

Scalable Domain-decomposed Monte Carlo Neutral Transport for Nuclear Fusion

Oskar Lappi^{*1}, Huw Leggate², Yannick Marandet³, Jan Åström^{1,4}, Keijo Heljanko^{1,5}, and Dmitriy V. Borodin⁶

¹Department of Computer Science, University of Helsinki, Helsinki, Finland

²School of Physical Sciences, Dublin City University, Glasnevin, Dublin 9, Ireland

³PIIM, Aix-Marseille University, CNRS, Marseille, France

⁴CSC – IT Center for Science Ltd., Espoo, Finland

⁶Forschungszentrum Jülich GmbH, Institute of Fusion Energy and Nuclear Waste Management – Plasma Physics, Jülich, Germany

⁵Helsinki Institute for Information Technology, Helsinki, Finland

Abstract

EIRENE [1] is a Monte Carlo neutral transport solver heavily used in the fusion community. EIRENE does not implement domain decomposition, making it impossible to use for simulations where the grid data does not fit on one compute node (see e.g. [2]). This paper presents a domain-decomposed Monte Carlo (DDMC) algorithm implemented in a new open source Monte Carlo code, Eiron. Two parallel algorithms currently used in EIRENE are also implemented in Eiron, and the three algorithms are compared by running strong scaling tests, with DDMC performing better than the other two algorithms in nearly all cases. On the supercomputer Mahti [3], DDMC strong scaling is superlinear for grids that do not fit into an L3 cache slice (4 MiB). The DDMC algorithm is also scaled up to 16384 cores in weak scaling tests, with a weak scaling efficiency of 45% in a high-collisional (heavier compute load) case, and 26% in a low-collisional (lighter compute load) case. We conclude that implementing this domain decomposition algorithm in EIRENE would improve performance and enable simulations that are currently impossible due to memory constraints.

1 Introduction

Research in magnetized fusion aims at achieving low-carbon energy production by harnessing processes powering stars. This involves using powerful electromagnets to confine a mix of hydrogen isotopes heated at 150 million degrees, a temperature at which the fuel forms a fully ionized plasma. Continuous progress has been achieved by building larger and larger machines [4], now culminating with the construction of ITER, in Cadarache, France.

One of the key challenges of fusion research is power exhaust, because plasma-wall interactions are concentrated on very small areas of the reactor wall (on the order of 0.1% in ITER), risking damage to the wall. In order to spread the energy fluxes over a larger area, the thermal energy

^{*}Corresponding author

in the plasma edge has to be transferred to neutral particles (photons, atoms, molecules). This can be facilitated by the so-called recycling process, by which ions impinging on the wall are neutralized into atoms, possibly recombining into molecules. The resulting atoms or molecules are then either ionized in the plasma, or pumped away. Properly describing their transport is key to understanding the particle balance in the machine.

Numerical simulation plays an important role in this endeavor, both for interpreting current experiments and modeling operational regimes in next-generation devices. The Knudsen number for neutrals has a strong spatial dependence in a given machine and can further vary depending on experimental conditions [5],[6], indicating that a fluid treatment is generally not valid. A kinetic transport model is therefore used in high fidelity calculations, and when neutral-neutral collisions are neglected, the problem is linear. The transport problem is thus similar to that encountered in neutronics or radiation transfer. Because of the complexity of the geometry, of the potentially large number of species to follow, and in order to avoid working in a 6D grid (3D3V), a Monte Carlo approach is used. The EIRENE code [1] is such a solver and is used iteratively with different plasma fluid solvers in a number of code packages, e.g.: SOLPS-ITER [7], EMC3-EIRENE [8], SOLEDGE3X-EIRENE [9].

EIRENE uses a particle-based algorithm, which is parallelized using MPI, OpenMP, or a hybrid of the two. The MPI scheme uses a traditional domain replication algorithm, parallelizing over particles. For 2D grids with a typical 10^5 DoF (Degrees of Freedom), the domain can easily be replicated to all cores on a node using the MPI scheme. However, for 3D applications with more than 10^7 DoF this leads to memory issues at the node level, and in some cases, the full grid does not fit on the memory available to a compute node. Indeed, applications where EIRENE is coupled to 3D turbulence codes such as SOLEDGE3X using a Larmor radius resolution are currently severely limited by this memory constraint [2, sec. 2.3]. The OpenMP scheme also parallelizes over particles, but uses shared memory to avoid replicating the domain within a node, somewhat alleviating these memory issues. However, its scalability is limited [10, sec. 3], and it still fails in cases where the full grid does not fit in the memory of one node. In recent articles, both Quadri et al. [2] and Borodin et al. [10] have come to the conclusion that domain decomposition is the solution to this memory issue.

However, domain decomposition would require extensive changes to EIRENE and the interfaces to coupled plasma codes. Redesigning EIRENE to implement an unverified domain decomposition solution would involve much work and risk. We have therefore started this effort by developing a new code for the specific purpose of studying parallel algorithms that could be implemented in EIRENE (or other Monte Carlo codes).

The new open source code introduced in this work is called Eiron, and it supports three parallel particle Monte Carlo algorithms: traditional domain replication, EIRENE's shared memory algorithm, and domain-decomposed Monte Carlo. The physics model is severely reduced compared to EIRENE, the development focus has been scalability, reliability, and reproducibility.

The rest of the article is laid out as follows: Section 2 is an overview of the particle Monte Carlo method used by EIRENE and Eiron, starting with the transport equations and ending with the basic Monte Carlo particle transport algorithm (Algorithm 1) used by Eiron; Section 3 is a brief overview of previous work on DDMC; Section 4 presents the parallel algorithms in Eiron: a traditional domain replication algorithm (Algorithm 2), a shared memory algorithm (Algorithm 3), and a DDMC algorithm (Algorithm 4); Section 5 presents scalability results; and finally, Section 6 contains our conclusions and discusses future work.

2 Monte Carlo neutral transport

The governing equation for neutral particle collision density is the integral form of the linear time-independent Boltzmann equation, as written in the EIRENE manual [11, eq. 1.d]:

$$\psi(\mathbf{x}) = S(\mathbf{x}) + \int_{\Gamma} \psi(\mathbf{x}') K(\mathbf{x} | \mathbf{x}') d\mathbf{x}' \quad (1)$$

where $\mathbf{x} \in \Gamma$ describes the state of a neutral particle in flight, before collision; $\psi(\mathbf{x})$ is the total collision density at $\mathbf{x}$; $S(\mathbf{x})$ is the collision density at $\mathbf{x}$ due to uncollided particles (from particle sources); and $K(\mathbf{x} | \mathbf{x}')$ is the collision density at $\mathbf{x}$ due to particles from prior collisions at $\mathbf{x}'$. K is integrated over phase space Γ and weighted by the collision densities at the prior states, which gives us a term describing the total collision density due to already collided particles.

By splitting $\mathbf{x}$ into position $\mathbf{r}$ and velocity $\mathbf{v}$, we get the form of the equation given in the textbook by Spanier and Gelbard [12, eq. 2.4.7]:

$$\psi(\mathbf{r}, \mathbf{v}) = S(\mathbf{r}, \mathbf{v}) + \iint \psi(\mathbf{r}', \mathbf{v}') K(\mathbf{r}, \mathbf{v} | \mathbf{r}', \mathbf{v}') d\mathbf{v}' d\mathbf{r}' \quad (2)$$

We can then split the transform kernel K into a product of collision kernel C and transport kernel T :

$$K(\mathbf{r}, \mathbf{v} | \mathbf{r}', \mathbf{v}') = C(\mathbf{v} | \mathbf{r}', \mathbf{v}') T(\mathbf{r} | \mathbf{r}', \mathbf{v}) \quad (3)$$

where C is the proportion of particles colliding at $(\mathbf{r}', \mathbf{v}')$ which exit with velocity $\mathbf{v}$, and T is the proportion of particles starting in state $(\mathbf{r}', \mathbf{v})$ which collide at $\mathbf{r}$.

The transport kernel T involves an integral over the macroscopic cross section Σ_t along a particle's linear trajectory $(\mathbf{r}', \mathbf{v})$, and it is only non-zero if $\mathbf{r}$ is on that trajectory:

$$T(\mathbf{r} | \mathbf{r}', \mathbf{v}) = \begin{cases} T'(\mathbf{r} | \mathbf{r}', \mathbf{v}) & \text{if } \mathbf{r}' + c\mathbf{v} = \mathbf{r}, c > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$$T'(\mathbf{r} | \mathbf{r}', \mathbf{v}) = \Sigma_t(\mathbf{r}, \mathbf{v}) \exp \left[- \int_0^1 \Sigma_t(\mathbf{r}' + \tau(\mathbf{r} - \mathbf{r}'), \mathbf{v}) d\tau \right] \quad (5)$$

The collision kernel C is a sum of collision kernels C_j , belonging to the collision processes j , weighted by probabilities p_j , that a collision at $(\mathbf{r}', \mathbf{v}')$ is of type j :

$$C(\mathbf{v} | \mathbf{r}', \mathbf{v}') = \sum_{j \in J} p_j(\mathbf{r}', \mathbf{v}') C_j(\mathbf{v} | \mathbf{r}', \mathbf{v}') \quad (6)$$

note that C_j may not be a normalized density function, e.g. $C_j = 0$ for absorption processes.

These equations, eq. 5 and 6, characterize the computation in the particle Monte Carlo method used by EIRENE and Eiron. The transport kernel is used as an integrand when sampling the location of collisions from a Poisson point process along a particle's flight path, and the collision kernel determines a particle's velocity after a collision.

2.1 Sampling from the Poisson point process

The transport kernel (Eq. 5), forms a Poisson point process with inhomogeneous rate parameter Σ_t . Integrating over Σ_t gives us a homogeneous Poisson process on the real line. We can then

sample from this simpler process by using a standard exponential random variable X , which models how many expected collisions a particle undergoes before an actual collision occurs. We then numerically integrate over Σ_t until the value of the integral equals X ; once that condition is reached, the upper bound of the integral is L by definition:

$$X \sim \text{Exp}(1), \quad X = \int_0^L \Sigma_t(\mathbf{r}' + s \frac{\mathbf{v}}{|\mathbf{v}|}, \mathbf{v}) ds \quad (7)$$

2.2 Simplified collision rate coefficients

The total macroscopic cross section Σ_t is the sum of individual collision processes' cross sections Σ_j (Eq. 8):

$$\Sigma_t(\mathbf{r}, \mathbf{v}) = \sum_{j \in J} \Sigma_j(\mathbf{r}, \mathbf{v}) = \sum_{j \in J} \frac{n_j(\mathbf{r}) \langle \sigma_j \cdot |\mathbf{v}_{rel,j}| \rangle}{|\mathbf{v}|} \quad (8)$$

where n_j is the number density of the background species associated with collision process j , σ_j is the microscopic cross section of collision process j , $\mathbf{v}_{rel,j}$ is the relative velocity between the neutral test particle and the background plasma species related to process k , and $\mathbf{v}$ is the test particle velocity.

Typically, in the context of a neutral particle colliding with an electron background, $|\mathbf{v}_{bg}| \gg |\mathbf{v}| \implies |\mathbf{v}_{bg}| \simeq |\mathbf{v}_{rel}|$, which means that the *collision rate coefficient* $\langle \sigma_k \cdot |\mathbf{v}_{rel,j}| \rangle$ can be approximately modeled as constant w.r.t. $\mathbf{v}$ [11, sec. 1.3.5]. As the number density n is also a constant w.r.t. $\mathbf{v}$, we can combine these in a *collision rate* ν :

$$\Sigma_t(\mathbf{r}, \mathbf{v}) = \sum_{j \in J} \frac{\nu_j(\mathbf{r})}{|\mathbf{v}|} = \frac{\nu_t(\mathbf{r})}{|\mathbf{v}|} \quad (9)$$

where ν_j is the collision rate of collision process j , and ν_t is the total collision rate. As a simplification, we've used this model with constant rates for the scalability experiments presented later in this article.

2.3 A note on non-linear neutral-neutral processes

The transport problem described above is linear and cannot, by itself, capture non-linear neutral-neutral collision processes. EIRENE is still able to describe neutral-neutral collisions through an iterative approach using a linearized Bhatnagar–Gross–Krook (BGK) approximation, where Maxwellian neutral backgrounds are created from the previous iteration's neutral particle distribution estimates (see Reiter et al. 1997 [13]). EIRENE does not treat these BGK collision processes any different from other ones. Thus, any improvements to this linear transport model will be applicable to non-linear models using the BGK approximation, such as EIRENE. We have therefore decided to focus on the linear transport model only, and will not discuss neutral-neutral processes further.

2.4 Monte Carlo transport of a single particle

Algorithm 1 simulates the transport of a single neutral particle of species i with initial state $(\mathbf{r}, \mathbf{v}, \text{event}, X)$. The particle collides with a background, modeled as collision rates in *grid.v*. Collisions are sampled from the Poisson point process associated with the transport kernel T in the inner loop of the algorithm. The loop follows the particle until it either collides or encounters

Algorithm 1 Simulates a particle from some state $(rng_index, \mathbf{r}, \mathbf{v}, i, event, X)$ until it undergoes absorption or is outside either the local grid boundaries $grid.geometry$ or the global simulation boundaries $geometry$. rng_index identifies a position in a random number stream to be used when generating random numbers. Returns an updated state.

```

function SIMULATE_PARTICLE( $rng\_index, \mathbf{r}, \mathbf{v}, i, event, X, grid, geometry$ )
  while true do
    // Transport kernel
     $sum \leftarrow 0$ 
    while  $sum < X$  do
      if  $\mathbf{r} \notin geometry$  then
        if  $\mathbf{r}$  at a periodic boundary then
          return ( $rng\_index, \text{WRAP}(\mathbf{r}), \mathbf{v}, i, X - sum, boundary\_crossed$ )
        else
          return ( $rng\_index, \mathbf{r}, \mathbf{v}, i, X - sum, out\_of\_bounds$ )
      if  $\mathbf{r} \notin grid.geometry$  then
        return ( $rng\_index, \mathbf{r}, \mathbf{v}, i, X - sum, boundary\_crossed$ )
       $\mathbf{z} \leftarrow \text{CELL\_INDEX}(grid, \mathbf{r})$ 
       $l \leftarrow \text{TRACK\_LENGTH}(\mathbf{z}, \mathbf{r}, \mathbf{v})$ 
       $\mathbf{r} \leftarrow \text{NEXT\_CELL\_INTERSECTION}(\mathbf{r}, \mathbf{v})$ 
       $\nu_t \leftarrow \sum_{j \in J} grid.\nu[i, \mathbf{z}, j]$  ▷ Simplified collision rate model
       $sum \leftarrow sum + l \cdot \nu_t / |\mathbf{v}|$ 

      if  $sum > X$  then
         $l \leftarrow l - (sum - X) \cdot |\mathbf{v}| / \nu_t$  ▷ Position and track-length correction
         $\mathbf{r} \leftarrow \mathbf{r} - (sum - X) \cdot \mathbf{v} / \nu_t$ 
        for each  $t$  in  $grid.tallies$  do
          #pragma omp atomic ▷ Only needed for shared memory (Alg. 3)
           $grid.estimates[i, \mathbf{z}, t] \leftarrow grid.estimates[i, \mathbf{z}, t] + f_t(\mathbf{v}, i, l)$ 

    // Collision kernel
    ( $rng\_index, event$ )  $\leftarrow \text{SAMPLE\_EVENT}(rng\_index, \mathbf{z}, \mathbf{v}, i)$ 
    if  $event = \text{ABSORPTION}$  then
      return ( $rng\_index, \mathbf{r}, \mathbf{v}, i, 0, event$ )
    ( $rng\_index, \mathbf{v}, i$ )  $\leftarrow \text{RESOLVE\_EVENT}(rng\_index, \mathbf{z}, \mathbf{v}, event)$ 
    ( $rng\_index, X$ )  $\leftarrow \text{SAMPLE\_EXPONENTIAL}(rng\_index)$ 

```

a boundary. After a collision happens, the type and result of each collision is determined by sampling from a categorical distribution, with probabilities proportional to the collision probabilities p_i in the collision kernel C . Estimates are tallied to *grid.estimate*s. Monte Carlo simulations record a wide variety of estimates or tallies — e.g. particle density, particle energy, reaction rates. In Algorithm 1, each tally is represented as an index t with an associated function f_t . As an example, for particle density, $f_t(\mathbf{v}, i, l) = l$, the track length estimator.

By repeatedly sampling particles from a particle source and passing them through Algorithm 1 we can create a number of parallel algorithms that use the same underlying algorithm. We’ve thus decoupled the Monte Carlo model (Algorithm 1) from the execution model (Algorithms 2 - 4), which allows us to create new execution models without touching the physics. Decoupling the Monte Carlo model from the parallelization also allows alternative Monte Carlo models to be substituted for the regular kinetic model, e.g., hybrid models like kinetic-diffusive Monte Carlo [14].

To make deterministic simulations possible, we pass in an *rng_index* to the algorithm, which consists of an rng seed corresponding to an rng stream, and a position in that rng stream. In Eiron, each particle has their own rng stream, and by incrementing the position in that stream every time a random number is consumed, we can be sure that the same random numbers sequence is always used to simulate the particle, regardless of parallelization. However, in domain-decomposed simulations, ensuring that this deterministic property holds forces us to implement particle generation a serial manner, limiting scalability. As will be explained in detail in Section 4.3, domain-decomposed simulations may therefore be configured to break determinism in favor of performance.

3 Domain-decomposed Monte Carlo

The traditional way to parallelize particle Monte Carlo is to distribute particle ranges to threads, with each thread replicating the entire domain. This scales perfectly as the particle count increases, but scales badly as the DoFs increase (as the grid resolution, domain size, or number of tallies grow). The issue is the associated per-core memory cost. A solution to this scalability issue is domain-decomposed Monte Carlo (DDMC). In a DDMC algorithm, the simulation grid is decomposed into subdomains that are distributed among processes. This significantly reduces the per-core memory cost, but introduces point-to-point communication of particles crossing subdomain boundaries, making it more complicated than the traditional approach.

Previous work on DDMC has mostly been in the context of photon transport, radiation transport, and fission reactor modeling. The earliest work on DDMC that we have found is by Alme et al. in [15], [16], and [17], who identify the issue of increasing memory consumption, along with an impedance mismatch issue (in the software engineering sense) when Monte Carlo models are coupled with finite-element models. Finite-element models are typically domain-decomposed, and coupling them with a non-DD Monte Carlo model creates the mismatch; the two models differ in what data regions and computational resources are allocated to a thread at runtime. This incurs a performance penalty at the interface between the two models, and makes it difficult to create a single resource allocation that would run both models in one compute job.

Scalable DDMC algorithms have been presented in previous work. Brunner and Brantley have done the most extensive scalability experiments in [18], where they present a DDMC algorithm that scales well for radiation transport problems in 3D. Choi and Joo have implemented DDMC on GPUs [19]. Other DDMC algorithms have been presented by Horelik et al. [20], Liang et al. [21], and García et al. [22];

4 Parallel algorithms in Eiron

We have implemented three parallel algorithms for particle Monte Carlo in Eiron (Algorithms 2-4). Algorithm 2 is a traditional domain-replication algorithm implemented with OpenMP – resembling EIRENE’s current MPI-parallelization, Algorithm 3 is a shared memory algorithm implemented with OpenMP – resembling EIRENE’s current OpenMP-parallelization, and Algorithm 4 is a novel asynchronous DDMC algorithm.

Algorithm 2 Replicates the grid to each thread, parallelizes over particles

```

function DOMAIN_REPLICATION_MC(sources, grid)
  #pragma omp parallel
    grid'. $\nu$   $\leftarrow$  reference(grid. $\nu$ )
    grid'.estimates  $\leftarrow$  copy(grid.estimates)
    for each src in sources do
      #pragma omp for schedule(static, src.n_particles/n_threads)
      for j  $\leftarrow$  0, src.n_particles do
        p  $\leftarrow$  GENERATE_PARTICLE(src)
        while p.event  $\in$  {particle_generated, boundary_crossed} do
           $\triangleright$  We may have to resimulate a particle due to periodic boundaries
          p  $\leftarrow$  SIMULATE_PARTICLE(p, grid', grid.geometry)

    for each i, z, t in grid.estimates do
      #pragma omp atomic
      grid.estimates[i, z, t]  $\leftarrow$  grid.estimates[i, z, t] + grid'.estimates[i, z, t]

```

Algorithm 3 Stores grid in shared memory, parallelizes over particles.

```

function SHARED_MEMORY_MC(sources, grid)
  #pragma omp parallel
    for each src in sources do
      #pragma omp for schedule(static, src.n_particles/n_threads)
      for j  $\leftarrow$  0, src.n_particles do
        p  $\leftarrow$  GENERATE_PARTICLE(src)
        while p.event  $\in$  {particle_generated, boundary_crossed} do
           $\triangleright$  We may have to resimulate a particle due to periodic boundaries
          p  $\leftarrow$  SIMULATE_PARTICLE(p, grid, grid.geometry)

```

Algorithm 4 Asynchronous domain-decomposed Monte Carlo. SID stands for subdomain id, and $SID(p)$ gives the id of the subdomain that p is in; **self** stands for the local MPI rank; $generation_mode$ is explained in Section 4.3.

```

function DOMAIN_DECOMPOSED_MC(sources, grid, generation_mode)
  subdomains  $\leftarrow$  decompose grid into subdomains among all MPI ranks
  sources  $\leftarrow$  CREATE_LOCAL_SOURCES(sources, subdomains, generation_mode)
  live_ranks  $\leftarrow$  set of all MPI ranks
  live_parts  $\leftarrow$  0
  dead_parts  $\leftarrow$  hash table from MPI ranks to counts, one entry per rank
  buffers  $\leftarrow$  hash table from SIDs to buffers, one entry per neighbor subdomain
  iteration  $\leftarrow$  1
  while  $|live\_ranks| > 0$  do
    (sim_buffer, live_ranks, live_parts)  $\leftarrow$  TRY_RECV(live_ranks, live_parts)
    for each src in sources do
      while  $|sim\_buffer| < BUFFER\_SIZE \wedge src$  not depleted do
         $p \leftarrow$  GENERATE_PARTICLE(src)
        if  $p.r$  within subdomains then
           $p.generating\_rank \leftarrow self$ 
           $sim\_buffer \leftarrow p$ 
          Increment live_parts
      for each  $p$  in sim_buffer do
         $p \leftarrow$  SIMULATE_PARTICLE( $p$ , subdomains[ $SID(p)$ ], grid.geometry)
        if  $p.event = boundary\_crossed$  then
           $buffers[SID(p)] \leftarrow p$ 
          if  $|buffers[SID(p)]| = BUFFER\_SIZE$  then
             $dead\_parts \leftarrow$  SEND_BUFFER( $buffers[SID(p)]$ ,  $SID(p)$ , dead_parts)
          else
            Increment  $dead\_parts[p.rank]$ 
         $live\_parts \leftarrow live\_parts - dead\_parts[self]$ 
         $dead\_parts[self] \leftarrow 0$ 
      if  $iteration \equiv 0 \pmod{SEND\_PERIOD}$  then
        for each (sid, buffer) in buffers do
           $dead\_parts \leftarrow$  SEND_BUFFER(buffer, sid, dead_parts)
         $dead\_parts \leftarrow$  SEND_DEAD_PARTICLE_COUNT(dead_parts)
      if all local sources are depleted  $\wedge live\_parts = 0$  then
        Remove self from live_ranks
        BROADCAST_RANK_TERMINATION(c)
    Increment iteration

```

Listing 1 Communication procedures in the DDMC algorithm. **self** stands for the local MPI rank.

```

function TRY_RECV(live_ranks, live_parts)
  if a message msg has arrived then
    live_parts  $\leftarrow$  live_parts - msg.tag
    if msg.buffer is not empty then
      return (msg.buffer, live_ranks, live_parts)
    if msg.buffer is empty  $\wedge$  msg.tag = 0 then
      Remove msg.source from live_ranks
  return (empty buffer, live_ranks, live_parts)

function SEND_BUFFER(buffer, sid, dead_parts)
  rank  $\leftarrow$  Select an MPI rank that is assigned to subdomain sid
  t  $\leftarrow$  MIN(dead_parts[rank], MPI_TAG_UB)
  Asynchronously send buffer to rank with tag=t
  dead_parts[rank]  $\leftarrow$  dead_parts[rank] - t
  return dead_parts

function SEND_DEAD_PARTICLE_COUNT(dead_parts)
  for each (rank, count) in dead_parts do
    if count > 0 then
      t  $\leftarrow$  MIN(count, MPI_TAG_UB)
      Asynchronously send empty message to rank with tag=t
      dead_parts[rank]  $\leftarrow$  dead_parts[rank] - t
  return dead_parts

function BROADCAST_RANK_TERMINATION
  Asynchronously send empty msg with tag=0 to all MPI ranks except self

```

4.1 The domain replication algorithm

Algorithm 2 is the traditional domain replication algorithm, implemented using OpenMP. This algorithm copies *grid.estimate*s to each thread, but still shares the read-only *grid.v*. This algorithm scales well for small grids that fit entirely into cache, as discussed in Section 5.2.1. However, memory usage grows linearly with the number of threads, which is why this algorithm has trouble scaling to larger grids.

4.2 The shared memory algorithm

Algorithm 3 mimics the shared memory OpenMP algorithm implemented in EIRENE. In this algorithm, the entire *grid* is stored in shared memory; each OpenMP thread runs on a (usually separate) CPU core, and those cores are tallying to the same shared memory location. As they are writing to shared memory, the writes must be done using atomic increments (using the `omp atomic` directive).

The upside of a shared memory approach is a smaller memory footprint, as each added thread only adds a negligible memory cost in the form of particle data and OpenMP overhead (such as per-thread call stacks). The downside of this approach is that multiple threads are reading from and writing to the same memory locations, and so writes from core *A* will invalidate cache lines

in core B and vice versa. This may repeat many times when two cores are operating on the same memory region simultaneously, a behavior known as *cache thrashing*.

4.3 The domain-decomposed algorithm

Algorithm 4 is the DDMC algorithm, implemented with MPI, and Listing 1 shows the communication procedures used by that algorithm. The algorithm assigns a number of subdomains to each MPI rank, creates a set of local particle sources, and generates particles from those sources, recording the generating rank in the particle state.

There are two modes of particle generation: *deterministic* and *decomposed*. In the *deterministic* mode each rank selects those sources that overlap its assigned subdomains, and generates each particle from those sources, rejecting those that do not spawn within an assigned subdomain (the same scheme that Horelik et al. use [20, sec. 2.3]). This ensures that the same particle trajectories are generated regardless of how the grid is decomposed, but because each rank generates every particle, this is a serial cost proportional to the number of particles, limiting scalability. In the *decomposed* mode, the particle sources are decomposed into subsources, the subsources are aligned with the subdomains and particles are distributed among the subsources such that statistically, the distribution of generated particles remains the same. This eliminates the serial cost proportional to the total number of particles. However, in this mode the exact location of where particles are generated depends on how many subdomains there are.

Regardless of generation mode, the MPI ranks then simulate generated and received particle buffers within their own subdomains. If a particle crosses into a new subdomain, it is communicated to an MPI rank that has been assigned that subdomain; this may be the same MPI rank if many neighboring subdomains are assigned to one rank. The ranks coordinate when to end the simulation by keeping track of locally generated particles and signalling the other ranks when all of them have terminated. Any time a particle is generated, the generating rank increments its local *live_parts* counter. Each rank also records particle terminations per generating rank in a hash table. The terminated particle counts are communicated to the generating ranks using MPI tags, either piggybacking on a particle buffer message, or by sending empty messages periodically. Whenever a rank receives a message, the number of live particles is adjusted down by the terminated particle count in the tag. When a rank has generated all particles, and all those particles have terminated, it broadcasts an empty message with an MPI tag of 0, indicating that all work generated at the sending rank is complete. When a rank itself is terminated and has received a termination message from all other ranks, the rank exits the simulation loop.

The termination control method described above differs from those used by earlier algorithms that keep track of the total sum of generated and terminated particles with synchronous global operations (such as [18],[21]), or those that use synchronous iteration to ensure all particles are simulated (such as [19],[20],[22]). This method requires no globally synchronous communication operations, and can rely purely on asynchronous point-to-point messages. This should decrease the overall waiting time, as communication and computation can now overlap as much as possible.

The algorithm has two parameters: `BUFFER_SIZE` — controlling the size of the particle buffer — and `SEND_PERIOD` — controlling how often partial buffers and termination messages sent. It seems that tuning these parameters has little impact on the runtime; we have used `BUFFER_SIZE = 64` and `SEND_PERIOD = 32` in this work.

5 Scalability experiments

We ran a strong scaling experiment and a weak scaling experiment. In both experiments, there is one test particle species with two collision processes: one scattering process with collision rate

ν_s , and one absorption process with collision rate ν_a ; collision processes are uniform in the entire grid. There are two tallies being recorded: the particle density and the particle energy. Thus the collision grid and the tally grid both store two double-precision floating-point numbers in each cell, leading to a memory cost per cell of $(2 + 2) * \text{sizeof}(\text{double}) = 32B$. The grid geometry is a 2D square grid. The basic unit of the experiments is a grid of area 1. In the strong scaling experiment, this unit grid is subdivided into smaller parts; in the weak scaling experiment, the unit grid is the size of a subdomain, and so the total area of the grid grows with the number of CPU cores.

We decided to lock the mean speed of all particles to 1, and to have the scattering be an isotropic rotation operation. This leaves us with two parameters to adjust: the mean free path λ , and the scattering fraction ν_s/ν_t . Based on an initial parameter study, we saw that low λ and high ν_s/ν_t had a positive effect on scalability. Because this indicates that high-collisional regimes scale better than low-collisional regimes, we decided to test two parameter combinations: $\lambda = 0.25, \nu_s/\nu_t = 0.01$ (*low-collisional*), and $\lambda = 0.05, \nu_s/\nu_t = 0.99$ (*high-collisional*).

There are some differences between the strong scaling and weak scaling experiment configurations. First, the strong scaling grid has terminating boundaries; the weak scaling grid periodic boundaries. Second, the strong scaling experiment uses a line source at the bottom wall; the weak scaling experiment uses an area source covering the entire grid, in order to keep the work per core constant. Finally, the strong scaling experiment uses the *deterministic* particle generation mode; the weak scaling experiment uses the *decomposed* particle generation mode (see Section 4.3). The *decomposed* generation mode is necessary because in the *deterministic* mode, particle generation eventually becomes the bottleneck when thousands of cores each generate all particles in the simulations, but simulate only a fraction of them. Future improvements may speed this up, but as it stands deterministic particle generation is too slow.

5.1 Computing environment and reproducibility

The Eiron git repository is hosted at the University of Helsinki Gitlab [23]. All the scalability experiments were run on CSC’s Mahti supercomputer in Kajaani [3]. Eiron git commit *eee6a3ba* was compiled on Mahti with g++ 13.1.0 and Open-MPI 4.1.5. In order to ensure each working core had the maximum computational resources available, we reserved full nodes for each simulation. For Algorithms 2 and 3, we set the OpenMP affinity to *spread*. For Algorithm 4, we assigned subdomains to MPI ranks in Z-order [24], which minimizes the distance between cores processing neighboring subdomains within a node, improving communication performance. In order to run the experiments in a feasible time on a shared queue, we did not pin the nodes for the weak scaling experiment, which may have introduced some noise into those results.

Each CPU-node on Mahti has two 64-core AMD EPYC 7H12 CPUs, from the Zen 2 microarchitecture. The cores on a Zen 2 CPU reside on core complexes (CCXs); each CCX houses four cores. In a CCX, four cores share 16 MiBs of L3 cache, split into four slices of 4 MiB, for a total of 256MiB per CPU [25]. These cache details will be relevant for the analysis of the scaling results.

5.2 Strong scaling results

Strong scaling experiments were done by scaling a simulation from a single core to 128 CPU cores (a full node). Each simulation generates 128000 particles, and is run 6 times. We ignore the first run to eliminate potential warmup costs, and report the minimum of the last five.

We ran simulations at six different grid resolutions: 128^2 , 256^2 , 512^2 , 1024^2 , 2048^2 , 4096^2 . For the DDMC algorithm (Algorithm 4), each core is assigned a single subdomain. The subdomains

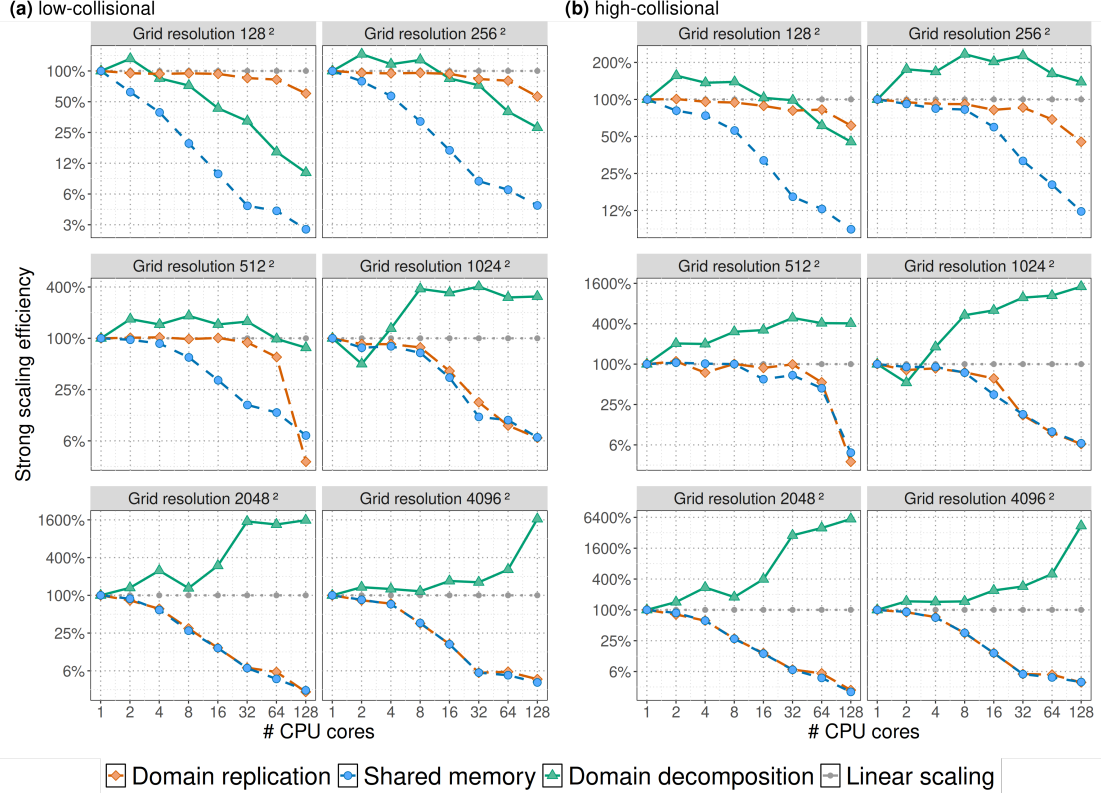


Figure 1: Log-log plots of the strong scaling efficiency for all three parallel algorithms and all six grid resolutions in the low-collisional case (a) and the high-collisional case (b). The strong scaling efficiency at n CPU cores is defined as $t_1/(n \cdot t_n)$, where t_m is the wallclock time when running with m CPU cores. Higher is better.

are therefore only square for core counts that are even powers of two (for odd powers-of-two, subdomains are 2:1 rectangles).

Figure 1 plots the strong scaling efficiency for all three parallel algorithms at these parameter configurations. Algorithm 4 has the best strong scaling efficiency in most cases, with superlinear scaling for larger grid resolutions. In the following three subsections, the strong scaling results of each parallel algorithm are discussed in more detail.

5.2.1 Strong scaling of the domain replication algorithm

Domain replication (Algorithm 2) scales well with lower-resolution grids that fit into the cache, but scalability deteriorates quickly when the grid resolution is increased. However, unlike the other algorithms, scalability is not at all sensitive to the other simulation parameters, as the low-collisional and high-collisional efficiency curves in Figure 1 look identical.

Still looking at Figure 1, we see that domain replication scales linearly for grid resolutions 128^2 , 256^2 , and 512^2 until 32 cores. This is explained by the cache sizes in the Zen 2 microarchitecture: (1) as stated in Section 5.1, groups of four cores share 16MiB of L3 cache, (2) a 512^2 -resolution grid takes up $512^2 \times 32 B = 8MiB$ — and this is most of the working set size, other data is negligible in size. In other words, for grids $\leq 512^2$, the entire working set fits into the shared L3 cache. At 512^2 , the efficiency drops a little for the final step or two, because the working set no longer fits into L3; each core no longer has use of a full L3 cache, but shares it with two (the 64 core case) or four others (the 128 core case). For higher-resolution grids, the scalability is significantly worse for the same reason: the working set is significantly larger than the L3 cache.

5.2.2 Strong scaling of the shared memory algorithm

Looking at Figure 1, Algorithm 3, the shared memory algorithm, scales linearly for a few steps in favorable configurations. After that, the scalability deteriorates as a higher core count increases the probability of cache collisions, which leads to cache thrashing. Then (due to OpenMP thread affinity *spread*), the L3 caches become shared by pairs of cores when the total core count is 64, and shared by four cores when the total core count is the full 128. The shared memory algorithm benefits from these shared caches, which is why performance tends to improve again at 64 and 128 CPU cores.

For the low-collisional case (Figure 1 A), the 256^2 grid is slightly sublinear until 2-4 cores, and the 512^2 grid slightly sublinear until 8 cores. For the high-collisional case (Figure 1 B), the algorithm scales linearly to about 8 cores for a 256^2 grid; 16 cores for a 512^2 grid. The most likely reason for the relatively worse performance in low-collisional regimes is that particles traveling further per transport step cause each core to access data from a larger and more spread out memory region, increasing the probability of cache collisions.

5.2.3 Strong scaling of the domain-decomposed algorithm

Domain decomposition (Algorithm 4) scales better than the other two algorithms in most cases, especially when it comes to high grid resolutions. Looking at Figure 1, at 128 cores, the strong scaling efficiency is above 100% in both high-collisional and low-collisional cases at grid resolutions 1024^2 , 2048^2 , and 4096^2 , reaching as high as 6400% for the largest grid resolution in the high-collisional case. The full-node efficiency is also above 100% in the high-collisional case with grid resolutions of 256^2 and 512^2 . This is due to increased cache efficiency; with one CPU core, only a fraction of the grid data can be held in that single CPU core’s cache; at 128 cores, a larger

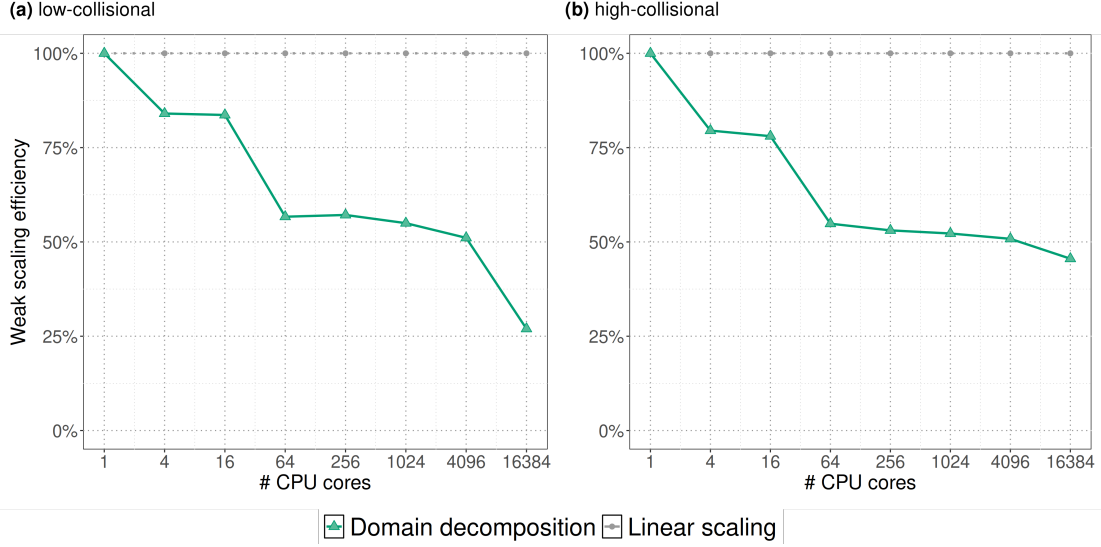


Figure 2: Log-log plots of weak scaling efficiency for Algorithm 4 (DDMC) with 256^2 subdomain resolution. The weak scaling efficiency at n CPU cores is defined as t_1/t_n , where t_m is the wallclock time when running with m CPU cores. Periodic boundaries and an area source covering the entire grid were used to make sure that the work per core is constant. (a) is the low-collisional case, (b) is the high-collisional case. Higher is better.

part of the grid (potentially all of it) is distributed across all the caches on the node. This effect is explored in more detail in Section 5.4.

The strong scaling of this algorithm is clearly sensitive to the simulation parameters, as scalability improves in the high-collisional case. The reason is, a shorter mean free path and a higher scattering ratio increases the relative amount of computation work compared to communication work. The higher communication overhead in the low-collisional cases limits scalability, which is why the domain replication algorithm (Algorithm 2) beats the domain decomposition algorithm in the low-collisional case when grid resolution is low, 256^2 and 128^2 .

5.3 Weak scaling of the domain-decomposed algorithm

Weak scaling experiments were done by scaling a DDMC simulation from a single core to 16384 CPU cores (128 nodes) while increasing both the number of grid cells and particles. Each CPU core is assigned a subdomain of grid resolution of 256^2 , and we generate 5000 particle per subdomain. We used the *decomposed* particle generation mode (see Section 4.3) for the weak scaling experiment, as we noticed that deterministic particle generation severely limits weak scalability at high core counts. The domain boundaries are periodic, making the problem symmetric and load balanced.

Figure 2 shows the weak scaling efficiency for the DDMC algorithm. The high-collisional regime scales better, the scalability falls until 64 cores, and then plateaus, ending up with a weak scaling efficiency of 45% at 16384 CPU cores. The low-collisional regime scales pretty much the same as the high-collisional regime up to 4096 CPU cores, but falls to 26% at 16384 CPU cores.

Grid resolution	Grid memory size	L3 miss rate	L3 slice miss rate
128^2	512 KiB	0%	0%
256^2	2 MiB	0%	0%
512^2	8 MiB	0%	50%
1024^2	32 MiB	50%	87.5%
2048^2	128 MiB	87.5%	98.4%
4096^2	512 MiB	98.4%	99.6%

Table 1: Cache miss rate model, $miss\ rate = 1 - cache\ size / grid\ memory\ size$. The Zen 2 L3 cache size is 16 MiB of which each core has a 4 MiB L3 slice attached to it. The model assumes that Monte Carlo transport simulation is characterized by random memory accesses to grid memory, and that other memory accesses are negligible.

5.4 Performance regimes

The amount of memory randomly accessed by a single CPU core is the biggest factor determining the performance of the Monte Carlo algorithms presented in this work. Performance is good in a cache-friendly regime, when the amount of data accessed is well serviced by the CPU’s cache; performance is bad in a cache-unfriendly regime, when the CPU cores access more data than the cache can store.

Table 1 shows a simple probabilistic model for the L3 cache miss rate of the grids we’re using in our scaling experiments. Figure 3 shows the relationship between the grid’s memory footprint and particle processing rate in the DDMC strong-scaling results. For a cleaner plot, the figure only shows data for square subdomain resolutions, or half the dataset. In Figure 3, we see that each series has the sharpest change in processing rate from 8 MiB to 2 MiB (512^2 to 256^2), marking a transition between performance regimes. In the cache-unfriendly performance regime — for each subdomain resolution greater than 2 MiB — each subdivision of the grid improves the processing rate. In the cache-friendly performance regime, the change in processing rate starts to drop off, and eventually become negative. This happens immediately after the performance regime transition in the low-collisional case and a little later in the high-collisional case.

We can see from Table 1 that the performance regime transition corresponds to a change from a 50% miss rate to a 0% miss rate in the 4 MiB L3 slice of the Zen 2 CPU. The L3 slice is local to each core, and while it is not private to the CPU core, because the subdomain data is private, it is acting as if it were a private cache. If each core can fit all the grid data it needs in this local cache, then it is not contending for cache resources with other cores. The fact that further subdivision improves performance in the high-collisional case is most likely due to improved L2 and L1 cache efficiency.

The performance regimes are visible in the strong-scaling plots of both the domain replication algorithm and the domain-decomposed algorithm. The domain replication algorithm has better strong scaling when the single-core run is in the cache-friendly regime, and DDMC scales much better when the single-core run is in the cache-unfriendly regime — because DDMC then transitions into the cache-friendly regime. We can clearly see the transition to the cache-friendly regime in the 1024^2 , 2048^2 , and 4096^2 DDMC series in Figure 1.

Which performance regime a simulation belongs to is a function of the specific hardware that is being used, and the memory cost per grid cell, which is much smaller for these performance experiments than for production simulations. As production simulations use more memory, they are more likely to lie in the cache-unfriendly regime, meaning that domain-decomposition may have an even greater advantage in production simulations.

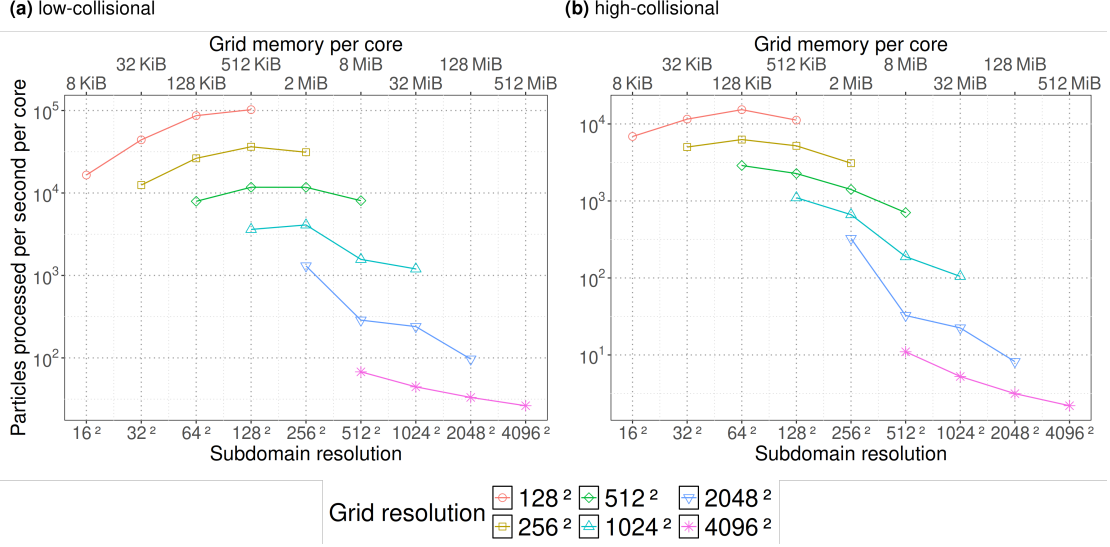


Figure 3: Log-log plot of particle processing rate as a function of subdomain resolution using the DDMC strong scaling data. Higher is better. We have left out the non-square results for easier comparison to Table 1, and to remove subdomain shape as a potential confounding factor.

6 Discussion

In this article, we introduced a new open source Monte Carlo code, Eiron, and presented a domain-decomposed Monte Carlo (DDMC) algorithm (Algorithm 4), aiming to solve the issue of increasing memory consumption in neutral transport simulations for large fusion reactors. The DDMC algorithm uses a novel asynchronous termination control method, making all communication in the algorithm asynchronous, unlike previous algorithms in the literature. The scalability of this algorithm was compared against two other algorithms that EIRENE uses (Algorithms 2 and 3), also implemented in Eiron, and it was shown that the DDMC algorithm scales better than these algorithms for large grids. We therefore conclude that domain decomposition would improve EIRENE’s scalability and performance, and would remove the constraint that the whole simulation grid must fit in the memory of a single node. Removing this memory constraint would enable numerical simulations requiring higher resolution grids — e.g. Larmor scale resolved global 3D turbulence simulations of next generation machines like ITER [2].

We found that two factors had significant effects on the scalability of the asynchronous DDMC algorithm and the shared memory algorithm (Algorithm 3): the per-core memory footprint of grid data, and the ratio of computation and communication work. On Mahti, the strong scaling of the DDMC algorithm is superlinear for grids larger than 4 MiB due to improved cache efficiency. Both strong and weak scaling improves when there is more computational work compared to communication work, but the overall scalability is not affected by this until very high core counts (16384 cores).

There are some limitations to these results. First, we have not explored the effect of load imbalance on the algorithm. Second, we have used a very reduced collision model. The more sophisticated rate coefficient models used by EIRENE will increase both the computational cost and memory costs of particle transport. Because a more complicated collision model increases the relative amount of computational work compared to communication, this indicates that

the scalability of DDMC in EIRENE may be better than what we have found in our scaling experiments.

Improvements that we wish to implement in future work include more realistic collision models, better performance of deterministic particle generation, as well as load-balancing schemes based on subdomain work estimates.

Finally, we want to discuss the issue of compute platform. GPUs are the compute engines of modern HPC clusters, and heavy numerical simulations — such as simulations of large fusion devices — should make use of them. We are aware of efforts within the fusion community to develop GPU models for neutrals, e.g. one model based on NESO-Particles [26] (not particle Monte Carlo) being developed at the United Kingdom Atomic Energy Authority (UKAEA). In future work, we want to create a GPU port of the DDMC algorithm presented in this work. DDMC has been implemented on GPUs before, e.g., by Choi and Joo for fission reactor problems [19], so we know it is possible. Some modifications to the scheduling and communication patterns will be necessary, and ideas from event-based Monte Carlo may have to be incorporated (see [27]), but we believe that the DDMC algorithm presented in this work can be used as a base for a scalable GPU implementation.

7 Acknowledgements

The authors wish to acknowledge CSC – IT Center for Science, Finland, for computational resources. This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

References

- [1] D. Reiter, M. Baelmans, and P. Börner, “The EIRENE and B2-EIRENE codes,” *Fusion Science and Technology*, vol. 47, no. 2, pp. 172–186, 2005.
- [2] V. Quadri, P. Tamain, Y. Marandet, H. Bufferand, N. Rivals, G. Ciraolo, G. Falchetto, R. Düll, S. Sureshkumar, N. Varadarajan, H. Yang, H. Reimerdes, D. Oliveira, and D. Mancini, “Edge plasma turbulence simulations in detached regimes with the SOLEDGE3X code,” *Nuclear Materials and Energy*, vol. 41, p. 101756, 2024.
- [3] CSC, *Technical details about Mahti*, 2025.
- [4] S. Wurzel and S. Hsu, “Progress toward fusion energy breakeven and gain as measured against the Lawson criterion,” *Physics of Plasmas*, vol. 29, p. 062103, 2022.
- [5] M. Valentinuzzi, Y. Marandet, H. Bufferand, G. Ciraolo, and P. Tamain, “Two-phases hybrid model for neutrals,” *Nuclear Materials and Energy*, vol. 18, pp. 41–45, 2019.
- [6] W. Van Uytven, W. Dekeyser, M. Blommaert, N. Horsten, Y. Marandet, and M. Baelmans, “Advanced spatially hybrid fluid-kinetic modelling of plasma-edge neutrals and application to ITER case using SOLPS-ITER,” *Contributions to Plasma Physics*, vol. 62, no. 5-6, p. e202100191, 2022.

- [7] S. Wiesen, D. Reiter, V. Kotov, M. Baelmans, W. Dekeyser, A. Kukushkin, S. Lisgo, R. Pitts, V. Rozhansky, G. Saibene, I. Veselova, and S. Voskoboynikov, “The new SOLPS-ITER code package,” *Journal of Nuclear Materials*, vol. 463, pp. 480–484, 2015.
- [8] Y. Feng, H. Frerichs, M. Kobayashi, and D. Reiter, “Monte-carlo fluid approaches to detached plasmas in non-axisymmetric divertor configurations,” *Plasma Physics and Controlled Fusion*, vol. 59, p. 034006, feb 2017.
- [9] H. Bufferand, G. Ciraolo, Y. Marandet, J. Bucalossi, P. Ghendrih, J. Gunn, N. Mellet, P. Tamain, R. Leybros, N. Fedorczak, F. Schwander, and E. Serre, “Numerical modelling for divertor design of the WEST device with a focus on plasma–wall interactions,” *Nuclear Fusion*, vol. 55, p. 053025, apr 2015.
- [10] D. Borodin, F. Schluck, S. Wiesen, D. Harting, P. Börner, S. Brezinsek, W. Dekeyser, S. Carli, M. Blommaert, W. Van Uytven, M. Baelmans, B. Mortier, G. Samaey, Y. Marandet, P. Genesio, H. Bufferand, E. Westerhof, J. Gonzalez, M. Groth, A. Holm, N. Horsten, and H. Leggate, “Fluid, kinetic and hybrid approaches for neutral and trace ion edge transport modelling in fusion devices,” *Nuclear Fusion*, vol. 62, no. 8, p. 086051, 2022.
- [11] D. Reiter *et al.*, *The EIRENE Code User Manual*, 1.0.0 ed., 2023.
- [12] J. Spanier and E. M. Gelbard, *Monte Carlo principles and neutron transport problems*. Addison-Wesley series in computer science and information processing, Addison-Wesley, 1969.
- [13] D. Reiter, C. May, M. Baelmans, and P. Börner, “Non-linear effects on neutral gas transport in divertors,” *Journal of Nuclear Materials*, vol. 241-243, pp. 342–348, 1997.
- [14] B. Mortier, M. Baelmans, and G. Samaey, “A kinetic-diffusion asymptotic-preserving Monte Carlo algorithm for the Boltzmann-BGK model in the diffusive scaling,” *SIAM Journal on Scientific Computing*, vol. 44, no. 2, pp. A720–A744, 2022.
- [15] H. J. Alme, G. H. Rodrigue, and G. B. Zimmerman, “Parallel domain decomposition methods in fluid models with Monte Carlo transport,” in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PP 1997, Hyatt Regency Minneapolis on Nicollel Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*, SIAM, 1997.
- [16] H. J. Alme, G. H. Rodrigue, and G. B. Zimmerman, “Domain decomposition methods for parallel laser-tissue models with Monte Carlo transport,” in *Monte-Carlo and Quasi-Monte Carlo Methods 1998* (H. Niederreiter and J. Spanier, eds.), (Berlin, Heidelberg), pp. 137–148, Springer Berlin Heidelberg, 2000.
- [17] H. J. Alme, G. H. Rodrigue, and G. B. Zimmerman, “Domain decomposition models for parallel Monte Carlo transport,” *The Journal of Supercomputing*, vol. 18, no. 1, pp. 5–23, 2001.
- [18] T. A. Brunner and P. S. Brantley, “An efficient, robust, domain-decomposition algorithm for particle Monte Carlo,” *Journal of Computational Physics*, vol. 228, no. 10, pp. 3882–3890, 2009.
- [19] N. Choi and H. G. Joo, “Domain decomposition for GPU-based continuous energy Monte Carlo power reactor calculation,” *Nuclear Engineering and Technology*, vol. 52, no. 11, pp. 2667–2677, 2020.

- [20] N. Horelik, A. Siegel, B. Forget, and K. Smith, “Monte carlo domain decomposition for robust nuclear reactor analysis,” *Parallel Computing*, vol. 40, no. 10, pp. 646–660, 2014.
- [21] J. Liang, K. Wang, Y. Qiu, X. Chai, and S. Qiang, “Domain decomposition strategy for pin-wise full-core Monte Carlo depletion calculation with the Reactor Monte Carlo code,” *Nuclear Engineering and Technology*, vol. 48, no. 3, pp. 635–641, 2016.
- [22] M. García, J. Leppänen, and V. Sanchez-Espinoza, “A collision-based domain decomposition scheme for large-scale depletion with the Serpent 2 Monte Carlo code,” *Annals of Nuclear Energy*, vol. 152, p. 108026, 2021.
- [23] O. Lappi, H. Leggate, E. Loevbak, and T. Steel, “Eiron.” <https://version.helsinki.fi/lapposka/eiron>, 2025.
- [24] G. M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing,” tech. rep., International Business Machines Company New York, 1966.
- [25] M. Velten, R. Schöne, T. Ilsche, and D. Hackenberg, “Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors,” in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, (Beijing China), pp. 165–175, ACM, Apr. 2022.
- [26] W. Saunders, J. Edgeley, S. Powell, J. Cook, M. Barton, C. MacMackin, and O. Parry, “NESO-Particles.” <https://github.com/ExCALIBUR-NEPTUNE/NESO-Particles>, 2025.
- [27] S. P. Hamilton, S. R. Slattery, and T. M. Evans, “Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms,” *Annals of Nuclear Energy*, vol. 113, pp. 506–518, 2018.