The Plasma Simulation Code: A modern particle-in-cell code with load-balancing and GPU support

Kai Germschewski\textsuperscript{a,}\textsuperscript{*}, William Fox\textsuperscript{b}, Narges Ahmadi\textsuperscript{a}, Liang Wang\textsuperscript{a}, Stephen Abbott\textsuperscript{a}, Hartmut Ruhl\textsuperscript{c}, Amitava Bhattacharjee\textsuperscript{d}

\textsuperscript{a}Space Science Center \& Department of Physics, University of New Hampshire, Durham, NH
\textsuperscript{b}Space Science Center, University of New Hampshire, Durham, NH
\textsuperscript{c}Faculty of Physics, Ludwig Maximilians University, München, Germany
\textsuperscript{d}Princeton Plasma Physics Laboratory \& Department of Astrophysical Sciences, Princeton University, Princeton, NJ

Abstract

Recent increases in supercomputing power, driven by the multi-core revolution and accelerators such as the IBM Cell processor, graphics processing units (GPUs) and Intel’s Many Integrated Core (MIC) technology have enabled kinetic simulations of plasmas at unprecedented resolutions, but changing HPC architectures also come with challenges for writing efficient numerical codes. This paper describes the Plasma Simulation Code (psc), an explicit, electromagnetic particle-in-cell code with support for different order particle shape functions. We focus on two distinguishing feature of the code: patch-based load balancing using space-filling curves, and support for Nvidia GPUs, which achieves substantial speed-up of up to more than 6× on the Cray XK7 architecture compared to a CPU-only implementation.

Keywords: particle-in-cell, kinetic, plasma, GPU

1. Introduction

Rapidly advancing computer technology has enabled large first-principles plasma simulations in recent years. The kinetic description of plasma, the Vlasov-Maxwell system of equations, while computationally much more expensive, overcomes many limitations of fluid descriptions like magnetohydrodynamics (MHD) or extended MHD models. Fluid models describe plasma behavior at large scales very well, but approximations need to be made at small scales, which occur in magnetic reconnection and turbulence. For example, at the ion skin depth scale $d_i$, the one-fluid approximation breaks down, electrons and ions decouple and the magnetic field remains frozen to the electron flow. Reconnection requires breaking the frozen-in condition which occurs at electron scales, which can be represented in a fluid model in a generalized Ohm’s Law including electron inertia and electron pressure tensor effects, but finding appropriate closures is still an area of active research, see e.g. [1]. Kinetic particle-in-cell simulations also allow investigation of problems beyond the scope of fluid models, e.g., particle acceleration [2]. Kinetic plasma models naturally incorporate all of the necessary physics. Particle-in-cell codes, while often run with modified physical parameters, e.g., reduced ion/electron mass ratio and speed of light, are now capable of simulating multi-scale problems reaching

\textsuperscript{*}Corresponding author

\textit{Email addresses:} kai.germaschewski@unh.edu (Kai Germschewski), william.fox@unh.edu (William Fox), narges.ahmadi@unh.edu (Narges Ahmadi), liang.wang@unh.edu (Liang Wang), s.abbott@unh.edu (Stephen Abbott), hartmut.ruhl@uni-muenchen.de (Hartmut Ruhl), amitava@princeton.edu (Amitava Bhattacharjee)
from electron through ion to global scales reaching 100’s of \(d_i\) in two and even three dimensions. While efforts are underway to overcome some of the algorithmic limitations of explicit particle-in-cell methods (see, e.g., [3, 4]), explicit particle-in-cell methods scale efficiently to the largest supercomputers available today and are commonly used to address challenging science problems.

The Plasma Simulation Code \((psc)\) is an explicit, electromagnetic particle-in-cell code implementing similar methods as, e.g., \(vpic\) [5], \(osiris\) [6] and \(vorpal\) [7]. \(psc\) is based on H. Ruhl’s original version [8], but has been rewritten as modular code which supports flexible algorithms and data structures. In this paper, we will review the main underlying particle-in-cell methods, and then focus on two distinguishing features implemented in \(psc\): Support for patch-based dynamic load balancing, which address both performance and memory issues in simulations where many particles move between local domains, and support of GPUs, which enhances performance by more than six-fold on the Cray XK7 architecture by making use of the Nvidia K20X GPU.

2. Particle-in-cell method

2.1. Kinetic description of plasmas

The particle-in-cell method [9] [10] [4] solves equations of motions for particles and Maxwell’s equations to find forces between those particles, very similar to the first-principle description of a plasma as a system of charged particles. It is, however, better understood as a numerical method to solve the Vlasov-Maxwell system of equations that describes the time evolution of the particle distribution function \(f_s(x, p, t)\) where \(s\) indicates the species:

\[
\frac{\partial f_s}{\partial t} + v \cdot \frac{\partial f_s}{\partial x} + q_s(E + v \times B) \cdot \frac{\partial f_s}{\partial p} = 0
\]

The electromagnetic fields \(E\) and \(B\) are self-consistently evolved using Maxwell’s equations:

\[
\nabla \cdot E = \frac{\rho}{\varepsilon_0} \tag{2}
\]

\[
\nabla \cdot B = 0 \tag{3}
\]

\[
\frac{\partial E}{\partial t} = c^2 \nabla \times B - \frac{j}{\varepsilon_0} \tag{4}
\]

\[
\frac{\partial B}{\partial t} = -\nabla \times E \tag{5}
\]

where charge density \(\rho\) and current density \(j\) are obtained from the particle distribution functions:

\[
\rho = \sum_s q_s \int f_s(x, p, t) \, d^3 p \tag{6}
\]

\[
j = \sum_s q_s \int v f_s(x, p, t) \, d^3 p \tag{7}
\]

The divergence equations (2), (3) in Maxwell’s equations can be considered as initial conditions. If they are satisfied at some initial time, it is easy to show from Ampère’s Law (4) and Faraday’s Law (5) that they will remain satisfied at all times given that the charge continuity equation also holds:

\[
\partial_t \rho + \nabla \cdot j = 0. \tag{8}
\]

2.1.1. Particle-in-Cell method

The particle-in-Cell method approximates the distribution function \(f_s\) by representing it using quasi-particles with finite extent in configuration space:

\[
f_s(x, p, t) = \sum_{i=1}^{N_i} N_i^s \phi(x - x_i^s(t)) \delta^3(p - p_i^s(t)) \tag{9}
\]
Using the δ-function in velocity space ensures that the spatial extent of each quasi-particle remains constant in time.

The selection of the shape function determines properties of the numerical method. In general, the 3-d shape function is chosen to be the tensor product of 1-d shape functions in each coordinate direction, and normalized, symmetric shape functions with compact support are used. Equations of motions for the quasi-particles can then be derived by taking moments of the Vlasov equation:

\[
\frac{dN^i_s}{dt} = 0, \quad \frac{d\mathbf{x}^i_s}{dt} = \mathbf{v}^i_s, \quad \frac{dp^i_s}{dt} = q_s(\mathbf{E}_s + \mathbf{v}^i_s \times \mathbf{B}_s) \tag{10}
\]

The first equation expresses that the number of actual particles \(N^i_s\) that each quasi-particle \(i\) of species \(s\) represents remains constant. The other two equations are the usual equations of motion for a point particle with the modification that the electromagnetic fields \(\mathbf{E}_s, \mathbf{B}_s\) acting on the particle are given by

\[
\mathbf{E}_s = \int \mathbf{E} \phi(\mathbf{x} - \mathbf{x}^i_s) d^3x, \quad \mathbf{B}_s = \int \mathbf{B} \phi(\mathbf{x} - \mathbf{x}^i_s) d^3x \tag{11}
\]

which means that the electromagnetic fields are averaged over the extent of the particle.

Using finite-size quasi-particles is the main advantage in the particle-in-cell method. It is computationally cheaper, since it allows to solve the field equations on a mesh, rather than directly calculating the interaction of each particle with all others, which scales like \(O(N^3)\), \(N\) being the total number of particles, even though this can be improved to \(O(N \log N)\) by fast multipole methods. More importantly, even with today’s very powerful computers, it is not possible to simulate as many particles as comprise real plasmas of interest, and it will remain unfeasible for the foreseeable future. Lowering the number of particles to a number that is possible to simulate, one must be careful to not change the nature of the plasma. Plasmas are traditionally weakly coupled, i.e., the interaction between particles is dominated by collective behavior rather than individual particle-particle forces. The particle-in-cell method reduces the occurrence of strong particle-particle interactions because particles are now of finite extent, which means that their interaction potential weakens when two particles approach closer than their spatial size, while representing the long-range interactions faithfully.

2.2. FDTD method for solving Maxwell’s equations

The finite-difference time domain (FDTD) method has a long history of being used for computationally solving Maxwell’s equations. The FDTD method has the desirable feature of satisfying some conservation properties of the underlying continuum equations in the discrete. It employs the staggered Yee grid, as shown in Fig. 1, to represent magnetic fields on faces, electric fields and current densities on edges and charge densities on corners of the computational mesh.

We define the following discrete curl operators:

\[
(\nabla^+ \times \mathbf{E})_{i,j+1/2,k+1/2} = \frac{E_{z,i,j+1,k+1/2} - E_{z,i,j,k+1/2}}{\Delta y} - \frac{E_{y,i+1/2,j,k+1} - E_{y,i,j+1/2,k}}{\Delta z} \tag{12}
\]

\[
(\nabla^- \times \mathbf{B})_{i+1/2,j,k} = \frac{B_{x,i+1/2,j,k+1} - B_{x,i+1/2,j,k-1/2}}{\Delta y} - \frac{B_{y,i+1,j,k} - B_{y,i+1,j-1/2,k}}{\Delta z} \tag{13}
\]

where the \(y\) and \(z\) components are obtained by cyclic permutation.

We also define the following discrete divergence operators:

\[
(\nabla^+ \cdot \mathbf{B})_{i+1/2,j+1/2,k} = \frac{B_{z,i+1/2,j+1/2,k+1} - B_{z,i+1/2,j+1/2,k}}{\Delta y} + \frac{B_{y,i+1,j+1,k+1/2} - B_{y,i+1,j+1/2,k}}{\Delta z} + \frac{B_{y,i+1/2,j,k+1} - B_{y,i+1/2,j,k-1/2}}{\Delta z} \tag{15}
\]

\[
(\nabla^- \cdot \mathbf{E})_{i,j,k} = \frac{E_{x,i+1/2,j,k} - E_{x,i,j,k+1/2}}{\Delta x} + \frac{E_{y,i+1/2,j,k} - E_{y,i,j,k-1/2}}{\Delta y} + \frac{E_{z,i,j+1/2,k} - E_{z,i,j,k+1/2}}{\Delta z} \tag{16}
\]
Figure 1. The staggered Yee grid unit cell. Depicted are the locations of magnetic fields on face centers (red), electric fields and current density (green), and charge density (blue).

Figure 2. Leap-frog time integration in the PIC method. Blue quantities represent electromagnetic field quantities and their update scheme. Red quantities are quasi-particle positions and momenta, also staggered in time. Interaction occurs by using the EM fields to find the Lorentz force on particles (black) and by using particle motion to find current density that feeds back into Maxwell’s equations (green).

Maxwell’s equations are discretized using these operators, and employ a leap-frog scheme staggered in time (see also Fig. 2):

\[
\frac{E_{ij}^{n+1/2} - E_{ij}^{n-1/2}}{\Delta t} = c^2 \nabla \times B_{ij}^n - \frac{j_{ij}^n}{\varepsilon_0} \tag{17}
\]

\[
\frac{B_{ij}^{n+1} - B_{ij}^n}{\Delta t} = -\nabla \times E_{ij}^{n+1/2} \tag{18}
\]

where

\[
E_{ij} = \left( E_{x,i+1/2,j,k}, E_{y,i,j+1/2,k}, E_{z,i,j,k+1/2} \right) \tag{20}
\]

\[
B_{ij} = \left( B_{x,i,j+1/2,k+1/2}, B_{y,i,j+1/2,k+1/2}, B_{z,i,j,k+1/2} \right) \tag{21}
\]

It is easy to show that the discrete operators satisfy

\[
\nabla^+ \cdot \nabla^+ = 0 \ , \ \nabla^- \cdot \nabla^- = 0 \tag{23}
\]

and hence, as in the continuum, the discretized divergence equations remain satisfied to round-off error at all times

\[
(\nabla^- \cdot E)_{ij} = \frac{\rho_{ij}}{\varepsilon_0} \ , \ (\nabla^+ \cdot B)_{i+1/2,j+1/2,k+1/2} = 0 \tag{24}
\]

provided that the charge continuity equation is also discretely satisfied:

\[
\frac{\rho_{ij}^{n+1/2} - \rho_{ij}^{n-1/2}}{\Delta t} + (\nabla^- \cdot j)_{ij} = 0 \tag{25}
\]

The FDTD method also satisfies a discrete version of Poynting’s Theorem, however when used in the context of a PIC method, energy is generally not exactly conserved because of the different discretization of \( \mathbf{j} \cdot \mathbf{E} \) in the Maxwell solver vs. the particle advance.
2.3. Time integration of the quasi-particle equations of motion

We use a standard leap-frog method to advance quasi-particles in time, see also Fig. 2.

\[
\frac{x_i^{n+1/2} - x_i^{n-1/2}}{\Delta t} = v_i^n
\]

(26)

\[
\frac{p_i^{n+1} - p_i^n}{\Delta t} = q_n \left( E_i^{n+1/2} + v_i^{n+1/2} \times B_i^{n+1/2} \right)
\]

(27)

where \(v_i^n = p_i^n/(m_i \gamma_i^n)\). We follow Boris [13] in choosing

\[
v_i^{n+1/2} = \frac{p_i^n + p_i^{n+1}}{2m_i \gamma_i^{n+1/2}}
\]

(28)

and splitting the momentum update into a half step acceleration by \(E\), a rotation by \(B\) and another half step acceleration by \(E\).

The shape functions used in the \(psc\) code are standard B-splines [4, 9]. The code currently supports both 1st and 2nd order interpolation by employing the flat-top \(b_0(\xi)\) B-spline and the triangular-shaped \(b_1(\xi)\) B-spline. \(b_0(\xi)\) is defined as

\[
b_0(\xi) = \begin{cases} 
1 & \text{if } |\xi| < 1/2 \\
0 & \text{otherwise}
\end{cases}
\]

(29)

Successive B-splines are defined recursively by folding the previous B-spline with \(b_0\):

\[
b_{n+1} = \int_{-\infty}^{\infty} b_0(\xi - \xi') b_n(\xi - \xi') d\xi'
\]

(30)

In particular, the \(psc\) code uses \(b_1\) and \(b_2\):

\[
b_1(\xi) = \begin{cases} 
1 + \xi & \text{if } -1 \leq \xi \leq 0 \\
1 - \xi & \text{if } 0 \leq \xi \leq 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
b_2(\xi) = \begin{cases} 
\frac{1}{2} \left( \frac{1}{2} + \xi \right)^2 & \text{if } -\frac{3}{2} \leq \xi \leq -\frac{1}{2} \\
\frac{3}{4} - \xi^2 & \text{if } -\frac{1}{2} \leq \xi \leq \frac{1}{2} \\
\frac{1}{2} \left( \frac{3}{2} - \xi \right)^2 & \text{if } \frac{1}{2} \leq \xi \leq \frac{3}{2} \\
0 & \text{otherwise}
\end{cases}
\]

(31)

B-splines are commonly used in PIC codes because of their simplicity and compact support. Also, when assuming that the electromagnetic fields are piecewise constant about their staggered grid locations, the integral in Eq. 11 is conveniently evaluated and found to be B-splines themselves, of order one higher than the shape function itself. Hence the \(psc\) code uses B-splines of order 1 and 2 to interpolate the electromagnetic fields to the quasi-particle position.

2.4. Time integration

The particle-in-cell method advances both electromagnetic fields and quasi-particles self-consistently. The time integration scheme used in \(psc\) is sketched out in Fig. 2. The figure shows the FDTD scheme (blue), and particle integrator (red), and also their interactions: To update the momentum, the electric and magnetic fields are needed to find the force on a given quasi-particle (black arrows). \(\mathbf{F}_i^{n+1/2}\) exists at the proper centered time to do so, while \(\mathbf{B}^{n+1/2}\) is in principle found by averaging \(\mathbf{B}^n\) and \(\mathbf{B}^{n+1}\). In practice, we rather split the \(\mathbf{B}^n \rightarrow \mathbf{B}^{n+1}\) update into two half steps.

Particle motion also feeds back into Maxwell’s equations by providing the source term \(\mathbf{j}\). The current density is computed from the particles to exactly satisfy the discrete charge continuity equation, which requires knowing particle positions at \(x_i^{n-1/2}\) and \(x_i^{n+1/2}\), which naturally exist, and fed back into Maxwell’s equations (green arrows).

\(psc\) uses two methods to satisfy charge continuity: For 1st-order particles, we use the scheme by Villasenor-Buneman [14], while for 2nd-order particles we follow the method by Esirkepov [15]. \(psc\) also implements some alternating-order interpolation schemes from [16] for improved energy conservation. For a discussion of conservation properties of particle-in-cell codes, see also [17].

When running simulations using single precision, we find that in simulations that run for a very large number of steps, round-off errors accumulate that lead to growing deviations from the discrete Gauss’s Law. \(psc\) implements the iterative method by Marder [18] that dissipates away violations of Gauss’s Law.
3. Overview of the PSC code

The Plasma Simulation Code (psc) presented in this paper is based on the original Fortran code by H. Ruhl [8], but has been largely rewritten. The original Fortran computational kernels (particle advance, FDTD Maxwell solver) are still available as modules, but the code’s overall framework is now written in the C programming language.

The structure of the code is based on libmrc, a parallel object model and library that forms the basis of a number of simulation codes maintained by the author, including the Magnetic Reconnection Code (MRCv3) [19, 20, 21] and J. Raeder’s global magnetosphere code OpenGGCM [22, 23]. We consider libmrc to be a library rather than a framework, because it consolidates commonly used computational techniques in order to avoid reimplementing and maintaining common tasks like domain decomposition and I/O in individual codes. It is designed so that only selected parts of it can be used (e.g., filling of ghost points) in an otherwise legacy code without requiring large changes to the structure of the code overall.

Libmrc is written in C, but supports Fortran-order multidimensional arrays to enable easy interfacing with existing Fortran code. Its basis is an object model quite similar to the one used in PETSc [24, 25, 26] – and libmrc can optionally interface with PETSc to provide linear and nonlinear solvers, etc., though this feature is not used in the psc code. Objects can be instantiated in parallel, i.e., they have an MPI communicator associated with them. Objects instantiate a given class, which essentially defines an interface – in the PIC context, for example, this may be a field pusher which provides two methods to update electric and magnetic fields, respectively. There may be more than one implementation for a given class, which we call “subclass” or “type”. In the example of the field pusher, this may be a single precision or double precision implementation of the FDTD method, or it could potentially encompass methods other than FDTD. In the case of the particle pusher, there are types for first vs second order particle shapes, or a particle pusher that runs on GPUs. Like PETSc objects, the type of a given object can be set at run time – potentially from a command line option, which allows to easily switch out modules in a given run.

The libmrc library provides a number of classes for common computational tasks, e.g., a parallel multi-dimensional field type which is distributed amongst MPI processes and associated coordinates. It handles parallel I/O, currently implemented options include the simple “one file per MPI process” approach as well as parallel XDMF/HDF5 [27] output. As we will explain in more detail in the load balancing section, libmrc fields can be decomposed into many “patches”, where a given MPI process may handle more than one patch – the very same interface is used to support block-structured adaptive mesh refinement, where again a given process handles multiple patches, which are then possibly at different levels of resolution. Libmrc objects maintain explicit information about their own state, e.g. an object knows about member variables that are parameters, so these can be automatically parsed from the command line. This also simplifies checkpoint/restart: Every object knows how to write itself to disk, and how to restore itself, which means that writing a checkpoint just consists of walking down the hierarchy of objects, asking each object to checkpoint itself.

The psc code uses libmrc objects extensively: There are objects for all computational kernels (particles, fields), for particle boundary exchange/filling ghost points, for outputting fields and particles, etc. All these objects are contained within one psc object that represents the overall simulation. To implement a particular case, one “derives” from the psc object, i.e., one implements a particular subclass. In this subclass, one can then overwrite various methods as needed – the create() methods to set defaults for domain size, resolution, normalization, particles species, etc., an init_fields() method to set initial conditions and similarly an init_npt() method to set the initial condition for particles.

The aforementioned objects are primarily used to select particular algorithms, i.e., a second or first order particle pusher has little or no state associated with it but rather just implements a different computational algorithm. The actual simulation state itself is maintained in two additional objects: psc_fields and psc_particles, which are the large distributed arrays that represent the field and particle state. Those objects themselves may actually be implemented as rather different data structures: The particle data may be a simple array of struct in double precision living in CPU memory, but it can also be a more complicated struct of arrays of small vectors in GPU memory in single precision, just by selecting its subclass to be “double” vs. “cuda”.

The flexibility of supporting multiple data layouts is crucial to supporting both CPUs and GPUs in one code, but it also presents a big problem in the algorithms that actually work on that data. For example, for obvious reasons, a CPU particle pusher will not work when the particle data passed to it actually lives in GPU memory and is in the wrong layout. In traditional object oriented programming, the solution to this is to not access data directly, but virtualize
it through methods of the particle object. However, for a high-performance code it is not acceptable to abstract all accesses through virtual (indirect) method calls because of the performance penalties incurred.

Another solution is to only support a matching set of modules – double precision CPU particle pusher with double precision CPU particles and double precision CPU fields. However, this means one essentially has to rewrite the entire code to support, e.g., a GPU implementation, which is a large effort and can easily lead to maintenance problems as CPU and GPU capabilities of the code can diverge.

psc resolves this problem differently: A particle pusher expecting double precision particles on the CPU needs to wrap its computations inside a pair of `particles_as("double")` and `particles_put_as()` calls. The particle data structure returned from the `get_as()` call is guaranteed to be of the requested type, so the actual computation can be performed by directly accessing the known data structures without any performance penalties. Behind the scenes, the `get_as()` and `put_as()` calls perform conversion of the data structures if needed – if the particle data was actually stored as single precision, it would be converted to double precision first, and the result will later, in `put_as()`, be converted back. If the particle data was already of the type requested, `get_as()` and `put_as()` perform no actual work.

The main advantage to this approach is that it is now possible to implement new computational kernels one at a time, while keeping the overall code functional. There is of course a performance penalty for the data layout conversion, and it is typically severe enough, that for production runs, one wants to select a matching set of modules, e.g., particle and field data, particle pusher and field pusher all of the “cuda” type for running on the GPU. Still, the main computational kernels are only a fraction of the code overall, and other functionality like I/O and analysis often occur rarely enough that for those routines the conversion penalty is small, so it is not necessary to rewrite them for the new data types.

It should be noted that while a particle pusher on the CPU looks very different from that on the GPU, so there is little room to share code, a second order pusher on the CPU working on single precision data is very similar to the same in double precision, so in this case we use a shared source file that gets compiled into a single and a double version by using the C preprocessor, so while we end up with two distinct subclasses (“2nd_single” and “2nd_double”), we avoid unnecessary code duplication.

The `psc` code supports a number of additional features, including an approximate Coulomb collision operator (see [28]), periodic and reflecting conducting wall boundary conditions, moving window and boost frame.

As mentioned before, output is typically written as XDMF/HDF5, which allows directly visualizing the data with Paraview, though we typically use custom scripts in Python or Matlab to downscale the resolution and perform specific analyses.

4. Parallelization and load balancing

4.1. Parallelizing particle-in-cell simulations

Due to their dual nature, particle-in-cell simulations are inherently more difficult to parallelize than either purely mesh-based or purely particle-based algorithms. For both mesh-based and particle-based method, a data parallel approach is fairly straightforward, but the two-way interactions between fields and particles in the PIC method requires an approach that takes these interactions into account. In the following, we consider parallelization on a distributed memory machine using the message passing paradigm. Virtually all large supercomputers follow this paradigm as the highest level of parallelism. Lower levels, like shared memory parallelization on a node or small vector instructions on a core can be used in addition and will be discussed later.

In a mesh-based simulation, the typical approach to parallelization is domain decomposition: the spatial domain is subdivided into smaller subdomains, each subdomain is assigned to a different processing unit and processed separately. This works well if the computation is local in space, e.g., a stencil computation with a small stencil, as is the case for the FDTD scheme employed in `psc`. Near the subdomain boundaries, some data points from neighboring subdomains are required to update the local domain, these need to be communicated by message passing and are typically handled by a layer of ghost cells (also called halo regions).

Distributing particles in a data-parallel way is fundamentally rather simple, by dividing a global list of particles into sub-lists and distributing them to processors. However, interactions between particles will need to be taken into
account and depending on their nature can make it rather challenging to find a parallel decomposition that still allows for efficient computation of those interactions, e.g., in the case of the fast multipole method [29].

For particle-in-cell simulations, interactions happen between particles and fields, but not between particles and particles directly. An exception is the implementation of a collision operator, which approximates interactions of close-by particles by randomly picking representative particle pairs that are in close spatial vicinity.

Performance of particle-in-cell simulations is normally dominated by particle-related computational kernels rather than field computations, simply due to the fact that there are typically 100 or more particles per grid cell. This assumption is not always true, though, in particular in a local sense. Simulations of laser-plasma interaction may have a significant fraction of the simulation just represent light waves in vacuum, and simulations of magnetic reconnection like-wise may have spatial regions at low density, which are represented by just a few particles per cell.

Given these constraints, two approaches have typically been used to parallelize particle-in-cell simulations on distributed memory machines:

1. Distribute particles equally between processing units, redundantly keep copies of the fields on all processing units.

2. Use a spatial decomposition of the domain and distribute particles according to which subdomain they are located in, so that particles and fields are maintained together on the same processing unit.

The main advantage to (1) is its simplicity. Particles can be pushed independently on where they are located since all fields are available. As they move, current or charge density is deposited into the corresponding global fields. There are no load balancing issues – particles are distributed to processes equally in the beginning, and since they never move between processes, this balance is maintained. The main drawback in this scheme is that the source terms for Maxwell’s equations need to include contributions of all particles, so a global reduction for each global grid point is required at each time step. Maxwell’s equations can then be solved on one processor and the resulting fields be broadcast to all processes, or the aggregated source field(s) can be broadcast and the computation be performed redundantly on all processing units. The large global reductions severely limit the parallel scalability and limit the applicability of the scheme to at most 100s of cores.

The method (2) overcomes the scalability limitations of the previous scheme and is the approach used in state-of-the-art codes like VPIC [5], OSIRIS [6] and psc as well. Its implementation is more involved – particles will leave the local subdomain and need to be communicated to their new home. The field integration is performed locally on the subdomain and hence scalable. Both field integration and particles moving near boundaries require appropriate layers of ghost cells, and near subdomain boundaries, proper care needs to be exerted to correctly find contributions to the current density from particles in a neighboring subdomain, whose shape functions extend into the local domain.

Relativistic electromagnetic particle-in-cell codes that follow the parallelization approach just outlined generally show excellent parallel scalability. Recently, OSIRIS [6] has shown to scale to the full machine at NSF’s Bluewaters Cray supercomputer. Fundamentally, this is easily explained by the underlying physics: Both particles and field information can propagate at most at the speed of light – since the time step is chosen so that information propagates at most one grid cell per time step, as required for stability of the explicit scheme, most interactions happen entirely locally, and at subdomain boundaries, only nearest-neighbor communication is required.

Other than the complexity of effectively implementing this parallelization method, there is one important drawback, though: It is hard to provide and maintain proper load balancing in this method. As long as the plasma density throughout the simulation remains approximately uniform, the number of particles assigned to each subdomain will be roughly constant and the simulations will perform well. However, in many cases, the initial density distribution may not be uniform, and even if it is initially, in many application areas like magnetic reconnection or laser-plasma interaction, it will not remain that way as the simulation proceeds.

At best, the ensuing load balance will just cause a performance slow-down. At worst, it can lead to the code crashing with out-of-memory conditions as some local subdomains may accumulate more particles than there is memory available to hold them – this problem occurred in some of our bubble reconnection simulations using the original version of psc.

A number of approaches to load balancing PIC simulations have been used in the past. The easiest approach is to just shift partition boundaries, although in general this does not offer enough degrees of freedom to always achieve good balance [8]. A more flexible approach is to partition the domain amongst one coordinate direction first to obtain columns with approximately balanced load. The next step then partitions columns in the next coordinate direction in order to obtain subdomains with approximately easy load. This scheme can work well, but it leads to subdomains
with varying sizes and complicated communication patterns, as subdomains may now have many neighbors [30, 31].

A number of balancing schemes have been proposed in [32], however these have not been implemented in actual PIC simulation codes.

4.2. Performance factors for a particle-in-cell simulation

Factors that determine the performance of a typical particle-in-cell simulation are best explained using performance data from a sample run, as shown in Fig. 3.

The performance measurements are plotted as a function of timestep executed. The thick jagged green, red and blue lines are all measures of the execution time for the particle push (including current deposition). The green curve gives the measurement from the fastest MPI process, the blue curve shows the slowest process, and the red line indicates the average time over all processes. While all measurements coincide in the beginning of the simulation, there is an increasing spread between slowest and fastest processes, while the average performance remains approximately constant. The reason for this becomes clear when considering the number of particles on each process, which are plotted as the thin green, red, and blue lines. These data were rescaled to match the initial particle push time. Initially the number of particles handled by each process are equal in this simulation, but they then debalance with some processes handling fewer than average, others handling more than average. The average particle number itself of course remains constant. The cause for the divergent particle push performance is now clear: Particles move between MPI process boundaries, leaving some processes with more computational work in the particle pusher, and others with less.

In black, we plot the total time per timestep – this number does not vary much between different processes, since at some point processes that finished their work faster still have to wait for others to finish before communication can be completed. The total time per timestep tracks the slowest process. While unfortunate, it is the weakest link that determines overall performance, and that is why an unbalanced simulation can slow down a run very substantially.

More can be learned from the data: Particle push time creeps up slightly until it suddenly falls back down to the expected level every 100 steps. That is because we sort particles by cell every 100 steps in this run – processing particles in sorted order is more cache-friendly, since $E$ and $B$ fields used to find the Lorentz force will be reused for many particles in the same cell before the pusher moves on to the next cell. The cost of sorting can also be seen in the total time per timestep as the small spikes in total time (black) every 100 steps. Additionally, we see larger spikes in total time every 1000 steps. These are caused by performing I/O.

4.3. Load balancing by shifting subdomain boundaries
Germaschewski / 00 (2014) 1–34

rsc originally supported static load balancing by shifting subdomain boundaries with the goal of achieving approximately equal number of particles in each subdomain at initialization time. While this is guaranteed to work well if the domain is decomposed just along one direction, it will not necessarily be possible to achieve good load balance in larger runs that require a 2-d or 3-d decomposition.

Even an initially well-balanced simulation would become significantly unbalanced in our work on bubble reconnection. We implemented a dynamic load balancing step working by the same paradigm, that is, shifting subdomain boundaries to rebalance the particle load. We achieved significant improvement, but we would still observe a slowdown by up to $3 \times$ compared to the start of the simulation.

4.4. Dynamic load balancing using space filling curves

In this work, we present a new approach to load balancing particle-in-cell simulations and investigate the performance costs and benefits.

The idea is easily stated: Given a number of processing elements $N_{\text{proc}}$, decompose the domain into many more patches $N \gg N_{\text{proc}}$ than there are processing elements, and hence have each processing element handle a number of patches, typically 10 – 100. By dynamically shifting the assignment of patches to processing elements, we can ensure that each processing element is assigned a comparable total load.

We will start by demonstrating the idea in a number of artificially set up simple cases. Later on, we will analyze how it performs in a real-world production run.

4.4.1. Example: Uniform density

The basic idea for load balancing by using many patches per processor is demonstrated in Fig. 4. We start out with a case of uniform density, chosen to be 0.1 here. We use 16 MPI processes to run the simulation, and use standard domain decomposition to divide the domain into $4 \times 4$ subdomains, one on each rank. This case is of course trivially load balanced already, since every subdomain is the same size and contains the same number of particles, see Fig. 4(a).

Still, to demonstrate our approach, we divide the domain into $16 \times 16$ subdomains instead, as shown in (Fig. 4(b)). Since there are now many more patches (256) than processes (16), each process needs to handle multiple patches, and it is necessary to define a policy that assigns patches to processes. Fig. 4(c) shows the Hilbert-Peano space-filling curve [33, 34]. Following along the 1-d curve, each patch is visited exactly once. The 256-patch long curve is then partitioned into as many segments as we have processes, in this case we obtain 16 segments of 16 patches each. The segments of patches are then successively assigned to each MPI process. In the end (Fig. 4(d)), we end up with the same spatial decomposition as the standard partitioning using $4 \times 4$ subdomains, but we gained additional flexibility to react to changing loads by moving patches from processes with higher load to neighboring processes with lower load.

This is obviously just one of many possibilities. Choosing a simple row-major enumeration of the patches instead of the Hilbert-Peano curve, one obtains a rather different assignment of patches to processes, as shown in Fig. 5. This particular case also corresponds to a 1-d decomposition by dividing the domain into 16 strips.

The main goal of a load balancing strategy is of course to decompose the domain onto processes in a way that the computational work is approximately equally divided. However, a strategy will only be successful if it does not interfere too negatively with the communication requirements within a simulation step.

Which of the before-mentioned strategies performs better, dividing into $4 \times 4$ vs $16 \times 1$ subdomains is not actually clear. The amount of data to be exchanged is greater in the 1-d case: Each process needs to send 16 patch boundary data both up and down to the next and previous process, respectively, leading to a total of 32. The 2-d decomposition requires sending 4 patch boundaries in each of all 4 direction, which amounts to a total of 16. The 1-d decomposition requires talking with only two neighbors, while in the 2-d case, requires communication with 4 neighbors, and potentially diagonal neighbors, too, which we neglected in this analysis. Which approach works better depends in practice on actual parameters / message sizes and the specific performance of the interconnect network.

For large simulations, a 1-d decomposition is often not an option since the amount of data to transfer becomes excessive, and there may not even be as many grid cells in just one spatial direction as one wants to use processes. It is therefore in general desirable to reduce the amount of data to transfer, which essentially means minimizing the surface of the local subdomain while keeping its volume constant. In 2-d, this means having the local domain to be shaped approximately as a square, in 3-d as a cube. The Hilbert-Peano curve has the property that points that are close
Figure 4. Demonstration of using the Hilbert-Peano curve to assign patches to processes. (a) Standard partitioning into $4 \times 4$ subdomains. (b) Decomposition into $16 \times 16$ patches. (c) Enumeration of the patches using the Hilbert-Peano curve. (d) Assignment of patches to processes, boundaries between processes are delineated by thicker black lines.

Figure 5. Demonstration of using a simple row-major curve to assign patches to processes. (a) Enumeration of the patches using row-major mapping. (b) Resulting assignment of patches to processes.
in regular 2-d or 3-d space are also close, in an average sense, on the 1-d curve. It hence leads to local domains that are clusters in 2-d / 3-d, minimizing off-processor communication. This is the reason why the space-filling curves has been commonly used to load balance block-structured adaptive mesh refinement codes [35, 36, 37], and why, as we will show, it also works well for load balancing particle-in-cell simulations.

Up to this point, we have only considered a case with uniform density, which is trivially load balanced, and so our load-balancing approach does not offer any major benefit here, but as we have shown, it creates good decompositions, essentially the same that one would have chosen in a one subdomain per process approach. There are actually some potential benefits, though, that are worth mentioning: The requirement to have a specific number of processes, e. g., square numbers like 4 x 4 can be abandoned – the same decomposition into 64 patches can be run on 64, 16, 15, or 2 processes. We envision this to be useful feature when dealing with node failures, which are expected to become a more common problem as simulations use an increasing numbers of cores as machine performance moves towards the exascale. If one node, say 32 cores, die in a 250,000 core run, the code would be able to continue the simulation on just 249,968 cores by redistributing patches among the remaining cores – though obviously the data on those patches need to be recovered first, which requires some kind of frequent (possibly in-memory) checkpointing.

While having many small patches on a process means increased synchronization work at patch boundaries, in particular handling ghost points and exchanging particles, most of this work is wholly within the local subdomain and so can be handled directly or via shared memory, rather than by more expensive MPI communication. As we will show later, dividing the work into smaller patches can even have a positive impact on computational performance, due to enhanced data locality which makes better use of processor caches.

4.4.2. Example: Enhanced density across the diagonal

The next example demonstrates the load balancing algorithm in action. We chose a uniform low background density of $n_b = 0.1$ and enhance it by up to $n = 1.0$ along the diagonal of the domain, as shown in Fig. 6. This density
distribution is chosen to be one where the rsc’s former approach to load balancing is not effective at all: After dividing the domain into 4 × 4 subdomains, it is not possible to shift those boundaries in a way that reduces load imbalance, this simulation will always be imbalanced by a factor of more than 7× as shown in Fig. 6(b). Subdomains near the high-density diagonal have an estimated load of 81200, while away from it, the load is as low as 11264.

Patch-based load balancing however works quite well. The resulting decomposition is shown in Fig. 6 by the thicker black lines. It is visible that some subdomains contain only a few patches, including some with a large load (high density, i.e., many particles), while other subdomains contain more patches, but mostly at a low load.

Fig. 7(b) shows the load for each patch, which is calculated for each patch as number of particles plus number of cells in that patch – clearly, this mirrors the particle density. Fig. 7(c) plots the total load per process, calculated as the sum of the individual loads for each patch in that process’s subdomain. It is clear that the load is not perfectly balanced, but it is contained within ±7.5% of the average load of 29300. This is certainly a vast improvement over an imbalance by a factor of more than 7× in the original code.

4.4.3. Load balancing algorithm

The goal of the actual balancing algorithm is quite straightforward: Divide the 1-d space filling curve that enumerates all patches into \( N_{\text{proc}} \) segments, where \( N_{\text{proc}} \) is the number of processes such that all processes have approximately equal load. In order to accommodate inhomogeneous machines, we actually add a “capability” specification. For example, we may want to run 16 MPI processes on a Cray XK7 node. 15 of those processes run on one core each, while the last one is used to drive the GPU on the node. In this case, we would assign a capability of 1 to the first 15 processes, and a capability of 60 to the last process, since the GPU performance is roughly 60× faster than a single CPU core, so we want it to get correspondingly more work.

The balancing algorithm hence divides the space-filling curve into segments that approximately match the capability for each rank – in the simple case of a homogeneous machine, all capabilities are equal and the algorithm reduces to distributing the load equally. The algorithm is described in more detail in Appendix A.

4.4.4. Synchronization points

As previously laid out, a time step in the psc consists of a number of substeps that advance electric and magnetic fields and update particle positions and moments. Substeps depend on previous substeps, often not only within the local domain but near the boundaries also on results from remote processes. Hence, communication is required and introduces synchronization points between processes, which interferes with load balancing the entire step.

While the PIC algorithm we use is naturally staggered in time for both particles and fields, the implementation in the original rsc broke up all but one step into two half steps so that the timestep would start with all quantities known at time \( t^n \), and advance them all to time \( t^{n+1} \), as shown in Fig. 8.

Communication occurs at 5 different points during the timestep as indicated, separated by computational kernels on either field or particles. Communication is implemented using non-blocking MPI send and receive calls; the time while messages are in flight is used to exchange field boundary data and particles between local patches that do not require inter-process communication. However, these communications still introduce synchronization points. A given process will not be able to, e.g., update the fields until current density data has been received from neighboring processes. Neighboring processes won’t be able to send these data until they finished pushing their local particles. The consequence is that it is not enough to just balance the total computational work, which includes both particle and field work. It is rather necessary to balance both particle work and field work individually between all processors. However, this is impossible unless the number of particles per patch is constant, in which case load balancing is trivial, anyway.

In practice, experiments showed that for typical cases, using our approach to load balancing still worked quite well because performance is dominated by particle work, with field work being comparatively fast, so that imbalance in the field work does not cause a great loss in performance. We set up the load balancing to equally distribute the number of particles that each process handles, up to the patch granularity. In a typical case, we observed a slow-down of particle-dependent kernels by about 15% over the cause of the run, which is consistent with the 15% deviation in particle number balance the algorithm achieved. The overall performance, however, would slow down by 30%. Using the previous approach to load balancing by shifting process boundaries, we observed a slow-down by 200% in the same case, so this was still a large improvement. It does, however, show that as we balance the particle load well,
the field load becomes imbalanced and creates an additional loss of performance that manifests itself in the overall timestep slow-down.

With careful consideration, it is possible to improve balancing to include both particle and field work. The basis for the updated load balancing is to rewrite the time step closer to the natural time-staggered form in our numerical algorithms. In the new algorithm, we start a time step with the quantities known as $E^n$, $B^n$, and $p^n$, and propagate them to $E^{n+1/2}$, $B^{n+1}$, $x^{n+1/2}$, and $p^{n+1}$ as shown in Fig. 9.

\[
\begin{align*}
\text{push_field_E_half();} & \quad \text{// } E^n \rightarrow E^{n+1/2} \\
\text{fill_ghosts_E();} & \quad \text{// communicate} \\
\text{push_field_B_half();} & \quad \text{// } B^n \rightarrow B^{n+1/2} \\
\text{fill_ghosts_B();} & \quad \text{// communicate} \\
\text{foreach(particle prt) \{} & \quad \text{// communicate} \\
\text{\quad push_particle_x_half(prt);} & \quad \text{// } x^n \rightarrow x^{n+1/2}, \text{ save for current} \\
\text{\quad push_particle_p(prt);} & \quad \text{// } p^n \rightarrow p^{n+1} \\
\text{\quad push_particle_x_half(prt);} & \quad \text{// } x^{n+1/2} \rightarrow x^{n+1} \\
\text{\quad push_particle_x_half_temp(prt);} & \quad \text{// } x^{n+1} \rightarrow x^{n+3/2}, \text{ use for current, then disregard this update} \\
\text{\quad deposit_j();} & \quad \text{// charge conservative current deposition using } x^{n+1/2} \text{ and } x^{n+3/2} \\
\text{\}} & \quad \text{// communicate} \\
\text{exchange_particles();} & \quad \text{// communicate} \\
\text{push_field_B_half();} & \quad \text{// } B^{n+1/2} \rightarrow B^{n+1} \\
\text{fill_ghosts_B();} & \quad \text{// communicate} \\
\text{add_and_fill_ghosts_j();} & \quad \text{// communicate} \\
\text{push_field_E_half();} & \quad \text{// } E^{n+1/2} \rightarrow E^{n+1} \\
\text{fill_ghosts_E();} & \quad \text{// communicate} \\
\end{align*}
\]

\[
\begin{align*}
\text{push_field_B_half();} & \quad \text{// } B^n \rightarrow B^{n+1/2} \\
\text{foreach(particle prt) \{} & \quad \text{// communicate} \\
\text{\quad push_particle_p(prt);} & \quad \text{// } p^n \rightarrow p^{n+1} \\
\text{\quad push_particle_x(prt);} & \quad \text{// save } x^{n+1/2} \text{ for current, then } x^{n+1/2} \rightarrow x^{n+3/2} \\
\text{\quad deposit_j();} & \quad \text{// charge conservative current deposition using } x^{n+1/2} \text{ and } x^{n+3/2} \\
\text{\}} & \quad \text{// communicate} \\
\text{push_field_B_half();} & \quad \text{// } B^{n+1/2} \rightarrow B^{n+1} \\
\text{exchange_particles();} & \quad \text{// communicate} \\
\text{fill_ghosts_B();} & \quad \text{// communicate} \\
\text{add_and_fill_ghosts_j();} & \quad \text{// communicate} \\
\text{push_field_E();} & \quad \text{// } E^{n+1/2} \rightarrow E^{n+3/2} \\
\end{align*}
\]

Every quantity is now updated only once, by a full step, with the exception of $B^n \rightarrow B^{n+1/2} \rightarrow B^{n+1}$, which is required at the intermediate time to interpolate the Lorentz force acting on particles.

Other than the drawback of having to handle quantities at different time levels at the initial condition and output, the scheme in its natural form actually presents a number of advantages: Less computational work is required due to combining half steps into full steps. The discrete version of Gauss’s Law is actually required to be satisfied exactly at half-integer time levels, which can now be satisfied more easily in the initial condition. Particles are exchanged according to their positions at time $n+1/2$, so particles are guaranteed to actually be inside the local domain at the time that the electromagnetic fields are interpolated to the particle position. This is in contrast to the old scheme, where a particle already moved a half time step, and hence might have left the local domain. This means that fewer levels of
Most importantly, the rewritten scheme can be recast to have only a single synchronization point. We now do all communication after the particle push and completing the second half step to update \( \mathbf{B} \). At that point, particles are ready to be exchanged as their positions have been advanced to \( x^{n+3/2} \). The current density \( j^{n+1} \) has been calculated and can be added up and used to fill ghost points. After we also fill ghost points for \( \mathbf{B}^{n+1} \), enough information is available to perform the remaining field updates all the way to the next particle push, while still providing the necessary ghost cell data for the field interpolations in that next particle push.

As we show in detail in Appendix B, for second-order particle shape functions, two layers of ghost points for the fields \( j, \mathbf{E}, \) and \( \mathbf{B} \) are sufficient to perform a full time step, including field and particle updates, without any further communication.

### 4.4.5. Calculating the load function

As the time integration now requires only a single synchronization point, as just described, it is sufficient to balance the total computational load per timestep, including both particle and field updates.

The load balancing algorithm requires as input an estimate of the load \( L_p \) associated with the computations occurring on each of the patches, \( p \). A promising candidate is a function of the form

\[
L_p = N_{\text{particles}}(p) + CN_{\text{cells}}(p),
\]

as the work in the particle push scales with the number of particles being pushed, while the field updates scale with the number of grid cells. The constant \( C \) can be used to adjust the weighting between particle push and field updates, as the work of pushing one particle is not expected to be equal to the work of advancing the fields in one cell. As we will show in a case study later, this simple approximation works quite well to achieve good balance, however it requires an appropriate choice of the parameter \( C \), so we also pursued an alternate approach of actually measuring the time spent in the computational kernels.

### 4.4.6. Performance cost of subdividing the domain

Subdividing the spatial domain into many more patches than processing units can improve the load balance of a particle-in-cell simulation dramatically. On the other hand, besides the increased complexity in implementing the approach, there are also potential performance costs: (1) Rebalancing the domain, including moving patches to other processes in order to improve load balance takes up processing time itself. (2) Handling many small patches on a process rather than just one large patch creates costs both in managing those patches, in increased computational work, and in increased communication both between local patches and between local and remote patches.

While the cost of rebalancing (1) is substantial, typically equal to a couple of regular timesteps, rebalancing only needs to be performed occasionally, typically every 100 – 500 steps, so this cost gets amortized over a large number of steps. As shown in a case study below, we find that the cost of rebalancing only adds 1–2% to the cost of each timestep.

In order to address issue (2), we performed a number of simulations at identical physical and numerical parameters, while varying the number of patches that the domain is divided into. We used physical plasma parameters motivated by the bubble reconnection simulations that will be described in more detail later, but changed the initial condition to be a uniform plasma. Using a bubble simulation directly is not feasible, since the initial density is non-uniform, and even if the simulation is initially load balanced, it does quickly de-balance. The initially uniform plasma remains uniform, which means that no actual load balancing is required, so we can exclude the impact of growing imbalance and focus just on the performance cost of varying the partitioning into patches. Our example case is run at a resolution of 1200×800 grid cells using 600 cores, using 200 particles per cell per species. We start with a simple decomposition into 30×20 patches, which means that every process handles only a single patch, of size 40×40 grid cells. In this case, there is no additional cost from subdividing the domain into smaller patches, it is just standard domain decomposition. We then increase the number of patches the domain is divided up to 240×160, i.e. 64 patches per process of size 5×5. Table I lists the parameters for the simulations we performed.

Fig. 10 plots average performance data vs the number of patches per MPI process. The first thing to notice is that the variation of the total time per time step (red curve with circles) is fairly small, it varies between 179.4 and 185.8, that is the slowest case is less than 4% slower than the fastest. The plotted data were obtained by averaging the
Table 1. List of runs to study the performance cost of dividing the domain into many small patches.

| Global number of patches | Patches per process | Patch size |
|--------------------------|---------------------|------------|
| 30 × 20                  | 1                   | 40 × 40    |
| 60 × 40                  | 4                   | 20 × 20    |
| 120 × 80                 | 16                  | 10 × 10    |
| 150 × 100                | 25                  | 8 × 8      |
| 240 × 160                | 64                  | 5 × 5      |

The total time per time step actually initially goes down (solid red curve) as the number of patches is increased up to 16 patches per process, and then goes up again. This might seem surprising, but is easily explained by the limited cache memory available to each core. As the patch size decreases to 10 × 10, field data is more effectively cached – all particles in any one patch are processed before moving on to the next patch, so the fields will quickly become resident in cache, allowing to push all particles in that patch without further access to field data in main memory. We confirm this by looking at the total computational work per timestep (blue curve with squares) and the particle push time (green curve with triangles). It is clear that the fast total time per time step originates in the particle pusher. The clearest evidence comes from re-running the simulations with particle sorting enabled every 50 steps (dashed lines). Since particles are now always sorted, fields are accessed in a structured manner independent of the patch size, and we consistently see fast performance. With sorting enabled, performance is now in fact fastest when using only one patch per processor, avoiding the additional overhead of multiple patches. However, the performance cost of using many patches is very small. Even at 64 patches / core, which corresponds to a very small patch size of 5 × 5 grid cells, the time step is only 1.5% slower than in the fastest case. This performance loss, small as it is, can be traced down to two causes: (1) The total computational work per timestep (blue curve) increases – this is mainly caused by the additional ghost cells that Maxwell’s equations are solved in (as we laid out before, some ghost cell values are calculated rather than communicated to avoid additional communication and synchronization points). (2) The time spent exchanging particles and ghost cell values increases. This time can be seen in the figure as the difference between the blue and the red curve, and it clearly increases as the number of patches per core increases. Most of the additional communication occurs between patches on the same MPI process, where it is handled by simple copies, rather than actual message...
From the data we presented here, it is clear that the overhead of using many patches per process remains insignificant as long as the patch size is not made unreasonably small.

4.5. Performance study: Load balancing a bubble reconnection simulation

Our first example for studying the utility and performance of the space-filling curve based load balancing scheme in psc is a simulation of magnetic reconnection of laser-produced plasma bubbles. More detail about those simulations and the underlying physics can be found in [38, 28]. The runs presented below used a background plasma density of 0.1 and a peak density of 1.1 in the center of the bubbles in normalized units. The plasma bubbles expand into each other, driving magnetic reconnection; in the process, the peak density moves from the bubble center to the edge. We used a mass ratio of \(m_i/m_e = 100\). The domain size is \(60d_i \times 40d_i\), we use 2400 \(\times\) 1600 grid cells. All runs were performed using 2048 cores of the Cray XE6m supercomputer Trillian located at the University of New Hampshire.

4.5.1. Static balancing

We now study the degree of load balancing that is achieved as we vary the number of patches per core. All runs were using the same parameters and same resolution, and were performed on 2048 cores. We varied the number of patches that the domain is divided into from \(75 \times 50\) through \(150 \times 100\) to \(300 \times 200\), which means the average number of patches per core varied from 1.83 to 29.3. For details described in the following, see table 2.

Our load balancing algorithm works based on shifting whole patches between processes to achieve approximately equal distribution of the total load. This implies that we will not be able to achieve perfect balance (neither is that required in practice), but rather balance at the granularity of moving single patches, which requires that the load of a single patch is “small enough” to reduce the imbalance to a given amount. The table shows how the average number of patches per core translates to actual number of patches per core in the simulation. In the case of an average number
of 29.30 patches per core, some processes are assigned just 7 patches, others as many as 72. This should not be surprising – the density in this simulation varies by a factor of 10, so 7 patches in an area of high density contain about as many particles as 70 patches in low density areas.

The achievable balance is limited by those processes that have the fewest patches (with correspondingly high loads), as moving changing, e.g., the number of patches from \( m \) to \( m + 1 \) increases the load by approximately \( 1/m \), similarly for decreasing the number of patches from \( m \) to \( m - 1 \). In the particular case discussed, \( m = 7 \), so we expect a granularity of \( 1/7 \approx 14\% \), which is consistent with the observed remaining imbalance of \(-12\% \), +13\%. We hence find that in order to achieve good balance, we need to have a minimum of at least several (5-10) patches on each process.

Fig. 11 shows measurements of load imbalance for the 3 runs on the left hand side. Red, blue and green bars, which are shown on top of each other, measure the total time spent in computations related to particles and fields. The red bar indicates that time spent on the fastest process, the top of the blue bar is the average time spent over all processes, and the top of the green bar indicates the time spent on the process that took the longest. This numbers are indicative of the corresponding minimum, average, and maximum load, respectively. The average load, shown by the blue bars, is nearly the same in all cases – that is expected, as the total load for the three simulations is essentially identical, only its distribution changes. It is quite clear that the the load imbalance, the spread between red and green bar is reduced as the number of patches is increased, which makes the load balancing more and more fine grained. It should be noted, though, that even having just 1.8 patches per core on average already reduces load imbalance dramatically: A totally unbalanced run using equal sized patches, one per core, would have a load imbalance of +900\%, so an imbalance of +123\% is a vast improvement. The yellow bars in the plot, barely visible, indicates the difference between the total time per time step and computation time on the process with the max load. The amount of yellow hence shows how much time is spent in ghost point exchanges and communication. The time is generally small and has little impact on overall performance, though it can be seen that it increases, as expected, as the number of patches rises.

The right hand side of Fig. 11 shows the corresponding data at a late time near the peak of reconnection. The geometry of the bubbles has changed drastically by then, so even an initially well-balanced simulation has become very unbalanced, by up to 500\%. The imbalance at the late time is only weakly dependent on the initial balance, and is a clear indication that static load balancing just at the start of the simulation is not sufficient to maintain good performance. Fig. 12 shows the min/average/max computation times as the simulation with 300 × 200 patches proceeds, the strong debalancing over time is clearly visible until about 15000 steps, at which point the reconnection is basically complete, the dynamics slows down and the density profile starts spreading out, which slightly improves load balance again.

4.5.2. Dynamic balancing

Clearly, static load balancing at the beginning of the simulation is not sufficient to maintain good load balance over the run of our bubble simulations. In Fig. 13 we show the timing data for the same simulation that was used in Fig. 12, but with dynamic rebalancing turned on every 200 steps. The total time per timestep over all steps in the simulation is indicated by black dots; for reference, the corresponding data for the statically balanced case are also plotted using grey dots. Very good load balance has been achieved by our dynamic load balancing scheme, the execution time per timestep remains almost constant. The computational load is nicely evened out. The slowest processor (blue dots) and fast processor (green dots) remain close to the average computational load (red load) throughout the entire run. The plot also confirms that the time spent in communication routines, i.e., the difference between blue and black line, remains only a small fraction of the entire time spent per timestep.

As mentioned before, the analytic formula to estimate the load of processing a given patch requires appropriate choice of a weighting parameter \( C \) that indicates relative performance of particle pushes and field updates. We performed the same run choosing \( C = 1, 2, 3 \). Best performance and balance was achieved at \( C = 2 \), this run is shown on the left of Fig. 13.

The parameter \( C \) depends on a number of factors including the architecture of the particular machine, cache sizes, the order of the shape function, and the simple form also does not take into account that some ghost cells are updated, too. We therefore also tried an alternative, instead of estimating the load analytically, it is estimated based on past performance measurements. The code keeps track of the time spent in computational kernels for each patch separately since the last rebalancing. We purposefully accumulate many measurements so that outliers and noise in the measurements can average out. The timing measurements include all field computations and all particle computations including the pre-exchange step that sorts out all particles that left the local patch into a separate list.
The right panel of Fig. [13] shows the performance of a run based on actual timing measurements as the run proceeds. It can be seen that in comparison to the simple load function, this approach introduces some small irregularity in the performance results, but it still performs nicely and has the advantage that it doesn’t require figuring out a proper choice of the parameter $C$ in advance of a run.

Further investigation into the irregularities shows that the timing measurements are not exactly reproducible – there are occasional outliers, patches that get processed more quickly or more slowly than what previous measurements indicate, which can be caused by a number of factors including cache effects and errors associated with the timing measurements (we measure wallclock time, so if a core switches to kernel space or another user process, this time is still accounted for as patch processing time). A consequence is that the load balance achieved is not as good as estimated – the code may balance the measured loads to within 6 or 7% of the average, but subsequently, the actual spread may be closer to 10%.

The plots also show, as faint black dots, the impact of the actual rebalancing process. Every 200 steps, a new assignment of patches to processes is calculated and data is then communicated correspondingly. This shows up as spikes every 200 steps, as these steps take about 500 ms rather than the usual 140 ms. This additional cost of load balancing is amortized over the following 200 steps, though, adding effectively about $(500\text{ms} - 140\text{ms})/200 = 1.8\text{ms}$ to the timestep duration, which is small compared to the 140 ms that the actual steps takes – a small price to pay for the dramatically improvement in performance as the simulation remaining balanced. Fig. [14] shows the actual distribution of patches to processors in the run just described. It is easy to see how some processors handle regions of many low-density patches, while others work on just a few high-density patches. The work distribution is dynamically adapted as the simulation proceeds.

4.6. Performance study: Load balancing a particle acceleration study

We performed another performance study on a simulation of particle acceleration in magnetic reconnection. The simulation parameters used here are motivated by the work in [2]. The initial condition is a Harris sheet with a central density of $n = 1$ and a background density of $n = 0.05$. The domain size is $204.8d_i \times 51.2d_i = 4096d_e \times 1024d_e$ using a $16384 \times 4096$ cell mesh. The mass ratio is $m_i/m_e = 400$. We used 200 particles per cell to represent $n = 1$ and the 2nd order particle shape function.

This simulation poses substantial challenges for maintaining load balance. Initially, the plasma density away from the current sheet is $20\times$ lower than in the central current sheet, shown in Fig. [15a]. Consequently, the load per patch is similarly unbalanced, varying from 1344 to 26592 in the center, a ratio of 19.8x. The simulation divides the domain into $1024 \times 256 = 262144$ patches of $8 \times 8$ cells, and is run on 8192 cores. This averages to 32 patches / core, but...
Figure 14. Load balancing data for the bubble simulation at (a) step 0 and (b) step 15000. Show are density (top), computational load for each patch (center), and computational load for each process (bottom).
due to the strong imbalance, the achieved load balance is limited to ±32%, which is still a vast improvement over the original 20x.

As the simulation proceeds, load balancing becomes even more challenging since reconnection proceeds, some plasmoids form and get ejected, and almost all of the density gets concentrated in few islands while the plasma in the rest of the domain becomes very tenuous.

After 70000 steps (see Fig. 15(b)), the load varies from a minimum of 228 to a maximum of 52011, an imbalance of 228x. The code handles the increasing load balance quite well, though, balancing process load to within ±50%.

Fig. 16 shows the timing information throughout the run. The black curve again indicates the execution time of an entire timestep – it only increases mildly at around 50000 steps, which is when plasmoids start forming, and then remains approximately flat again thereafter. We again see that the black line is only slightly above the blue line, the computational time used on the process with the heaviest load. This indicates that communication / boundary exchange is not a significant performance factor. The spread of the green (fastest) and blue (slowest) process from the average (red) indicates that load balance is not perfect, in fact these mirror the aforementioned ±32%, ±50% deviation from perfect balance. This spread is certainly acceptable considering that the underlying imbalance is greater than 200x. Maybe even more important than the performance itself is the fact that the load balancing also balances memory requirements, preventing single nodes from running out of memory and crashing the entire simulation. This is a very important consideration for GPU based simulations, as GPUs come with limited memory resources: For example, on the Cray XK7 a node has 32 MB CPU memory, but only 6 GB GPU memory.

5. GPU algorithms

5.1. Introduction

Originally designed to speed up processing and display of images on computer screens, graphics processing units (GPUs) have in recent years evolved into powerful processors that can accelerate general computationally-intensive tasks. Optimized for highly parallel computations, they have shown potential to accelerate numerical simulation codes significantly. The two fastest supercomputers in the world, according to the TOP500 list from June 2013 [39], and many others down the list derive their computing capabilities from accelerator technology like Nvidia GPUs and Intel’s Many-Integrated-Cores (Xeon Phi) processors. DOE’s supercomputer Titan consists of 18,688 nodes with one 16-core AMD 6274 CPU and one Nvidia Tesla K20X GPU each. About 90% of its theoretical capability in floating point operations per second are provided by the GPUs, which clearly shows that only GPU-enabled codes can get close to using all of its potential.

GPUs and other accelerator technologies hold great promise in enhancing scientific discovery through their increased computational power, but they come with challenges to adopt codes to efficiently use their theoretical capabilities. Different programming models exist, from using provided libraries (e.g., cudaBLAS) through relatively minor changes to existing code using an annotation-based programming model like OpenACC to rewriting computational kernels from scratch using CUDA C/C++ or CUDA Fortran.

Most of the existing work [40, 41, 42, 43] on porting particle-in-cell codes to GPUs focuses on basic algorithms, e.g., electrostatic simulations where only the charge density needs to be deposited back onto the grid and work is performed more in proof-of-concept codes rather than full-scale production codes. psc uses charge-conservative current deposition, as also discussed in [43]. We added GPU capabilities to the psc code based on its existing modular architecture that enabled us to implement new computational kernels as well as new underlying GPU data structures within the existing code. In the particle-in-cell method, the important computational work generally scales linearly with the number of particles and is of low to moderate computational intensity, which means that copies between main memory and GPU generally have to be avoided to achieve good performance. Therefore, the simpler programming approach of keeping most of the code and data structures on the host CPU unchanged and using the GPU to just accelerate selected kernels is not feasible.

psc has been run on thousands of GPUs and achieved a performance of up to 670 million particles / second per node on Titan using the Nvidia K20X GPUs using first order single precision particles in a 2-d simulation, which is more than six times faster than its performance on the CPUs only.
Figure 15. (a) Initial Harris sheet and (b) reconnection in progress after 70000 steps. Shown are the density on a logarithmic scale (top), and the load for each patch and the subdomain boundaries (bottom).
Figure 16. Timing data for the particle acceleration run, with dynamic load balancing performed every 100 timesteps. The execution time for the entire timestep is plotted in black. The time spent in actual computation is shown in blue, red and green for the slowest, average and fastest process, respectively

Figure 17. Processing particles on the GPU. (a) If particles are unsorted (left): to interpolate electromagnetic forces to the particle position, field values need to be loaded from global memory for the grid points surrounding each particle. (b) If particles are sorted (right) by super-cell (red square), all required field values can be loaded into shared memory (red circles) first and then be used to interpolate forces for all particles.
### Table 3. Performance of 2nd order particle push on Nvidia Tesla C1060 card using 200 particles per cell as a function of super-cell size.

| Kernel                  | Performance (particles / sec) |
|-------------------------|-------------------------------|
| 2d push, no field caching | $199 \times 10^6$            |
| 2d push, cache $1 \times 1$ blocks | $624 \times 10^6$          |
| 2d push, cache $2 \times 2$ blocks | $901 \times 10^6$          |
| 2d push, cache $4 \times 4$ blocks | $972 \times 10^6$          |
| 2d push, cache $8 \times 8$ blocks | $962 \times 10^6$          |
| 2d push, cache $16 \times 16$ blocks | $531 \times 10^6$         |

5.2. Particle Advance

The particle advance consists of two components: Stepping the equations of motion in time, and calculating the currents from the particle motion, which act as a source term to Maxwell’s equations.

The equations of motion can be solved for each particle independently of all the other particles, making the particle push itself highly parallel. The particles do not interact directly, but via the electromagnetic fields only. Each particle push requires finding electromagnetic field values on grid locations in the vicinity and interpolating them to the particle location. Whereas many particles may access the field values in the same grid location concurrently, these are read-only accesses that do not require serialization.

For performance reasons, it is advisable to avoid accessing the field values in random order, as this would introduce a memory access bottleneck. On traditional CPUs, many cache misses will occur, and performance can degrade greatly. Figure 17(a) depicts this approach. For each particle, field values from surrounding grid points need to be loaded to find the Lorentz force. As particles are accessed in random order, these accesses are all over the place. Similarly, current density updates go back to main/global memory in random order. Avoiding these random accesses is the reason why particles are kept approximately sorted by grid cell in most PIC codes. As those sorted particles are accessed sequentially on a single process (or thread), most of the field values will already be in cache.

On GPUs, it is most efficient to sort particles and use shared memory for field access. It is generally sufficient to sort particles by super-cell (e.g., a block of $2 \times 2$ cells in 2-d). Fig. 17(b) shows all particles in a given $2 \times 2$ super-cell – in this example, we show just 4 particles for simplicity, though we typically have many more particles in a super-cell. A CUDA threadblock will then load all field values needed into shared memory first, then push all particles in that super-cell, before continuing on to the next super-cell and repeating the process. A super-cell typically still contains thousands of particles. The cost of first loading the fields from global memory is amortized over those thousands of particles.

An initial benchmark of a 2-d, second order particle push on a Tesla C1060 card showed that we can achieve a performance of 200 million particles / second even without caching fields in shared memory, which is quite impressive considering that a typical single CPU core only achieves about 3 million particles / second. Using the GPU shared memory provided a substantial increase in performance which depends on the chosen size of super-cell, topping out at 972 million particles per second using a $4 \times 4$ super-cell, as shown in Table 3.

On the latest Nvidia K20X hardware, using a 1st order particle shape, we achieve a performance of 2290 million particles per second for the particle push itself, which includes interpolation of the electromagnetic fields and updating positions and momenta, but not the current deposition.

5.3. Current deposition

Implementing a highly-parallel GPU current deposition algorithm is substantially more difficult than the particle push. Each particle, as it moves, contributes to the current density, which is accumulated on the field mesh. In doing so, we have to find the corresponding current for each particle and deposit it onto nearby grid locations according to the particle’s shape function. The deposit is a read-modify(add)-write operation. As multiple particles are processed in parallel, they may modify the same grid value concurrently, depending on their exact position in the domain. It is therefore necessary to introduce synchronization to ensure that multiple threads do not update the same value at the same time and lose contributions. Our experience mirrors that of [43]. We found that the most efficient way to handle this problem is to use `atomic_add()` in shared memory on Nvidia Fermi and newer compute architectures.
It is still important to avoid getting many memory conflicts. Even though \texttt{atomic\_add()} guarantees correctness, performance slows down when frequent memory access collisions occur. Sorting only by a not-too-small super-cell rather than ordering by cell helps. The GPU will process 32 particles in one warp – if all of those particles are in the same cell, chances that multiple threads want to update the same current density value are much higher than if those 32 particles are randomly spread inside, e.g., a 4 × 4 super-cell. We actually observed the current deposition speed up by a factor of 2 during the initial phase of the simulation as particles, which are initially ordered by cell when setting up initial conditions, randomize by their thermal motion and mix within the super-cells. Conflicts, of course, could be avoided entirely if every thread was to write into a private copy of the current density field, in this case it is not even necessary to use the atomic updates – though in the end, it is necessary to add up values from all private copies, but that is generally quite fast. There is, unfortunately, not enough shared memory available in current GPU hardware to be able to support that many private copies, even in 2-d, without severely reducing the occupancy of the GPU multiprocessors, which greatly reduces performance. It is, however, possible to maintain a limited number of private copies in shared memory, which reduces conflicts in the atomic operations while maintaining good occupancy. In 2-d, we typically use 16 redundant copies, which increased performance almost two-fold.

We also tried alternative approaches using reductions in shared memory rather than the atomic add, but obtained our best performance for an atomic Villasenor-Buneman charge-conservative current deposition \cite{14}. On the Nvidia K20X, we achieve a performance of 1970 million particles per second for the current deposition. Our implementation of the current deposition avoids thread divergence, i.e., different threads do not execute different if branches, though some threads may, via predicates, skip instructions.

The combined performance of particle push and current deposition is up to 1060 million particles per second. These two algorithms comprise all the particle computational work, and the Maxwell solver is typically only a small fraction of the overall computational effort. However, as mentioned before, these algorithms require particles to be sorted by super-cell, and in a parallel run involving multiple GPUs, it is also necessary to exchange particle and field values across computational domains, which means an overall performance of greater than 1 billion particles per second and GPU could not actually be attained.

5.4. Sorting and Communication

On cache-based architectures, sorting particles is used to maintain cache-friendly access patterns, however having particles slightly out of order will only incur a small performance penalty, so the cost of sorting is often amortized over tens of timesteps.

On the GPU, we essentially use shared memory as a user-managed cache, e.g., loading electromagnetic field data for a super-cell worth of particles into shared memory first, then processing all those particles directly using the field data in the shared memory. It is hence necessary to have particles sorted by super-cell before performing the particle push, because having particles out of order does not just incur a performance penalty, but will rather lead to incorrect results or crashes.

Therefore, particles need to be sorted at each timestep. Like on the CPU, we also need to handle particle exchange across patch boundaries: particles that leave the local patch need to be moved to their new home patch – which may be on the same GPU, but it also might be a patch on another node, requiring MPI communication. In order to reduce the number of memory accesses, we handle sorting and boundary exchange in one fused algorithm.

The basis for this algorithm is a high performance GPU sort, which in itself is a complex problem, due to the need to exploit a lot of parallelism to achieve good performance on GPUs.

The good news is that efficient algorithms are available for sorting particles by cell: Sorting \(N\) particles into \(M\) cells a counting sort takes \(O(N + M)\) steps, which in the typical case of many particles per cell becomes \(O(N)\). The usual limit of \(ON\log N\) does not apply, since the sort is not comparison-based. Here and in the following, we talk about sorting \textit{by cell}, which may refer to sorting by the actual unit cell in the underlying field mesh, but the algorithms are exactly the same when sorting by super-cell, which is what is actually used in the GPU algorithms, as particles being ordered by super-cell is sufficient. It is also possible to achieve an ordering by cell and super-cell simultaneously by proper choice of the cell index used as sort key.

The basic counting sort algorithm itself is not well suitable for GPUs due to its lack of parallelism, but it can be used as a building block. We also use counting sorts for sorting particles on the CPU.
The input for the counting sort is the unsorted array of particles: \( P_n \), where \( 0 \leq n < N_{\text{particles}} \). The sort is performed out-of-place, i.e., every particle gets moved into the right position in a new array \( \tilde{P}_n \) such that this new array is sorted.

The first step is to calculate the target cell index \( C_n \) for each particle:

```python
for n in range(0, N_particles):
    pos = [int(P[n].x * dxi),
           int(P[n].y * dyi),
           int(P[n].z * dzi)]
    C[n] = cell_map_3to1(pos)  # map to 1-d cell index
```

The function `cell_map_3to1()` maps the 3-d cell index to a 1-d index that is used as the sort key. The most straightforward choice is a simple column-major ordering:

\[
C_n = (pos_x \cdot M_y + pos_y) \cdot M_x + pos_z,
\]

where \( M_x, M_y \) are the number of grid cells in \( x, y \) direction, respectively. It is, as mentioned earlier, possible to use a more complicated map that achieves ordering by cell as well as super-cell after sorting by using some bit-level manipulations. This is useful if one wants to maintain ordering by super-cell for the GPU particle push, but also wants ordering by cell for the collision algorithm.

The actual counting sort consists of three stages: (1) count the number of particles in each cell, (2) find offsets for the first particle in each cell and (3) move particles into their new, sorted positions.

```python
for n in range(0, N_particles):
    cnts[C[n]] += 1

for m in range(0, N_cells):
    n = cnts[m]
    cnts[m] = cnt
    cnt += n

for n in range(0, N_particles):
    Ptilde[cnts[C[n]]] = P[n]
    cnts[C[n]] += 1
```

While algorithmically perfectly scalable, the counting sort algorithm is inherently serial. The building blocks are fairly simple (count and scan), though, and efficient GPU algorithms have been developed [44, 45], whose implementations are too complex to describe in detail here. The underlying idea is best explained for the counting algorithm: Multiple threads can read cell indices for multiple particles concurrently, however updating a common `cnts` array would require atomic accesses. Instead, the `cnts` array is replicated for each thread, removing the data hazard. At the end of counting phase, the replicated counts are then summed up across threads to find the final counts in the `cnts` array. It is not feasible to actually replicate the large `cnts` array many times, in fact one wants to have the replicated arrays fit into shared memory to avoid the large latencies in repeatedly accessing global memory.

To decrease the size of the replicated count arrays, one needs to reduce the space of possible keys. The radix sort algorithm achieves this by splitting the sort into multiple phases. In decimal notation, one could sort a list of numbers first by least significant digit (1s) only. The partially sorted numbers are then sorted by the next digit (10s), and so on, until all digits have been processed. The method requires a stable sort for the partial sort, such as the counting sort described above. The Thrust library implements this radix sort [44], sorting 4 bit “digits” at a time using essentially a parallel counting sort for each digit. The parallelization for each digit actually involves a second level: The global list of keys is first split amongst GPU threadblocks. Each thread block then counts digits for its assigned partial list, and writes the result back into individual count arrays in global memory (bottom level count). A top-level count/scan, run on a single threadblock, then calculates offsets for each digit for each threadblock, and finally a second bottom level scan finds the target position for each element of the list and moves it there.
5.4.1. Implementation of the sort in psc

psc’s GPU sort is based on the thrust implementation of the radix sort. Rather than moving particles into their new sorted order, though, we just obtain a list of particle target positions, though – that means that at every stage of the radix sort, only the 32-bit particle position has to be written, rather than moving the whole particle. In fact, in the most integrated and optimized version of the GPU algorithms, the actual particle move into the new position only occurs during the subsequent particle push, where the particle is read, updated and written, anyway.

We also achieved a large improvement over the original radix sort by exploiting a particular property of the explicit relativistic particle-in-cell method: In a single timestep, a particle can move at most into a neighboring cell, as its speed is limited to the speed of light, and the timestep is limited by the corresponding CFL condition. Hence, particles which are initially sorted by cell will still be “almost ordered” after the timestep, that is, before the next sort. In fact, for each particle in 2-d, there are only 9 possibilities: The particle remains in the same cell, or into any of the adjacent cells, including diagonals. We add one other option: The particle leaving the current patch. Those 10 possibilities can be represented by just 4 bits, allowing us to perform just a single radix sort pass for count/scan at the bottom level. In 3-d, there are 27 + 1 possibilities, which again can be handled by a single efficient 5 bit pass, though we have not implemented this case yet.

The advantage of working with a single “particle direction” tag, rather than the explicit target super-cell index becomes clear considering that we have typically 10,000 or more super-cells handled by a single GPU. This key space implies that the existing thrust algorithm needs to perform four 4-bit passes to sort the particles, while our algorithm can do it in a single pass, and we did in fact observe speed-ups of about 4× with the new algorithm.

Fig. 18 explains the improvement to the sorting algorithm for “almost sorted” particles. Fig. 18(a) shows the original thrust algorithm: We start out with 27 particles, which were originally sorted by the 4 super-cells. After the timestep, some particles have moved to a neighboring cell and are hence out of order (top row): E.g., particles 2 and 5, which were in super-cell 0, moved to super-cell 1, while particles 11 and 14 moved from super-cell 1 to super-cell 0.

The first step of the algorithm is a count of particles by super-cell. This count happens independently (and in parallel by GPU threadblocks) for each particles in each super-cell. E.g., super-cell 0 now has 6 zeros and 2 ones.

The second step is the top-level scan, which is performed by a single threadblock only. It performs an exclusive prefix sum of the counts from the previous steps along the arrows shown: It starts at 0, then 6 is added: “6”, then 2 is added: “6 + 2 = 8”, then 0 is added: “6 + 2 + 0 = 8”, etc. The results are, for each threadblock, the starting offsets by super-cell.

Control now returns to the individual threadblocks, which finish the sort in parallel. Starting from the original cell indices, the offset from the top-level prefix sum is added, as is the local particle number (“scan cells”): Particles 0, 1, 3, 4, 6, 7 are enumerated by 0, 1, 2, 3, 4, 5, and the local offset for 0’s, which is 0, is added. Particles 2 and 5, which move to super-cell 1, are enumerated as 0, 1, and the offset 8 is added, giving them target indices of 8, 9.

Now, as for each particle its target position is known, each particle gets moved into that position (“move”). The final step is to find the boundaries between super-cells in the now newly sorted particle array. These in fact have already been calculated as the first column of the “prefix sum” array.

The new particle sort algorithm is shown in Fig. 18(b). Its structure is similar to the general thrust implementation, but instead of working on target cell indices, it works on the relative direction that each particle moved. As this example is 1-d, the possible shifts are 0 or ±1 only. The main advantage of this algorithm comes from the fact that there are only 3 (or 9, 27 in 2-d, 3-d, respectively) possible movement directions, while there may be 10,000s of actual cell indices in a real simulation. The first step of the algorithm is to convert the cell indices to shifted indices relative to the current super-cell.

We then count, for each super-cell in parallel, the number of particles that remain or move in either direction.

The main difference in the new algorithm is the prefix sum, which now is not row-by-row, but rather follow a diagonal pattern, as indicated in the figure. The results are actually exactly the same offsets as in the original algorithm, as they should be.

The “scan cells” step again finds the final target position for each particle, so all that is left to do is to move particles to their new position, and adjust the bounds between super-cells.

Benchmarks of the new algorithm confirmed that it was in fact 4.2× faster in a case with 10,000 super-cells on the GPU than the generic thrust algorithm, as expected as we perform only one pass rather than four.
Figure 18. GPU Sort based on (a) cell indices and (b) relative directions.
5.4.2. Integrated particle boundary exchange and sort

Particle sort and boundary exchange across neighboring patches share commonalities: For the particle sort, the particle position determines the target super-cell. For boundary exchange, the particle position determines whether the particle has left the patch, and where it should go. Both require going through the list of all particles in a patch, and updating it – either by reordering, or by removing / inserting particles. We take advantage of this by integrating boundary exchange and sorting. In fact, the boundary exchange can be considered as part of a global sort. However, boundary exchange involves communication across processors while local sort does not, so the overall algorithm needs to distinguish those cases, and also needs to take into account that particles live in GPU memory, while MPI communication is performed by the CPU.

Simply stated, the integrated boundary exchange and sort involves three steps: prepare, communicate, and post-process.

The prepare and post-process steps are performed mostly on the GPU. The prepare step’s main task is to find particles that left the local patch and put them into a contiguous array, which can then be copied to CPU memory. The communicate step runs on the CPU as in the non-GPU case, it communicates particles to their respective target patch. Finally, the new arrivals for each patch are copied back to GPU memory and are sorted into place amongst the remaining particles in the post-process step.

Looking at those steps in more detail, the prepare step can also do useful preliminary work to help sorting the particles that remained in the patch, and more of the local sort work can be performed while particle exchange messages are in flight.

The details of the prepare step are as follows:

```plaintext
cuda_find_block_keys()
cuda_spine_reduce()
cuda_find_n_send()
cuda_scan_send_buf_total()
copy_from_dev()
convert_from_cuda()
```

The first step (cuda_find_block_keys()) is to find the target super-cell (“block”) index, which will be used as the sort key, for each particle in the patch.

The second step (cuda_spine_reduce()) iterates through each super-cell and keeps the following counts: how many particles move in each of the 8 possible directions, how many particles remain in the super-cell, and how many particles left the patch altogether (“out-of-bounds”). It will then perform an exclusive scan across the out-of-bounds particle numbers, which provide the starting offsets that tell each super-cell where out-of-bounds particles in that cells need to move.

cuda_find_n_send() computes the total count of out-of-bounds particles that leave the patch and reserves contiguous space for those particles after the existing particle array.

The fourth step (cuda_scan_send_buf_total()) finds the target index in the contiguous reserved space, and conversely the source index for each out-of-boundary particles that is to be moved into the contiguous area. It then moves all those particles into the new position.

The final steps copy the contiguous array of out-of-bounds particles onto the CPU and converts them into regular CPU data layout.

Those particles are then processed on the CPU by the usual boundary exchange routines, i.e., aggregated into messages per process rank and sent off. The result of the communication is a contiguous array of particles that newly entered a given patch. These are now put back onto the GPU and included with the existing particles in sorted order by psc_bnd_particles_post().

```plaintext
convert_to_cuda()
copy_to_dev()
cuda_find_block_indices_new()
cuda_sort()
cuda_update_offsets()
cuda_reorder()
```
The first two steps perform the conversion back into GPU layout and the copy into GPU memory. The new particles are added at the end of the existing particle array, overwriting the contiguous array of particles that left which was previously in that position.

At this point, conceptually all that remains to be done is to sort the newly extended array of existing (which includes particles that left) and new particles in a way that the particles that left are put at the end of the array, where they will be discarded by shrinking the array accordingly. A part of this work has already been performed: Particles to go into each super-cell have been counted. What is left is to add in the counts for the newly arrived super-cell. `cuda_find_block_indices_new()` finds the target super-cell for each of those particles and updates the counts. A scan by super-cell finds offsets on where to move each category of particle in that super-cell, and finally finds the target index for every particle (`cuda_sort()`).

Two tasks finish the job: `cuda_update_offsets()` finds the new boundaries between super-cells, i.e., for each super-cell it provides the index of the first and last particle in that super-cell, which is needed to process particles by super-cell in the subsequent particle push. Finally, `cuda_reorder()` actually moves each particle into its new, sorted position.

### 5.4.3. Performance optimization

Integrating particle exchange and sort, as just described, avoids unnecessary repeated loops over particles, however, it still needs to loop once in the beginning to find the target super-cell for each particle, and loops again in the end to actually reorder the particles into sorted order. These two operations can be further optimized by integrating them with existing kernels:

The particle push updates each particle’s position – at that point, the new position is known, and the super-cell index can be calculated and saved with little additional work. Similarly, the reorder step loops over positions in the new particle array, finds the particle index of the particle to be moved there (the result of the actual sort routine), reads the particle, and writes it into its new position. This can also be naturally integrated with the particle push, which already reads each particle, updates it, and writes it back. The difference is that the particles being read are not completely sorted then, as they haven’t been moved, but they are still mostly ordered. They are read in the right, sorted order, so there are no correctness issues. There is a small performance penalty for the unordered reads, but it is much less than the cost of a separate reorder loop.

Adding the computation of the super-cell index and the reorder step into the particle push / current deposition kernel slows it down by 0.120 ns (from 0.946 ns to 1.066 ns) per particle, but it accelerates the sort / particle exchange by 0.549 ns (from 0.913 ns to 0.364 ns).

We tested the GPU capabilities of rsc by reproducing the simulation of the Kelvin-Helmholtz instability and transition to turbulence from [46]. The parameters used were: Domain size $25d_i \times 50d_i$, mass ratio $m_i/m_e = 100$, resolution $8192 \times 4096$ cells, $\omega_{pe}/\omega_{ce} = 2$. The run was performed on BlueWaters using 256 Cray XK7 nodes and integrated 1,000,000 time steps in 20 hours of wallclock time, which includes I/O and checkpoint/restart. Fig. [19] shows two selected frames from the laminar and turbulent phases.

Table 4 breaks down where the computation time is spent. The simulation is initialized with $3.93 \times 10^7$ particles on each GPU, which take 58.82 ms to process, which translates into each GPU processing 669 million particles per second, which is a rather significant speed-up of 6.7x over the 16-core AMD CPU performance of 99 million particles per second on the same machine . 71% of the time are spent in particle advance and current deposition, and an additional 23% in particle boundary exchange and sort. So a total of 94% of the computation time are used for

| Task                                      | Time     |
|-------------------------------------------|----------|
| Particle advance & current deposit        | 41.69 ms |
| Particle sort / boundary exchange         | 13.70 ms |
| Field advance                             | 0.32 ms  |
| Field boundary exchange                   | 2.31 ms  |
| Total timestep                            | 58.82 ms |

Table 4. Breakdown of the computational cost per timestep running the Kelvin-Helmholtz test case on K20X GPUs.
particle-based tasks, while field-based tasks take up about 4%, with the remaining 2% spent in miscellaneous overhead like profiling, logging, etc.

This test case naturally maintains good load balance, after 500,000 steps, the time per step increased to 64.39 ms, a slow-down of less than 10%.

5.4.4. Multiple patches on a single GPU

Simulations on GPUs are still subject to potential load unbalance, so we employ our previously presented patch-based load balancing scheme. It is hence necessary to support multiple patches being processed by a single GPU. While our implementation supported this from the beginning, the performance degraded substantially when having many small patches rather than one big patch on a GPU. This is in contrast to running on the CPU, where subdivision into many patches made little difference. The issue is even more important when trying to use both CPU cores and GPUs at the same time, because load balancing in this case requires, e.g., assignment of one patch to each of 15 CPU cores, and then assignment of 90 patches to the GPU to even out the load based on computational capability, clearly leading to the GPU processing many small patches.

The straight-forward method of handling multiple patches on the GPU is to have a loop on the host CPU over patches, and for each patch call the GPU kernel to perform the work. Each of this GPU kernels splits the patch into super-cells and processes super-cells in parallel using CUDA threadblocks. However, if patches are too small, they consist of only a small number (tens or even less) of super-cells, which means that we do not launch enough threadblocks to keep all multi-processors on the GPU occupied, and performance suffers. In order to overcome this limitation, we rewrote the particles push / current deposition kernels to act on all patches at the same time – we now have the threadblocks handle super-cells of all local patches concurrently, rather than for one patch at a time, which means there is always sufficient work to keep the entire GPU busy. The new kernels successfully eliminated the penalty for using many small patches in the particle push kernel, though there remains some slow-down in boundary related tasks, as having smaller patches means that there are more internal boundaries to process.
6. Conclusions

psc is an explicit electromagnetic particle-in-cell code. It exploits the capabilities of today’s massively parallel super-computers and can be run on 100,000’s of conventional cores. Its flexible design allows for changing algorithms and data structures to support current and future processor designs. In previous sections, we have focused on two distinguishing features of psc: (1) Support for patch-based dynamic load-balancing using space-filling curves, which effectively resolves issues with performance degradation as particles move between local domains. (2) Support for Nvidia GPU’s, achieving a speed-up of more than 6× on Cray XK7 nodes.

psc encompasses both production and development capabilities in one code, allowing for state-of-the-art science simulations while at the same time, experimental new features are being developed, e.g., we are currently in the process of implementing support for Intel’s new many-integrated-cores processor technology (Intel Xeon Phi) which takes advantage of the new 512-bit SIMD capabilities.

Acknowledgements. This research was supported by DOE grant DE-FG02-07ER46372, NSF grants AGS-105689 and , and NASA grant NNX13AK31G . Computational work has been performed on DOE’s Titan machine at ORNL and NERSC systems, NSF’s Blue Waters at NCSA, and the Cray XE6m Trillian at the University of New Hampshire (funded with support from ).

Appendix A. Load balancing algorithm

The load balancing algorithm takes as input an array of the loads for each patch \( L \), and a list of capabilities for each process rank \( C_r \). The output is an array of the patch count \( N_r \) that each process rank is assigned.

\[
\begin{align*}
L_{\text{total}} &= \text{sum}(L) \quad \text{// } L_{\text{total}} &= \sum_p L_p \\
C_{\text{total}} &= \text{sum}(C) \quad \text{// } C_{\text{total}} &= \sum_r C_r \\
T_{\text{hat}} &= \frac{L_{\text{total}}}{C_{\text{total}}} \quad \text{// } T_{\text{hat}} &= \frac{L_{\text{total}}}{C_{\text{total}}} \\
r &= 0 \quad \text{// } r &= 0 \\
N_{\text{cur}} &= 0 \quad \text{// } N_{\text{cur}} &= 0 \\
L_{\text{cur}} &= 0 \quad \text{// } L_{\text{cur}} &= 0 \\
\text{for } p \text{ in range}(N_{\text{patches}}): \quad \text{// } \text{for } p \text{ in range}(N_{\text{patches}}): \\
L_{\text{cur}} &= L[p] \quad \text{// } L_{\text{cur}} &= L[p] \\
N_{\text{cur}} &= 1 \quad \text{// } N_{\text{cur}} &= 1 \\
\text{if } r < N_{\text{procs}} - 1: \quad \text{// } \text{if } r < N_{\text{procs}} - 1: \\
T_{\text{cur}} &= T_{\text{hat}} \cdot C_r \quad \text{// } T_{\text{cur}} &= T_{\text{hat}} \cdot C_r \\
\text{if } (L_{\text{cur}} > T_{\text{cur}} \text{ or } N_{\text{procs}} - r \geq N_{\text{patches}} - p): \quad \text{// } \text{if } (L_{\text{cur}} > T_{\text{cur}} \text{ or } N_{\text{procs}} - r \geq N_{\text{patches}} - p): \\
\text{above_target} &= L_{\text{cur}} - T_{\text{cur}} \quad \text{// } \text{above_target} &= L_{\text{cur}} - T_{\text{cur}} \\
\text{below_target} &= T_{\text{cur}} - (L_{\text{cur}} - L[p]) \quad \text{// } \text{below_target} &= T_{\text{cur}} - (L_{\text{cur}} - L[p]) \\
\text{if } (\text{above_target} > \text{below_target} \text{ and } nr_{\text{new_patches}} > 1): \quad \text{// } \text{if } (\text{above_target} > \text{below_target} \text{ and } nr_{\text{new_patches}} > 1): \\
N[r] &= N_{\text{cur}} - 1 \quad \text{// } N[r] &= N_{\text{cur}} - 1 \\
N_{\text{cur}} &= 1 \quad \text{// } N_{\text{cur}} &= 1 \\
\text{else:} \quad \text{// } \text{else:} \\
N[r] &= N_{\text{cur}} \quad \text{// } N[r] &= N_{\text{cur}} \\
N_{\text{cur}} &= 0 \quad \text{// } N_{\text{cur}} &= 0 \\
r &= r + 1 \quad \text{// } r &= r + 1 \\
\text{if } p == N_{\text{patches}} - 1: \quad \text{// } \text{if } p == N_{\text{patches}} - 1: \\
N[N_{\text{proc}} - 1] &= N_{\text{cur}} \quad \text{// } N[N_{\text{proc}} - 1] &= N_{\text{cur}}
\end{align*}
\]

Appendix B. Ghostpoints needed to support only one synchronization point
References

[1] J. Egedal, A. Le, W. Daughton, A review of pressure anisotropy caused by electron trapping in collisionless plasma, and its implications for magnetic reconnection, Physics of Plasmas (1994-present) 20 (2013) –.

[2] J. Egedal, W. Daughton, A. Le, Large-scale electron acceleration by parallel electric fields during magnetic reconnection, Nature Physics (2012).

[3] L. Chacón, G. Chen, D. Barnes, A charge- and energy-conserving implicit, electrostatic particle-in-cell algorithm on mapped computational meshes, Journal of Computational Physics 233 (2013) 1 – 9.

[4] G. Lapenta, Particle simulations of space weather, Journal of Computational Physics 231 (2012) 795 – 821.

[5] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, T. J. T. Kwan, Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation, Physics of Plasmas 15 (2008) 055703.

[6] R. A. Fonseca, L. O. Silva, F. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsoulea, et al., Orsis: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators, in: Computational Science—ICCS 2002, Springer, 2002, pp. 342–351.

[7] C. Nieder, J. R. Cary, Vorpel: a versatile plasma simulation code, Journal of Computational Physics 196 (2004) 448 – 473.

[8] H. Ruhl, Classical particle simulations, in: M. Bonitz, D. Semkat (Eds.), Introduction to Computational Methods in Many Body Physics, Rinton Press, 2006.

[9] C. Birdsal, A. Langdon, Plasma Physics Via Computer Simulation, Series in plasma physics, Taylor & Francis, 1991.

[10] R. Hockney, J. Eastwood, Computer Simulation Using Particles, McGraw-Hill, 1981.

[11] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 135 (1997) 280 – 292.

[12] K. Yee, Numerical solution of initial boundary value problems involving maxwell’s equations in isotropic media, Antennas and Propagation, IEEE Transactions on 14 (1966) 302–307.

[13] J. Boris, Acceleration calculation from a scalar potential, 1970.

[14] J. Villasenor, O. Buneman, Rigorous charge conservation for local electromagnetic field solvers, Computer Physics Communications 69 (1992) 306–316.

[15] T. Esirkepov, Exact charge conservation scheme for particle-in-cell simulation with an arbitrary form-factor, Computer Physics Communications 135 (2001) 144 – 153.

[16] I. V. Sokолов, Alternating-order interpolation in a charge-conserving scheme for particle-in-cell simulations, Computer Physics Communications 184 (2013) 320 – 328.

[17] E. Evstatiev, B. Shadwick, Variational formulation of particle algorithms for kinetic plasma simulations, Journal of Computational Physics 245 (2013) 376 – 389.

[18] A. B. Langdon, On enforcing Gauss’ law in electromagnetic particle-in-cell codes, Computer Physics Communications 70 (1992) 447–450.

[19] A. Bhattacharjee, K. Germaschewski, C. Ng, Current singularities: Drivers of impulsive reconnection, Phys. Plasmas 12 (2005) 042305.

[20] K. Germaschewski, A. Bhattacharjee, C. Ng, The magnetic reconnection code: an amr-based fully implicit simulation suite, in: N. Pogorelov, G. Zank (Eds.), Numerical Modeling of Space Plasma Flows: Astronum-2006, volume 359, ASP Conference Series, 2007, p. 15.

[21] K. Germaschewski, J. Raeder, D. J. Larson, A. Bhattacharjee, New developments in modeling MHD systems on high performance computing architectures, in: N. Pogorelov, G. Zank (Eds.), Numerical Modeling of Space Plasma Flows: Astronum-2008, volume 406, ASP Conference Series, 2009, pp. 223–230.

[22] J. Raeder, Global Magnetohydrodynamics – A Tutorial, in: J. Büchner, C. T. Dum, M. Scholes (Eds.), Space Plasma Simulation, Springer Verlag, Berlin Heidelberg New York, 2005.

[23] K. Germaschewski, J. Raeder, Using automated code generation to support high performance extended mhd integration in openpgcm, in: N. Pogorelov, G. Zank (Eds.), Numerical Modeling of Space Plasma Flows: Astronum-2010, volume 444, ASP Conference Series, 2011, p. 197.

[24] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Web page, 2001. http://www.mcs.anl.gov/petsc

[25] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Users Manual, Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[26] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.

[27] T. H. Group, Hierarchical data format version 5, 2000-2010.al data format version 5,http://www.hdfgroup.org/HDF5

[28] W. Fox, A. Bhattacharjee, K. Germaschewski, Magnetic reconnection in high-energy-density laser-produced plasmas, Physics of Plasmas 19 (2012) 056309.

[29] H. Cheng, L. Greengard, V. Rokhlin, A fast adaptive multipole algorithm in three dimensions, Journal of Computational Physics 155 (1999) 468 – 498.

[30] R. D. Ferraro, P. C. Liewer, V. K. Decyk, Dynamic load balancing for a 2d concurrent plasma (PIC) code, Journal of Computational Physics 109 (1993) 329 – 341.

[31] A. Pukhov, Three-dimensional electromagnetic relativistic particle-in-cell code vlpl (virtual laser plasma lab), Journal of Plasma Physics 61 (1999) 425–433.

[32] E. Saule, E. O. Bağ, Ü. V. Catalyurek, Load-balancing spatially located computations using rectangular partitions, Journal of Parallel and Distributed Computing 72 (2012) 1201 – 1214.

[33] D. Hilbert. Ueber die stetige abildung einer line auf ein flächenstück, Mathematische Annalen 38 (1891) 459–460.

[34] G. Peano, Sur une courbe, qui remplit toute une aire plane, Mathematische Annalen 36 (1890) 157–160.

[35] K. Germaschewski, A. Bhattacharjee, R. Grauer, D. Keyes, B. Smith, Using krylov-schwarz methods in an adaptive mesh refinement environment, in: T. Plewa, T. Linde, V. G. Weirs (Eds.), Adaptive Mesh Refinement - Theory and Applications, Springer, 2005, pp. 115–124.
[36] B. van der Holst, R. Keppens, Z. Meliani, A multidimensional grid-adaptive relativistic magnetofluid code, Computer Physics Communications 179 (2008) 617 – 627.

[37] A. Calder, B. Fryxell, T. Plewa, R. Rosner, L. Dursi, V. Weirs, T. Dupont, H. Robey, J. Kane, B. Remington, R. Drake, G. Dimonte, M. Zingale, F. Timmes, K. Olson, P. Ricker, P. Macneice, H. Tuflo, On validating an astrophysical simulation code, Astrophysical Journal, Supplement Series 143 (2002) 201–229. Cited By (since 1996)112.

[38] W. Fox, A. Bhattacharjee, K. Germaschewski, Fast magnetic reconnection in laser-produced plasma bubbles, Phys. Rev. Lett. 106 (2011) 215003.

[39] TOP500, Top 500 supercomputing sites, 2013. http://www.top500.org/.

[40] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma (PIC) simulations on the (GPU), Journal of Parallel and Distributed Computing 68 (2008) 1339 – 1349. General-Purpose Processing using Graphics Processing Units/General-Purpose Processing using Graphics Processing Units.

[41] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. Cowan, R. Sauerbrey, M. Busmann, Picongpu: A fully relativistic particle-in-cell code for a gpu cluster, Plasma Science, IEEE Transactions on 38 (2010) 2831–2839.

[42] V. K. Decyk, T. V. Singh, Adaptable particle-in-cell algorithms for graphical processing units, Computer Physics Communications 182 (2011) 641 – 648.

[43] X. Kong, M. C. Huang, C. Ren, V. K. Decyk, Particle-in-cell simulations with charge-conserving current deposition on graphic processing units, Journal of Computational Physics 230 (2011) 1676 – 1685.

[44] D. Merrill, A. Grimshaw, High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing, Parallel Processing Letters 21 (2011) 245–272.

[45] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore gpus, in: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 2009. pp. 1–10. doi:10.1109/IPDPS.2009.5181005.

[46] H. Karimabadi, V. Roytershteyn, M. Wan, W. H. Matthaeus, W. Daughton, P. Wu, M. Shay, B. Loring, J. Borovsky, E. Leonardis, S. C. Chapman, T. K. M. Nakamura, Coherent structures, intermittent turbulence, and dissipation in high-temperature plasmas, Physics of Plasmas 20 (2013) 012303.