calculate currents – done within a three-deep nested loop structure (one for each dimension).

concentrated our vectorisation efforts within the nested loop structure within loop 3

Particle Push Vectorisation

```
!DIR$ VECTOR ALIGNED
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
```

Intel Compiler v15.0.2.164 used to apply vectorisation/data alignment.

Adjust the vectorisation by changing the position of SIMD directive.

Important to pre-calculate `jxh` variables immediately before loop to avoid exaggeration of final energies.

Aligned on 64 byte arrays.

Particle Push Vectorisation

`Particle Push Type`	Runtime	Final Particle Energy	Final Field Energy
SoA	10:53	8.51	0.0445
Split Only	12:54	8.51	0.0445
Data Alignment	10:51	8.51	0.0445
Outer (z) SIMD	10:09	10.097	1.258
Middle (y) SIMD	10:10	10.065	1.253
Inner (x) SIMD	10:17	23.708	1.888

No pre-calculation of `jxh` etc

Particle Push Vectorisation

Particle Push Type	Runtime	Final Particle Energy	Final Field Energy
SoA	10:53	8.51	0.0445
Split Only	12:54	8.51	0.0445
Data Alignment	10:51	8.51	0.0445
Outer (z) SIMD	10:09	8.51	0.0445
Middle (y) SIMD	10:10	8.51	0.0445
Inner (x) SIMD	10:17	8.51	0.0445

Thanks to Adrian Jackson, EPCC

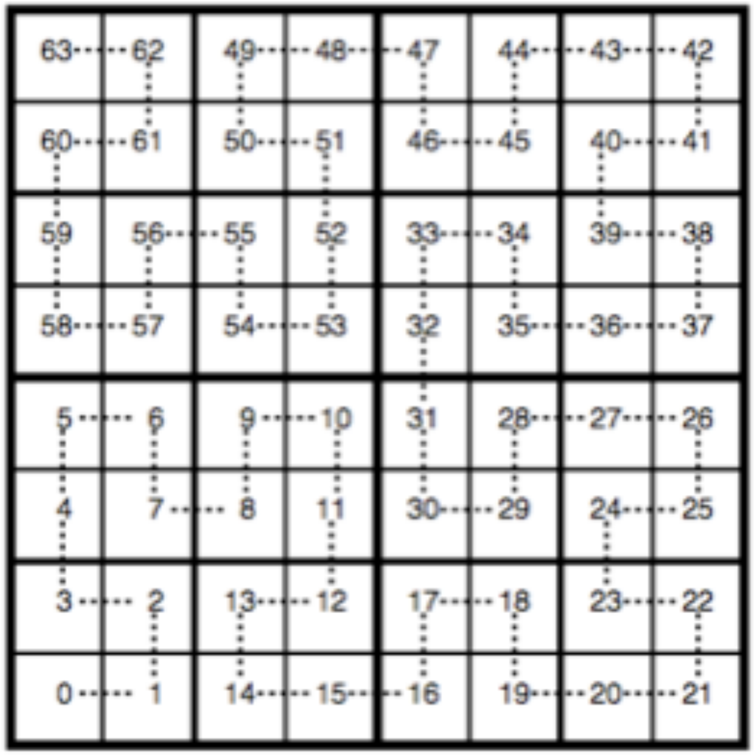
Particle Load Balancing

Currently, the simulation is divided such that each rank handles an similar-sized portion of the global grid.

Instead, we could divide workload such that each rank handles approximately the same number of particles.

Hence, ranks may be assigned grid spaces that **vary in volume** and are **not necessarily cuboid**.

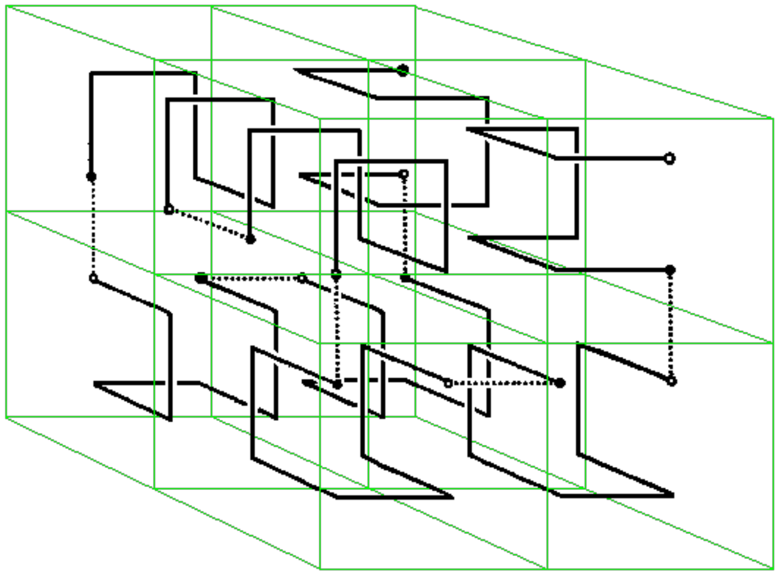
Hilbert Space Filling Curve



Use a 1D data structure to capture the coordinates of every cell within a 3D space.

Space filling curve can be plotted using a recursive algorithm.

Hilbert Space Filling Curve



Every process has an identical copy of the SFC.

Every rank knows which parts of the sfc are managed by other ranks.

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

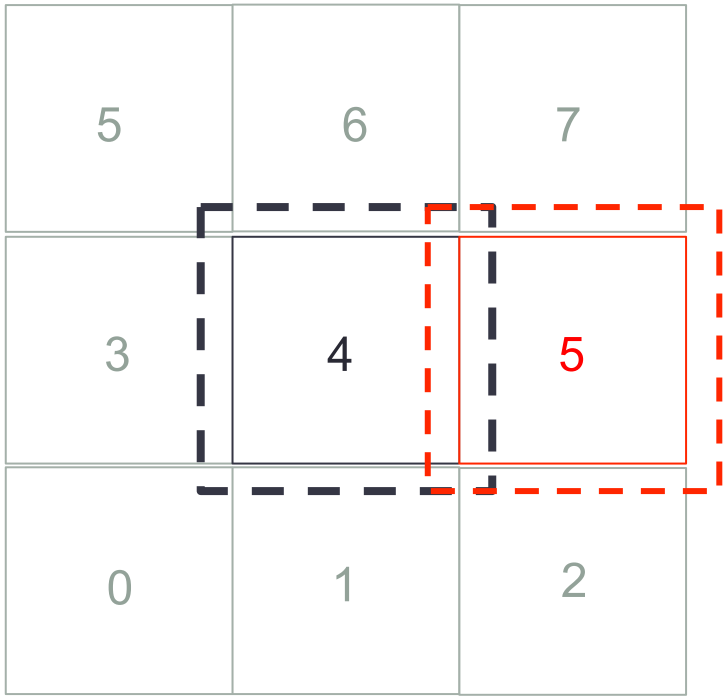
$$\mathtt{icell} = i + (j-1)n_x + (k-1)n_x n_y$$

`total particle number / number of ranks = particle count per rank`

Traverse curve, counting particles until *particle count per rank* is reached. Assign associated grid space to rank 0, repeat for next rank and so on.

The local grid is a 3d array, representing the smallest cuboid that encloses the space traversed by the section of sfc assigned to the rank.

Boundary Communications



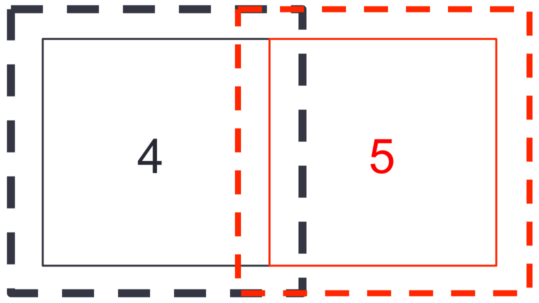
EPOCH uses `MPI_SENDRECV`

➔0, ←7
➔1, ←6
➔2, ←5
...

Cannot rely on this regular arrangement for space filling curve approach.

For example, rank 4 might have more neighbours on the left than on the right.

Boundary Communications



A rank's local grid has two types of boundary cell.

External, those cells that are controlled by neighbouring ranks.

Internal, those cells that are external boundary cells for neighbouring ranks.

```
TYPE cell_data
  INTEGER :: size, count
  INTEGER, DIMENSION(:), POINTER :: icell
END TYPE cell_data

TYPE neighbour_comms
  INTEGER :: size, count
  INTEGER, DIMENSION(:), POINTER :: rank
  TYPE(cell_data), DIMENSION(:), POINTER :: cell_list
END TYPE neighbour_comms

TYPE(neighbour_comms), POINTER :: sfc_neigh_ext, sfc_neigh_int
```

Boundary Communications

For an arbitrary list of neighbours, `MPI_SENDRECV` will deadlock, so instead...

```
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
```

Could also do `MPI_ISEND` then `MPI_RECV` or `MPI_ISSEND` then `MPI_RECV`.

Particle Boundary Communications

After each particle push, every process checks if any particles have moved outside the local grid.

How particle departures are handled depends on whether any particles have travelled beyond the neighbouring local grids, which can be determined from the `hilbert_sfc` structure.

Once each process knows how many particles it will be receiving and from where (i.e., which process rank), it can then proceed with **`MPI_IRECV`** followed by **`MPI_ISEND.`**

Second point is not an issue during particle push iterations due to cfl condition, but particles might well need to be communicated beyond neighbours whenever a rebalancing is performed.

Particle Load Balancing

Setup time is the time between start of application and the start of the particle push loop.

Push time is the time spent in the particle push loop.

Domain now consists of 64×64×64 cells and simulation runs for 0.05 s (3490 timesteps).
c_part_shift = true with **no** particle reordering, still using one node.

For minEPOCH3D SoA this takes 00:01 (setup) and 11:26 (push) using Cray compiler.

MPI Comms	**Initial Balance Only setup time, push time**	**Balance every 100 iterations setup time, push time**
irecv, isend	00:18, 13:03	00:18, 13:03
isend, recv	00:18, 12:58	00:18, 12:57
issend, recv	00:18, 12:56	00:18, 13:02

Compared to SoA, time lost during boundary communications, specifically when each rank updates the field in the external boundary cells of a neighbouring rank.

Non-uniform Particle Distribution

Still two species of particles streaming in opposite directions, but concentrated into two density peaks.

$$x_l(x) = \left(x - \frac{1}{4}x_{\max}\right)^2 \qquad y_l(y) = \left(y - \frac{1}{2}y_{\max}\right)^2 \qquad z_l(z) = \left(z - \frac{1}{2}z_{\max}\right)^2$$

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

Particle number peaks (~600) at the centres of the left and right halves of grid. Around 1.1 million particles in total.

Uniform distribution had 4 particles per cell over the entire grid.

Non-uniform Particle Distribution

For minEPOCH3D SoA this takes 05:34 (setup) and 10:16 (push) using Cray compiler.

MPI Comms	Initial Balance Only setup time, push time	Balance every 100 iterations setup time, push time
isend, recv	06:25, 12:53	06:23, 08:26

With initial particle balance only, simulation finishes in ~ 19 mins.

Push time drops to below 10 min if balance done every 100 iterations.

	Particle and Field Energies
Start	3.42e-2, 0.0
End	2.57e-2, 8.29e-3

Summary and Conclusions

SoA with particle sorting shows performance improvement over linked list – 30% (Cray), 15% (Intel)

Vectorisation can improve performance further (~ 9%), but actually SIMD directives not required – automatic vectorisation that comes with `–O2` is sufficient.

Particle balancing appears to be working...

only periodic bcs supported for fields

testing required for multiple nodes / other test cases