Computer Physics Communications 259 (2021) 107633

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Dynamic load balancing with enhanced shared-memory parallelism for particle-in-cell codes☆

Kyle G. Miller [a], Roman P. Lee [a,*], Adam Tableman [a], Anton Helm [b], Ricardo A. Fonseca [c,b], Viktor K. Decyk [a], Warren B. Mori [a]

[a] Department of Physics and Astronomy, University of California, Los Angeles, CA 90095, USA
[b] GoLP/Instituto de Plasmas e Fusão Nuclear, Instituto Superior Técnico, 1049-001 Lisboa, Portugal
[c] ISCTE - Instituto Universitário de Lisboa, Av. Forças Armadas, 1649-026 Lisboa, Portugal

ARTICLE INFO

Article history:
Received 18 March 2020
Received in revised form 16 July 2020
Accepted 21 September 2020
Available online 25 September 2020

Keywords:
Particle-in-cell (PIC)
Dynamic load balancing
OpenMP parallelization
Plasma kinetic simulation
High-performance computing

ABSTRACT

Furthering our understanding of many of today's interesting problems in plasma physics – including plasma based acceleration and magnetic reconnection with pair production due to quantum electrodynamic effects – requires large-scale kinetic simulations using particle-in-cell (PIC) codes. However, these simulations are extremely demanding, requiring that contemporary PIC codes be designed to efficiently use a new fleet of exascale computing architectures. To this end, the key issue of parallel load balance across computational nodes must be addressed. We discuss the implementation of dynamic load balancing by dividing the simulation space into many small, self-contained regions or "tiles," along with shared-memory (e.g., OpenMP) parallelism both over many tiles and within single tiles. The load balancing algorithm can be used with three different topologies, including two space-filling curves. We tested this implementation in the code Osiris and show low overhead and improved scalability with OpenMP thread number on simulations with both uniform load and severe load imbalance. Compared to other load-balancing techniques, our algorithm gives order-of-magnitude improvement in parallel scalability for simulations with severe load imbalance issues.

1. Introduction

The particle-in-cell (PIC) algorithm is widely used to study interesting problems where discrete particles or agents interact through fields. The PIC algorithm has thus been widely used in the kinetic modeling of plasmas, where the fields can either be electrostatic or electromagnetic. However, the nature of tracking individual particles over long periods of time makes the PIC algorithm computationally expensive, requiring the use of large-scale high-performance computing (HPC) resources. With the advent of exascale computing, HPC architectures are undergoing rapid change; since 2004, clock rates have stabilized and growth on top-ranked systems has come almost entirely from increased parallelism. The PIC algorithm is sensitive to load imbalance on excessively parallel machines because simulation particles move about and may accumulate on a fraction of the computing resources. As the scale of massively parallel computing architectures continues to intensify, the final push toward exascale and beyond will require significant adaptation of software to take advantage of the increased parallelism available in the hardware.

Distributing computational load evenly across resources can be achieved through multiple levels of parallelism. At the highest level one can parallelize across distributed-memory processing elements (PEs). Parallelism on this level is often implemented via the Message Passing Interface (MPI), and for clarity, in this paper PE always refers to an MPI process. In addition, the increasing number of shared-memory CPUs inside compute nodes, as well as the increasing number of cores inside today's CPUs, allows for parallelism within a processing element (e.g., via OpenMP or Pthreads). Finally, the use of many-core accelerators from multiple vendors such as Graphical Processing Units (GPUs) also allows for parallelism via CUDA, OpenACC, HIP, or SYCL/DPC++.

In this paper we present developments to improve the parallel scalability of the particle-in-cell (PIC) code Osiris [1,2], which can be applied to any massively parallel PIC code. On top of the parallel computing challenges presented by evolving hardware architectures, parallelizing a PIC code while maintaining load balance is inherently challenging on an algorithmic level. A PIC code contains two main data structures that comprise the computational load in a simulation: particles that can occupy any position in the simulation domain and field quantities that

☆ The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author.
E-mail addresses: kylemiller@physics.ucla.edu (K.G. Miller), romanlee@physics.ucla.edu (R.P. Lee), ricardo.fonseca@iscte-iul.pt (R.A. Fonseca), mori@physics.ucla.edu (W.B. Mori).

https://doi.org/10.1016/j.cpc.2020.107633
0010-4655/

are discretized on a mesh grid. Parallelization is done by distributing particles and grid points among PEs, with load balance being achieved when the computational load, here defined as the calculation time, associated with these structures is distributed uniformly. Osiris [1,2] and most cutting-edge PIC codes use a spatial grid-based domain decomposition [3–5], where each PE is responsible for a subset of the global spatial grid as well as any particles located there. While this domain decomposition algorithm is widely used, it suffers from the possibility that a large number of particles may move into a single PE (as is often the case for simulations of plasma-based acceleration, laser-solid interactions, or magnetic reconnection with pair production). Thus maintaining acceptable load balance can be challenging, given that the computational load will generally scale with the number of particles.

Various strategies have been implemented in an effort to dynamically load balance PIC codes using grid-based decomposition. Perhaps the most straightforward load balancing scheme is to enlarge distributed-memory spatial domains by using a large number of shared-memory cores on each PE and distribute the computational load evenly across these cores. This allows for localized load-imbalance situations, such as large density spikes, to be smoothed out, generally leading to good improvements in performance and scalability [2]. This offers limited relief, however, due to hardware limitations on the available number of cores. Another solution based on a static equidistant domain decomposition is known as the taskfarm alternative [6]. Here, each regular and equally sized distributed-memory sub-domain is further subdivided uniformly into tasks, with particles sorted accordingly. To process particles, each PE works serially through its own set of tasks before accessing, completing and returning the tasks of other PEs with higher load. If tasks are small enough, load is balanced since no PE is ever idly waiting for remaining tasks to be completed. A different approach is taken by Liewer and Decyk who pioneered the idea of shifting PE boundaries in their 1988 algorithm GCPIC [7,8]. Computational load is projected onto one axis so that the problem of load balancing becomes one dimensional. Partitions are found along this axis, and the resulting domains are partitioned in the second and third dimensions in the same way. Similar approaches based on rectilinear partitioning are taken in [2,9,10]. Dynamic load balancing can also be achieved by decomposing the simulation into many small units called tiles (or patches). Each PE handles one or more of these units, with the algorithm dynamically assigning them between PEs to maintain an even load [3,11]. Similar strategies have also been employed on GPU architectures [12].

In this paper we extend the previous dynamic load balancing algorithm of Osiris [2] by dividing the global simulation space into many small, self-contained "tiles", which contain all particle and grid quantities for a particular region of space. The parallel domain decomposition is determined by assigning one or more tiles to each PE such that computational load is as balanced as possible; in addition, one or more threads are assigned to each tile within a given PE based on computational load. The ability to assign multiple threads to each tile – following the shared-memory parallelization algorithm already present in Osiris – allows us to significantly improve performance for simulations with small regions of high particle density by enabling the parallel use of a multi-core PE on a single tile. This provides significant improvement over previous tile-based dynamic load balancing implementations [3,11] that allow for only one thread per tile on CPUs. We find that this feature allows for the use of larger tiles – reducing the overhead of passing particles between tiles – while still maintaining load balance. Our implementation also scales well with thread number and gives particularly large speedups for very imbalanced simulations, being well suited for efficient use in today's evolving HPC climate as available on-chip thread count continues to climb.

This paper is organized as follows: In Section 2, we discuss the implementation of the tile structure into Osiris, including both distributed-memory and shared-memory parallelization schemes. We discuss the overhead and performance of the tiling scheme in Section 3 by analyzing simulations both with and without load imbalance, including a 3-D simulation of particle wakefield acceleration. Compared to previous Osiris algorithms, our tile-based implementation of dynamic load balancing gives an order-of-magnitude increase in scalability with thread number and more than a factor of 2 overall speedup for two different physics simulations.

2. Methodology

In the last two decades, the continued growth of top-ranked HPC systems has come almost entirely from increased parallelism, with present systems comprising up to $\sim 10^6$ cores. At the highest level of parallelism, these massive computer systems are viewed as a network of distributed-memory PEs amongst which the simulation can be partitioned. Given that these PEs do not share memory, using a spatial domain decomposition requires that particle and field data be exchanged between neighboring PEs at each time step. Domains for each PE should be structured such that the computation time on a given region is much larger than the time spent communicating boundary information, i.e. maximizing the computational volume to boundary surface area ratio to minimize parallelization overhead.

Previously, the domain decomposition in Osiris was structured such that each PE had only one neighboring PE in each direction (i.e., domain corners always matched up). These domains could be statically assigned at the beginning of the simulation or changed dynamically throughout to maintain load balance [2]. Aligning domain corners simplified the communication pattern for the sharing of boundary information, but limited the achievable load balance. In an effort to improve dynamic load balancing and to enhance shared-memory parallelism in Osiris, we decompose the global simulation space into many small, static, regularly spaced rectangular regions called "tiles" which have aligned corners, following a strategy similar to [3,11]. The domain decomposition is then determined by assigning a collection of one or more tiles to each PE such that computational load is balanced. These tiles can be exchanged dynamically between all PEs at chosen intervals throughout the simulation to maintain load balance.

2.1. Distributed-memory parallelism

To determine the most balanced parallel partition, i.e., assignment of tiles to PEs, we must first devise a way to quantify and organize the computational work required to process each tile. To this end, we create a load array with the same dimensions as the simulation and containing one entry per tile. Assuming that the computational load scales linearly with both the number of particles and the number of grid points,[1] we compute the array entry for the ith tile, $\mathcal{L}_i$, as $\mathcal{L}_i = \mathcal{N}_{i,\text{part}} + \mathcal{C}\mathcal{N}_{i,\text{cells}}$ as in [11], where $\mathcal{N}_{i,\text{part}}$ and $\mathcal{N}_{i,\text{cells}}$ are the number of particles and cells in the ith tile, respectively, and the cell weight $\mathcal{C}$ represents the computational load of a single cell compared to one particle. This depends on simulation parameters such as interpolation level or field solver type, and is left for the user to define. We found that for typical 2- and 3-D simulations $\mathcal{C} \approx 0.5$–2 is effective. The

[1] This is accurate for finite-difference-based field solvers. For other types of field solvers a different load formula can be straightforwardly derived.

load array can then be partitioned into $\mathcal{N}_{\mathrm{PE}}$ regions of contiguous tiles such that the total load in each partition is as close as possible to the ideal average load, $\sum_i \mathcal{L}_i/\mathcal{N}_{\mathrm{PE}}$, where $\mathcal{N}_{\mathrm{PE}}$ is the total number of PEs. Partitioning a 1-D load array in this way is trivial since boundaries must be determined only along one dimension. However, partitioning a 2- or 3-D load array is more complex, and multiple solutions may be considered.

One approach proposed by Saule et al. [9] is to consider 2-D partitions where each PE is left with a rectangular region of space, with each boundary connecting to one or more neighbors. Partitions where each boundary connects to a single neighbor (matching corners) – referred to as a rectilinear partition in [9] – was previously implemented in Osiris [2]. A more complex scheme, referred to as $P \times Q$ jagged in [9], allows for multiple neighbors along boundaries in the x dimension, and only one neighbor in y. This allows for decomposing the multi-dimensional load balance problem into separate uni-dimensional scenarios: we first define P partitions along x so that the load is evenly divided among them; we then proceed by dividing each of these P partitions into Q sub-partitions such that the computational load for all Q sub-partitions in a P partition is equal. This process can be straightforwardly extended to 3 dimensions. The $P \times Q$ jagged partition is intuitive and has rather simple boundaries between PEs, but for some computational load distributions it may not yield a perfectly balanced configuration, given that the boundary positions are limited to grid cell boundaries. See [8] for an example of a PIC code using the jagged partition.

Another method to load balance in multiple dimensions is to place the tiles sequentially along a space-filling curve [11]. After estimating the optimal load per PE, we start with the first PE and, following the curve, assign tiles to it until the load is close to the optimal value. We then proceed to the next PE/tile until all tiles have been assigned. The advantage of using a space-filling curve for ordering the tiles as opposed to, say, choosing the tiles based solely on their computational load, is that we will end up with simulation domains that are contiguous in space and have relatively simple boundaries with a small number of neighbors. We implement this method by creating a 1-D load array, where tiles are ordered in the array using their position along the space-filling curve, then dividing this array into $\mathcal{N}_{\mathrm{PE}}$ segments with roughly equal load. Tiles falling on each segment will be assigned to the corresponding PE.

We implemented both the $P \times Q$ jagged partition and the space-filling curve methods in Osiris, with two choices for the space-filling curve. The first choice (Snake) simply passes through each tile by snaking back and forth in simulation space. This curve has no restrictions on domain size and yields the simplest boundaries, but does not maximize the ratio between computational volume and boundary surface area. The second choice (Hilbert) traces through the tiles using a Hilbert curve [13], will maximize the ratio between computational volume and boundary surface area, but requires that the tile number be a power of 2 in the smallest dimension (and integer multiples of that number in other dimensions) and leads to more complex boundary shapes, potentially with more neighbors. Fig. 1 shows schematics of various domain decompositions using 4 PEs for a simulation featuring a high-density diagonal stripe of particles (blue) surrounded by vacuum. We show a uniform partition without using tiles, as well as partitions using the three load balancing schemes implemented here. Note that with uniform partitioning, two PEs contain very few particles. For the density profile and tile size shown here, all three tile schemes achieve roughly the same degree of load balance across PEs, though for more complicated profiles the $P \times Q$ jagged scheme will usually not load balance as well as the Snake/Hilbert schemes. To minimize communication overhead, we group messages from multiple tiles with the same destination PE into a single message, which greatly improves MPI performance.

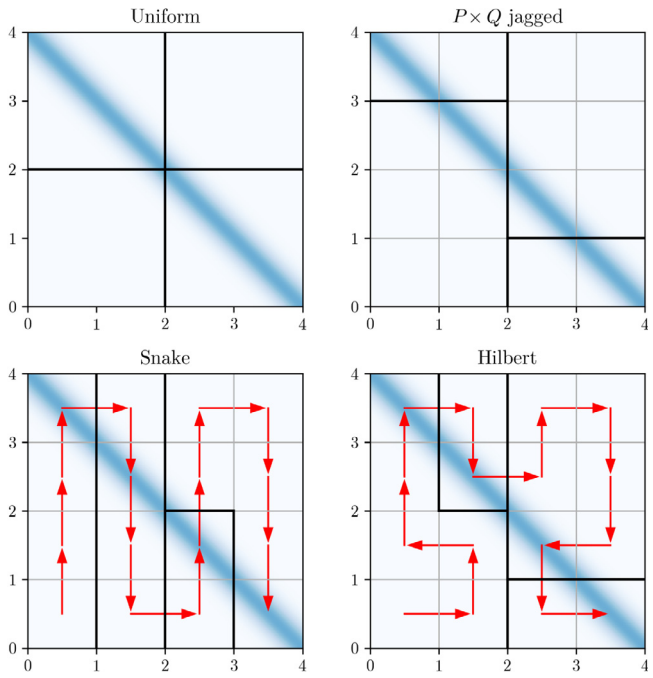

Fig. 1. Domain decomposition using 4 PEs for a high-density diagonal stripe of particles (blue). We show a uniform partition without tiles, as well as partitions using the three load balancing schemes implemented here. Gray lines indicate tile boundaries, black lines indicate PE (i.e., MPI) boundaries, and red arrows trace out tile ordering along the space-filling curve. Note that with uniform partitioning, two PEs contain very few particles. For the density profile and tile size shown here, all three tiles schemes achieve roughly the same degree of load balance.

To summarize, load balancing is performed by calculating the load array, determining the load-balanced domain decomposition, and then distributing the tiles among PEs accordingly. This can be done solely at the beginning of the simulation or dynamically throughout the simulation at chosen intervals using current simulation information. The domain decomposition defines a mapping between tiles and PEs, which is used to determine which tiles are to be sent and received, where tiles with the same destination PE are grouped into a single MPI buffer to minimize the effect of latency. Note that each successive domain decomposition is independent of the previous decomposition, i.e., a single PE may send all of its tiles to various other PEs and receive entirely new ones.

2.2. Shared-memory parallelism

The parallelization and load balancing algorithm described in the previous section works well for several problems, but does not guarantee ideal load balance for challenging scenarios where a large number of particles accumulate in a small number of tiles. Consider such a situation, where the load on a particular tile corresponds to α times the optimal load per PE, where $\alpha > 1$. In this case, the best the load balancing algorithm can do is to assign that tile to a single PE. However, the load for that PE will be α times the optimal value, and we will experience a slowdown by a factor of α. To address these situations and further improve parallel performance, we exploit the possibility for shared-memory parallelism that is available on most of today's computer systems, where a single PE may have many cores or threads that share memory with one another. While

load balancing across distributed memory must be done with the coarse resolution of one tile, load balancing within a single PE with many threads can be done to greater resolution, for example, by dividing particles evenly among shared-memory threads.

For the case where multiple tiles are assigned to a single PE, there are two common approaches for processing the tiles using shared-memory parallelism. If there are many more tiles than available threads on the PE, load balance can be achieved by looping over tiles using a dynamic scheduler ("first come, first served"), with one thread per tile. This has been previously demonstrated by [3]. However, if one single tile contains a very large number of particles, the associated thread will take much longer than the others and load balance will fail.

Alternatively, tiles can be processed in serial with all threads working on a single tile at any given time. This way a single thread is never stuck on a tile with many particles. Dividing up particles in a single region of space amongst shared-memory resources is a strategy commonly implemented in GPU codes [12,14]. However, given that all threads are assigned to the same region of space (tile), we must avoid memory collisions when doing the current deposition. This can be achieved either by using atomic operations, or by creating a separate electric current grid for each thread, then summing these arrays together after the current deposition is complete. Both options have specific drawbacks that impede parallel scalability for large numbers of threads, but the latter option generally gives better performance on CPUs and is used in Osiris [2].

We take a novel approach and include both types of shared-memory parallelism in a single framework. For a given PE with $\mathcal{T}$ threads and total load $\mathcal{L}$, let $\mathcal{L}_j$ be the load found on its jth tile. Each time step we calculate which tiles should be processed in parallel (one thread per tile) and which tiles should be processed in serial ($\mathcal{T}$ threads per tile). A tile is deemed "heavy" if its load is greater than the average load per thread, i.e., if $\mathcal{L}_j \geq \mathcal{L}/\mathcal{T}$, and "light" otherwise. However, if a PE has fewer tiles than threads, all tiles are classified as "heavy" to avoid idle threads. For routines like those in particle processing, light tiles are processed first by assigning one thread per tile using a dynamic scheduler (we use OpenMP). Once all light tiles are completed, each heavy tile is executed one at a time, with all threads processing particles on that tile in parallel.

A schematic of the heavy-light tile organization is shown in Fig. 2 for a PE with five light tiles, two heavy tiles, and four threads. Using light tiles avoids data dependency issues when there are many more tiles than threads. The inclusion of heavy tiles ensures that no threads are idle in cases with (1) fewer tiles than threads on a PE or (2) disproportionately large load on a single tile. This implementation is critical for simulations with very high particle densities, such as plasma wakefield or shock simulations, where the distributed-memory load balance may result in PEs having just one tile containing a majority of particles. Note that tile G is queued for execution by the first available thread. If all light tiles have similar load, this leftover tile could double the execution time of light tiles compared to having only four light tiles. The number of light tiles could be adjusted to be divisible by the number of threads, or light tiles could be sorted by decreasing load to mitigate this overhead. However, we do not address this particular issue here since the impact is minimal.

Alternatively, it is possible to use a task-based approach to thread over tiles. The algorithm begins by creating one task per tile, for all tiles. Light tiles will then be processed without further parallelism, while heavy tiles will be processed using multiple threads per tile, with the number of threads being proportional to the number of particles in the tile. The number of threads for each heavy tile can be calculated as $\mathcal{T}_i = \lfloor \mathcal{T} \times (\mathcal{N}_i/\mathcal{N}) \rceil$ (round to nearest integer), where $\mathcal{T}$ is the total number of available threads

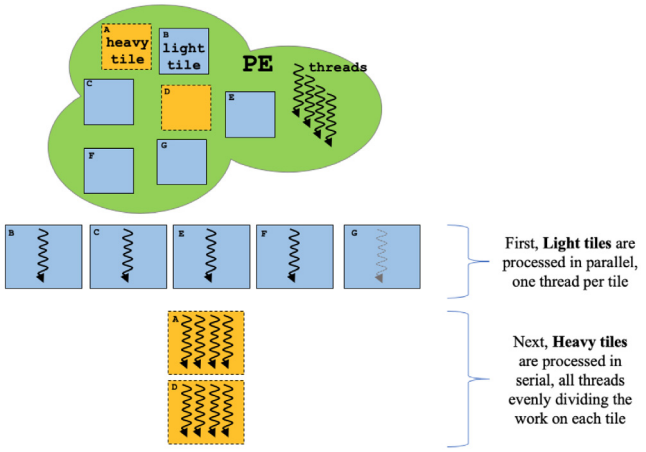

Fig. 2. Work flow for shared-memory parallelization on a single MPI process with four threads. Each MPI process handles its tiles differently depending on their computational weight – a feature unique to our implementation. Using shared-memory threads (OpenMP in our case), tiles with load less than the average load per thread on that MPI process (light tiles) are processed in parallel with one thread per tile. The gray dashed thread in tile G represents a light tile queued for execution by the first available thread. Lastly, tiles with above-average load (heavy tiles) are processed in serial with all threads dividing work on that tile evenly.

in the PE, $\mathcal{N}$ is the total number of particles in the PE, and $\mathcal{N}_i$ is the number of particles in the tile. These tasks will be scheduled dynamically, assigning, if possible, a higher priority to light tiles. This approach should, in most scenarios, reduce the number of threads assigned to individual heavy tiles and ensure better load distribution across cores inside a PE, and will be further explored in a future publication.

2.3. Tile boundary management

When using a parallelization scheme based on spatial domain decomposition (such as the tiling scheme described here), each tile is required to exchange information with the neighboring tiles at each time step. This includes both grid information (edge values of fields and densities) and particle information (particles moving to/coming from another tile). Since each tile is self-contained, boundary information must be exchanged between all neighboring tiles – both tiles found within a single PE and those located across a PE–PE boundary. The exchange of boundary information introduces overhead not only from the MPI communications, but also from reassigning particles to different tiles within the same PE, with the latter overhead increasing as tile size decreases. If not optimized, these overheads can outweigh the benefit of the load balance algorithm, severely limiting its applicability.

In our implementation, boundary information between different PEs is packed into buffers and shared via MPI; tile boundary information within a single PE is instead referenced directly in shared memory. Whenever tiles are (re)assigned to PEs, each PE will gather and store information regarding which of its tile boundaries are internal and which are external. Each time step when boundary information is exchanged, boundaries are processed sequentially over the number of dimensions to avoid corner communications. When exchanging grid information, we loop over all tiles using parallel threads; local tile boundary values are copied directly in shared memory (no buffer required), and boundary values to be sent to other PEs are packed into one buffer per target PE and sent via MPI. We then loop over all tiles with external boundaries (again using parallel threads) to copy

in boundary values from the received communication buffers. Particles moving between tiles are processed in a similar manner. Each tile maintains buffers to store exiting particles; buffered particles moving to a local tile are referenced and unpacked, while particles moving to a different PE are packed and sent over MPI. Grouping MPI messages going from a local PE to the same target PE greatly improves performance (as opposed to using one message per tile boundary, for example) by reducing the number of messages and limiting the impact of communication latency, since message sizes are larger.

Using the tiling scheme with either of the space-filling curves will lead to a small increase in communication time when compared to a uniform partition. This is mainly due to an increased number of neighboring PEs (the corners of each PE are no longer aligned) that communicate smaller messages (i.e., communication is mostly latency-dominated). This increase in communication time can be offset by obviating the particle sorting step required in most PIC codes while still ensuring data locality and cache coherency (due to the small size of a tile). Furthermore, the benefit of proper parallel load balance can greatly outweigh any penalties incurred from communication overhead for otherwise unbalanced runs.

2.4. Best practices

In this section we briefly discuss parameters a user or developer should consider when working with a tile-based PIC code. First, the choice of tile size has a significant impact on the performance of the algorithm: smaller tile sizes improve data locality and load balance at the expense of a higher overhead from tile boundary management, as the number of particles (compared to the total particles in a tile) crossing boundaries will increase, as will the number of boundaries between PEs. On the other hand, larger tile sizes reduce the boundary management overhead but can interfere with data locality and hinder parallel load balancing by limiting the available resolution of the parallel partition. The optimal tile size will be problem-specific, as discussed in further detail in Section 3.1.

Second, the choice of how frequently to perform the dynamic load balance is problem-specific, but in our experience it should be done rather frequently, e.g., once every 20 or 40 time steps. For all simulations in Section 3, the amount of total simulation time spent performing the dynamic load balance was always below 4%, (usually between 1%–2%). Good load balance was maintained for a very rapidly evolving plasma with little overhead by repartitioning every 20 time steps.

Finally, perhaps the biggest concern for developers and users of a tile-based PIC code is memory management. Specifically, extra buffers need to be maintained to store particles migrating between self-consistent tiles and PEs. As mentioned briefly in Section 2.3, our Osiris implementation maintains two types of buffers at the tile level and one extra type of particle buffer at the PE level. Each tile must have (1) a buffer to hold all of its particles and (2) a buffer (or multiple) to store the particles that have left its domain and need to be moved elsewhere. In addition, each PE requires separate buffers to store particles to be communicated to various neighbor PEs. The sizes of buffers used to store migrating particles must be chosen carefully to maximize performance (e.g., limit reallocation and number of MPI messages) without running out of memory. When buffers are grown to include additional particles, increasing the buffer size by only a small amount may trigger memory reallocation every few time steps during a large influx of particles. However, if buffers are grown by a large amount, a small influx of particles across a large portion of the simulation space could be very expensive. In our case, we give the user the ability to define the initial size of the main particle buffers in each tile, along with the amount by which the buffer is grown when necessary. These parameters are problem-specific, and may require some experimentation to appropriately define. Internal memory management could also be used to write checkpoint data when the allocated memory approaches the hardware limit.

3. Results

To evaluate the performance of our dynamic load balance algorithm using tiles we will benchmark it against the baseline performance of Osiris using a static, regular spatial domain decomposition (referred to as "no tiles" or "Osiris" in all figures). As mentioned earlier, these results depend on the type of problem and level of imbalance, as well as user choices such as tile size and dynamic load balance frequency. We will analyze simulations of a uniform warm plasma, an ambipolar diffusion problem, and a plasma wakefield accelerator. All simulations were performed on Haswell compute nodes of the Cori system at NERSC, each with two sockets, and each socket containing a 2.3 GHz 16-core Haswell CPU (Intel Xeon Processor E5-2698 v3) supporting 2 hyper-threads, leading to a total of 64 hardware threads per node.

3.1. Uniform warm plasma

A uniform warm plasma is a perfectly balanced problem that can be simulated using standard static spatial domain decomposition techniques with excellent parallel efficiency, since the load for every PE is uniform throughout the simulation. Simulating a warm plasma is thus ideal for benchmarking the overhead of the tiling algorithm compared to the default parallelization strategies – both pure MPI and MPI/OpenMP hybrid parallelization options are available in Osiris (see [2] for details) – and optimizing tile size for best performance. The default MPI/OpenMP hybrid parallelization scheme divides the simulation space evenly amongst PEs, then processes particles within an entire PE domain in parallel amongst threads, with each thread using a separate electric current grid that must be summed at each time step (see Section 2.2). We simulate a uniform warm plasma in 2D and 3D using four compute nodes, for a total of 128 cores (256 threads). The product of the number of MPI processes (M) and the number of threads per process (N) is kept constant at $M \times N = 256$, for $N = 2$, 4, 8, 16 and 32. The 2-D runs use a 1024^2 cell grid with cell size $(0.0174\,\mathrm{c}/\omega_p)^2$ and a total of $\sim$150 million particles, and the 3-D runs use a 128^3 cell grid with cell size $(0.0211\,\mathrm{c}/\omega_p)^3$ and a total of $\sim$134 million particles; both runs use a time step of $0.012\,\omega_p^{-1}$. Particle velocities were initialized from a thermal distribution with a proper thermal velocity of $u_{\mathrm{th}} = 0.1\,\mathrm{c}$. All simulations were done using second-order (quadratic) interpolation for particles with periodic boundary conditions in all directions, and were run for 1000 time steps. Tile size was 16^2 cells in 2D and 8^3 cells in 3D. Fig. 3 shows the performance of the code for the various cases.

The performance of the default hybrid parallelization in Osiris is shown to decrease with increasing thread number. This is to be expected as the shared-memory algorithm is required to do additional work (zeroing additional current arrays and reducing the results from all threads) with increasing thread count. Since all tiled simulations in this configuration have only "light" tiles (16 tiles for every thread), the new tiling algorithm does not suffer from this limitation and shows good scalability all the way up to 32 threads. The drop in performance is mostly due to the fact that not all routines have been shared-memory parallelized, as they do not represent a significant overhead for small thread counts.

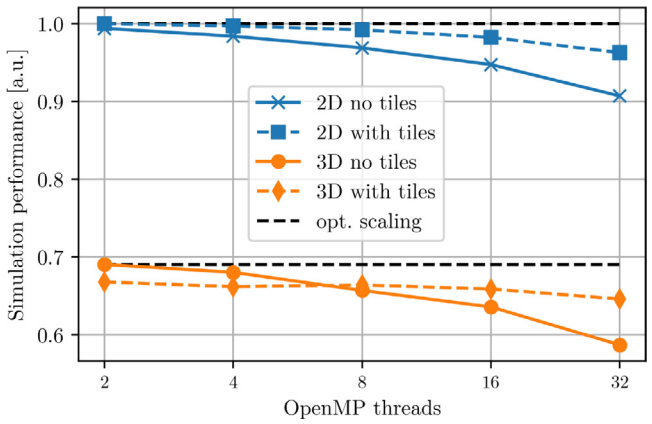

Fig. 3. Thermal plasma simulation performance in 2D and 3D for varying number of threads, with (dashed line) and without (solid line) tiling algorithm. Tiles were 16 cells square in 2D and 8 cells cube in 3D. Number of MPI processes times number of threads per process was kept constant at 256. Performance is calculated as the number of particles pushed (including field solve and other elements) per second, normalized to the fastest run.

Table 1
Optimal tile sizes (cells on each side) in 2D and 3D for a thermal plasma with varied particles per cell. Decreasing the tile size introduces more overhead but improves cache locality, which is important for simulations with many particles per cell.

	2D			3D	
Particles per cell	1	16	64+	1	8+
Optimal tile size	32–64	32	16	16	8

The new algorithm is only outperformed by default Osiris for 3D geometry with small (2,4) number of threads. This is related to the overhead of moving particles between domains; default Osiris without tiles uses larger domains, so there will be fewer particles (compared to the particles in the domain) crossing boundaries. However, for larger thread counts the limitations of default Osiris outweigh this overhead, and the new algorithm performs much better.

To determine the optimal tile size, we repeated the above simulations for varied tile sizes and number of particles per cell. The latter parameter is important because it directly impacts the computational load of each tile. The number of particles per cell were varied between 1, 16, 64 and 144, and the tile sizes were varied by powers of 2, with values ranging from 8–64 cells on each side. Smaller values are not possible using second-order (quadratic) interpolation, as a minimum of 5 cells is required. Table 1 summarizes our results. We found that the optimal tile side length varied between 16–64 cells in 2D and 8–16 cells in 3D. These results are consistent with our expectations: when the amount of computation per tile is larger (as is the case with higher numbers of particles per cell), the overhead of moving particles between tiles has a smaller impact, and the benefits of smaller tiles in terms of data locality lead to higher performance. For smaller numbers of particles per cell this is no longer the case, and larger tiles perform better.

3.2. Ambipolar diffusion

Ambipolar diffusion is of particular importance to a large range of physics scenarios, such as laser-solid interactions and inertial confinement fusion [15–17]. This is a particularly difficult problem to simulate using spatial domain decomposition as the plasma, which is initially confined to a small region of space, expands into vacuum (or near vacuum). We will use this problem to test the effectiveness of our algorithm under extreme load imbalance. We perform a simulation of the expansion of an electron–ion plasma undergoing ambipolar diffusion in 3D, using the various parallelization strategies discussed in Section 2. We start with a constant-density sphere of electrons and ions, where the electrons have a temperature of 130 keV, the ions are cold, the ion-to-electron mass ratio is 1836, and each species has 1000 particles per cell. The sphere of particles has radius $1.4\,c/\omega_p$ and is stationed in the center of a cubic periodic box of side length $16\,c/\omega_p$ with 160^3 cells. The electrons quickly diffuse outward, but are soon slowed by the ambipolar space charge fields of the ions. The ions are in turn pulled out and start to expand, which slows down the expansion of the electrons. Afterwards the electrons will again start to diffuse, leading to an oscillatory expansion of both species. We ran each simulation a total time of $300\,\omega_p^{-1}$ with a time step of $0.0577\,\omega_p^{-1}$, and we used first-order (linear) interpolation for particles.

Fig. 4 shows the combined electron–ion particle density for an analogous 2-D simulation at times 0, 153, and $300\,\omega_p^{-1}$ after the beginning of the expansion, where ω_p is the electron plasma frequency before expansion. The domains of the 16 PEs overlay the density; density slices along any dimension in 3D show similar behavior, but the parallel domain decomposition can be better visualized and understood in 2D. For the first half of the 3-D simulation, about 70% of the particles are contained within a sphere of radius $2.5\,c/\omega_p$, or just 1.6% of the entire simulation volume. By the end of the simulation, 70% of the particles are contained within a sphere of radius $5\,c/\omega_p$ (13% of the total volume). Since the particles largely reside in the center of the simulation box, a traditional spatial domain decomposition with uniform partition sizes will only allow good parallel load balance for up to 2 PEs per dimension (8 PEs total), and will show severe imbalance if the number of PEs is increased to 4 or more. This simulation is also challenging for our tile-based approach: using a total of 4096 cube-shaped tiles with 10 cells to a side, 70% of the particles are contained within just 56 tiles for the first third of the simulation, steadily increasing to 432 tiles by the end of the simulation.

We test the strong scaling parallel speedup of this simulation, keeping the problem size constant and running on 128 to 2048 cores for various cases: the default hybrid MPI/OpenMP algorithm without ("Osiris, no dlb") and with ("Osiris dlb") the previously implemented dynamic load balance [2], light tiles only and heavy tiles only, each with dynamic load balance using the Hilbert space-filling curve ("light tiles" and "heavy tiles", respectively), and a combination of light/heavy tiles with dynamic load balancing using all three schemes ("Hilbert", "Snake", and "$P \times Q$ jagged"). Recall that the previously implemented dynamic load balance uses the rectilinear partition from [9], similar to $P \times Q$ jagged but with only one neighboring PE on each domain side. As mentioned in Section 2.2, the shared-memory parallelization of light and heavy tiles is as follows: light tiles are assigned only one thread each, with tiles processed in parallel on a first-come-first-served basis; heavy tiles are processed in serial, with all threads working on a single tile at once. The two cases without tiles use the maximum number of threads per MPI process without hyperthreading (16), as this is most favorable for the default hybrid algorithm. The cases with tiles use 2, 4, 8, 16, and 16 threads per MPI process for the five runs with increasing core counts, respectively. Dynamic load balancing was performed at every 20 time steps and occupied just a few percent of the total simulation time. The chosen cell weight parameter was $\mathcal{C} = 2$. For the "Hilbert" run using 2048 cores, dynamically load balancing every 10 and 40 time steps caused the total run time to increase by 10% and 11%, respectively. Fig. 5 shows the results in terms of (a) overall simulation performance and (b) load per core, normalized to the fastest simulation with 128 cores.

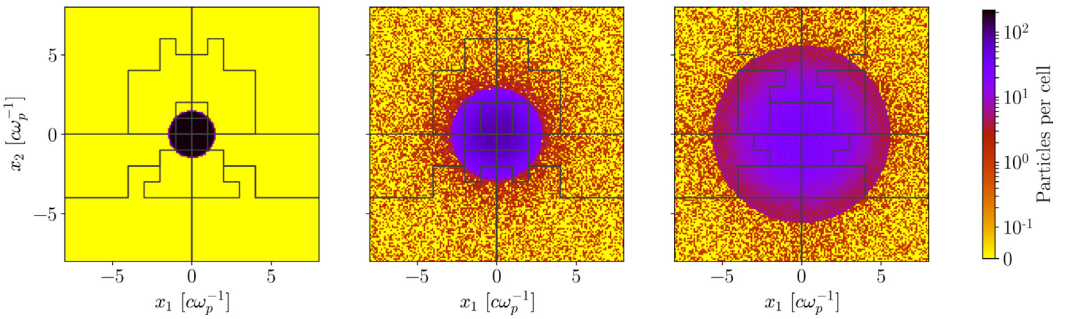

Fig. 4. The 16 PE subdivisions overlaying particle density of the ambipolar diffusion problem in 2D at various times. Cell weight is $c = 2.0$ for this case.

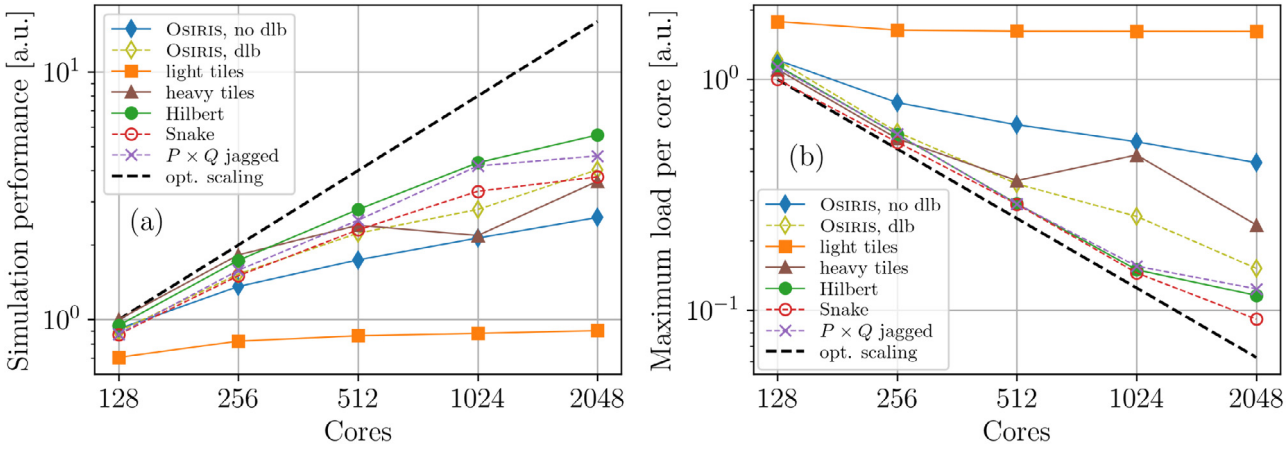

Fig. 5. Strong scaling test of a 3-D ambipolar diffusion simulation. Performance in (a) is calculated as the number of particles pushed (including field solve and other elements) per second, and load in (b) is calculated as the maximum time any core spent in the advance-deposit routine, both normalized to the fastest run with 128 total cores. The "OSIRIS, no dlb" and "OSIRIS, dlb" runs are both done without tiles, one without load balancing and one with the previously implemented dynamic load balance. The "light tiles" and "heavy tiles" runs use the new dynamic load balance via the Hilbert space-filling curve, but with only light or only heavy tiles, respectively. The "Hilbert", "Snake", and "$P \times Q$ jagged" runs use the new dynamic load balance along with a combination of light and heavy tiles, where the names refer to the load-balancing scheme used. The two cases without tiles are performed with 16 threads per MPI process, and tiled runs use an optimal configuration for each run (2, 4, 8, 16, and 16 threads per MPI process for the five runs with increasing core counts, respectively).

The default hybrid algorithm without dynamic load balancing is unable to scale due to load imbalance issues. Fig. 5 also shows that the scalability of the light tiles algorithm is severely limited as a result of its inability to parallelize within each tile, and in fact the light tiles algorithm is the slowest algorithm at all core counts. To explain the poor performance when using only light tiles, consider the simulations with 1024 or 2048 total cores, where each PE has 16 threads. During the first third of the simulation, the 56 computationally expensive tiles could be spread across just 4 PEs (64 cores) since the algorithm groups together tiles close in space and requires that each PE has at least as many tiles as threads. Each light tile can only be processed by a single core, leading to significant performance degradation that worsens with increasing core counts. Using only heavy tiles significantly boosts performance since a PE may be assigned just a single tile, but note the drastic drop in performance for the first run using 16 threads (1024 total cores). This overhead is a result of summing the 16 separate electric current grids for each tile, which is especially detrimental for PEs with many tiles containing few particles.

The limitation in parallel scalability when using only light or only heavy tiles is overcome by our new algorithm that uses a combination of light and heavy tiles. We implement an additional level of parallelization within each tile, which allows a PE to efficiently devote all of its cores to process a single computationally expensive tile. As seen in Fig. 5, our algorithm (a) maintains much better overall parallel scalability compared to any other method and (b) achieves near ideal load balance for up 1024 cores. At all core counts, the combination of heavy/light tiles always outperforms the use of light tiles only, being ∼2 times faster for 128 cores and ∼6 times faster for the largest core count. The average number of heavy tiles over the course of the simulation steadily increases with thread count, ranging between 0.5% and 8.2% of all tiles from 128 to 2048 cores, respectively. The small drop in (b) from ideal scalability at 2048 cores has to do with the problem/tile size: at this core count we will have on average only 2 tiles per PE, which does not allow for effective load balancing. Reducing the tile size may improve performance for this case.

When comparing the various load-balancing schemes, using the Hilbert space-filling curve was consistently the fastest, and with 2048 cores it was 1.2, 1.5, and 1.4 times faster than the $P \times Q$ jagged scheme, snake space-filling curve scheme, and previous dynamic load balance, respectively. An overall speedup of a factor of 2.2 was gained compared to original OSIRIS without dynamic load balance. Though the snake space-filling curve consistently gave excellent load balance, extra overhead from MPI communications due to the shapes of PE boundaries ultimately caused those runs to be slower than the other topologies.

3.3. Plasma wakefield acceleration

Since its inception in the seminal paper by Tajima and Dawson [18], the field of plasma wakefield acceleration has been an active area of intense research. Plasma based acceleration (PBA) [19] is an accelerator scheme which uses an intense electron bunch – particle wakefield acceleration (PWFA) [20] – or a

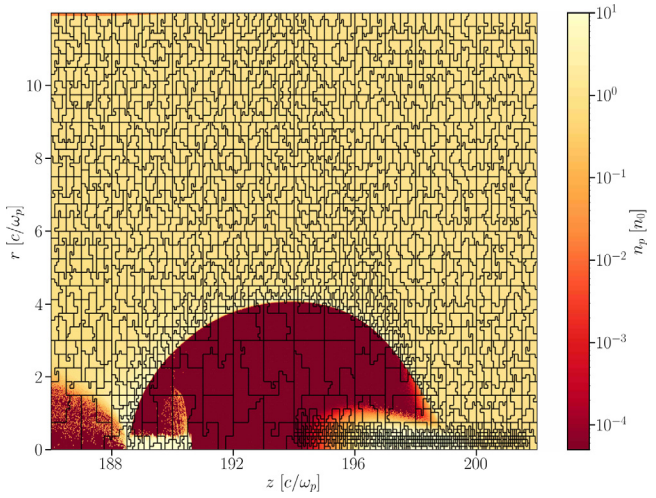

Fig. 6. MPI subdivisions (black lines) overlaying electron density for a quasi-3D PWFA simulation with 2048 MPI processes.

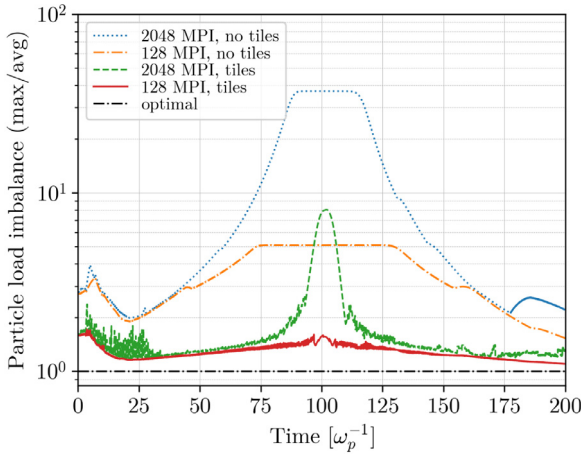

Fig. 7. Particle load imbalance (maximum/average load) of a 3-D PWFA simulation using 2048 total cores with and without dynamic load balancing using tiles for two different MPI/OpenMP configurations. The dot-dashed line shows an ideal load imbalance of 1.0.

laser – laser wakefield acceleration (LWFA) – to accelerate particles. As the beam propagates through the plasma, the electrons are pushed away from the beam to leave an exposed ion column, resulting in a region of space supporting large electric fields that travels at nearly the speed of light. Particles can be injected and trapped from the background plasma through various mechanisms [21–24], then accelerated to high energies. The excitation of the wake by the drive beam is strongly nonlinear, as can be the evolution of the driver and the injected and accelerated beams, making PIC simulations the tool of choice for modeling these scenarios.

Simulations of PWFA naturally contain regions of very high particle density – the injected particle bunch and beam driver – that are dynamic in nature, surrounded by regions of relatively low-density background plasma. These density distributions can result in severe load imbalance, for which finding a single PE decomposition that balances load for the entire simulation may be near impossible. Effective dynamic load balancing of the simulation is therefore crucial for efficient numerical modeling of PWFA, particularly when using large core counts. We test the performance of our dynamic load balancing algorithm with a PWFA scenario similar to that studied by Dalichaouch et al. [25]. The driving beam is initialized with a radius of $\sigma_r = 2.1\,\mathrm{c}/\omega_p$ and then evolves self-consistently, focusing down to a radius of $\sigma_r = 0.2\,\mathrm{c}/\omega_p$ in the moving simulation window. When initializing a beam, to provide good statistics we use a fixed number of particles per cell and variable weights on the particles to vary the density. The beam evolution leads to a large load imbalance due to the large number of simulation particles concentrated in a small cell volume. Additionally, the accelerating structure formed with particles from the background plasma will show a density spike at the back of this structure that can be several orders of magnitude larger than the background density, creating a second load imbalance region.

To illustrate the complexity of finding an optimal domain decomposition, we show in Fig. 6 a snapshot from an analogous simulation using the quasi-3D version of Osiris that uses a geometry (2-D r-z in space, azimuthal expansion in θ) [26]. Load balance was performed using the Hilbert space-filling curve and a cell weight $\mathcal{C} = 1$ for 2048 PEs; the simulation size was 2048×1536 cells, with 256×128 tiles each of size 8×12 cells. The total simulation space was of size $16 \times 12\,\mathrm{c}/\omega_p$, with 8 particles per cell for each of the driver and background electron species, and we used second-order (quadratic) interpolation for particles. This snapshot is about halfway through the simulation when the particle driver has focused tightly near to the axis, and the figure shows an overlay of PE boundaries with electron density. Note the large concentration of small domains near the beam driver, bubble sheath and injected particles, and the large domains in the near-vacuum region. Similar behavior occurs for 3D simulations, but the visualization of these structures is cluttered and brings little insight.

We tested the efficiency of our algorithm on 3D simulations of this PWFA scenario by performing a set of simulations with different parallelization options. All simulations were performed on 64 compute nodes with the architecture described above (2048 cores total) and varying numbers of MPI processes/threads, for the default hybrid parallelization without load balance and for the dynamic light/heavy tiles parallelization. The simulation size was $512 \times 768 \times 768$ cells, and the tile size was $16 \times 12 \times 12$ cells for the tile-based simulations. The total simulation space was of size $16 \times 24 \times 24\,\mathrm{c}/\omega_p$, with 1 particle per cell for each of the driver and background electron species and second-order interpolation for particles. The simulation was run for a total of 12 800 time steps to a time of $200\,\omega_p^{-1}$, with dynamic load balancing performed at every 40 time steps. For load calculations we chose a cell weight parameter of $\mathcal{C} = 0.5$.

Fig. 7 shows the evolution of the parallel particle load imbalance over the course of a simulation using 2048 total cores with two MPI/OpenMP configurations (see Table 2 for more detail) as the ratio between the maximum computational load on a single PE and the average computational load across all PEs. The load imbalance is shown for both the static uniform partition (no tiles) and the dynamic partition (tiles). As shown in the plot, the high-density regions near the propagation axis lead to a severe load imbalance about halfway through the simulation. For 2048 processes each with 1 thread, the imbalance reaches a peak (average) value of 37.1 (10.6) without dynamic load balancing. This result can be improved through the use of the hybrid MPI/OpenMP algorithm [2] that smears out the load imbalance by allowing for large domains to be assigned to each PE, leading to a peak (average) imbalance value of 5.1 (3.5) at 128 processes each with 16 threads. It should, however, be noted that this large number of threads leads to some performance degradation due to overhead, as described in Sections 2.2 and 3.1. To further reduce the parallel load imbalance we must use the dynamic tile load balance. Using this algorithm we can lower the peak (average) value of the

Table 2
Total simulation time with and without tiles (and dynamic load balancing) for the 3-D PWFA simulation. The total number of cores was kept constant (2048), only varying the number of OpenMP threads and MPI processes.

MPI processes	OpenMP threads	Time [a.u.] no tiles	Time [a.u.] tiles
2048	1	4.63	1.19
1024	2	4.55	1.03
512	4	2.45	1.00
256	8	2.57	1.03
128	16	2.18	1.10

imbalance down to 8.0 (1.7) with 2048 processes, as shown in Fig. 7. Furthermore, using our heavy/light tile parallelization we can assign multiple threads to the heavy tiles, achieving a peak (average) load imbalance of just 1.7 (1.3) with 128 processes. Note that for this run, nearly all tiles were processed as light except for an average of 175 heavy tiles near the focus of the beam driver during the middle 10% of the simulation time.

Finally, we compare the total simulation times for the different configurations of MPI processes and OpenMP threads for the static uniform partition and the dynamic tiles partition. Table 2 shows the results, with time values normalized to the fastest simulation (512 MPI processes with 4 OpenMP threads and dynamic load balance). As shown in the table, performance of the static partition (no tiles) can be improved by over a factor of 2 simply by increasing the number of threads. However, the tile-based algorithm maintains good performance for all thread counts and always outperforms the static partition, being 2.18 times faster between the fastest runs. Cumulative load balance is roughly the same for tile-based runs with 4, 8, and 16 OpenMP threads, but the overhead discussed in Section 2.3 is slightly larger with 16 threads. The overhead associated with the dynamic load balance effort, however, was found to be only $\sim$1% of the total simulation time in all cases. We also see that the timings for the tile-based algorithm vary less than 10% with varying number of threads, which is to be expected, as the computation should scale well with number of threads. This small variation highlights the versatility of the algorithm in terms of efficiency on various architectures supporting different numbers of threads.

4. Conclusion

Many of today's frontier problems in plasma physics necessitate fully kinetic simulations, which can in many cases only be achieved through particle-in-cell simulations. Although the PIC algorithm has been quite successful in addressing many kinetic plasma problems (and can be extended straightforwardly with additional physics models), it is numerically expensive, often requiring large-scale, massively-parallel computational resources. However, the traditional parallelization of the PIC algorithm is susceptible to parallel load imbalance. In problems such as plasma-based acceleration, laser-solid interactions, and magnetic reconnection with pair production, simulation particles may accumulate in small regions of space, leading to an uneven computational load. Dynamic load balancing across distributed-memory processing elements (PEs) and the efficient use of shared-memory cores are therefore essential to perform large-scale simulations of these physics problems.

In this paper we presented a novel hybrid parallelization strategy for the PIC algorithm that combines two different shared-memory parallelization algorithms, achieving excellent performance even for simulations with extreme imbalance. Our algorithm uses small, regularly spaced, self-contained regions of the simulation space that we refer to as tiles. These tiles contain all particle and grid quantities for a particular region of space and are dynamically traded between all PEs at chosen intervals to maintain computational load balance. Unique to our tile-based implementation of dynamic load balancing is the ability to assign either one or multiple threads to each tile depending on computational weight. This is especially useful for simulations with localized high-density regions for which the load-balanced domain decomposition may result in a PE having a single tile containing a majority of particles, or for simulations where a given PE has fewer tiles than threads. Furthermore, the ability to assign multiple threads to each tile allows us to use larger tiles, which can reduce the overhead of the tiling algorithm while still maintaining good load balance and high performance.

Our algorithm was shown to perform well for balanced simulations: it was on par with or faster than the traditional parallelization algorithm on the same hardware and scaled better with increasing thread count, with an overhead of less than $\sim$5% for the highest thread count. Our algorithm gave a speedup of more than a factor of 2 compared to simulations without dynamic load balance of an expanding plasma undergoing ambipolar diffusion, a critically difficult problem to parallelize with 70% of the particles contained within only 13% of the simulation volume. In particular, it was $\sim$6 times faster than running the simulation using only 1 thread per tile and attained near-ideal load balance for 8 times as many cores compared to running without tiles. We also analyzed the performance of our algorithm on plasma accelerator simulations and verified a speedup of over a factor of 2 when compared to performance without dynamic load balance, with the dynamic load balance itself only taking about 1% of the total simulation time. Even greater speedups are expected for larger simulations, where shared-memory thread number is small compared to total computing resources. Our results also show that the performance of the algorithm does not vary significantly with different combinations of PE and thread numbers.

This algorithm was tested on a CPU architecture using MPI/OpenMP for distributed/shared memory parallelism, and can be straightforwardly extended to other architectures and programming models, such as the MPI-3 Shared Memory model, Coarray Fortran, extension to GPUs using, for example, CUDA, or extension to other architectures using Intel oneAPI. We are currently investigating the best way to implement this load-balancing algorithm on many-core accelerators, though similar algorithms have been shown to operate efficiently on such architectures [11,27]. The software may need to be carefully structured to ensure that all threads are executing the same kernel while allowing for a combination of heavy/light tiles. The algorithm can also be combined with vectorized versions of the PIC algorithm, efficiently exploiting all parallelism levels available in present and near-future HPC systems and opening new avenues for the numerical simulation of kinetic plasmas.

CRediT authorship contribution statement

Kyle G. Miller: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Roman P. Lee:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Adam Tableman:** Methodology, Software, Validation, Writing - review & editing. **Anton Helm:** Conceptualization, Methodology, Writing - review & editing. **Ricardo A. Fonseca:** Conceptualization, Methodology, Software, Resources, Writing - review & editing, Visualization, Supervision. **Viktor K. Decyk:** Methodology, Writing - review & editing. **Warren B. Mori:** Conceptualization, Resources, Writing - review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in parts by the US National Science Foundation [Grant Numbers ACI-1339893, 1806046]; the US Department of Energy [Grant Numbers DE-NA0003842, DE-SC0019010, DE-SC0010064]; Lawrence Livermore National Laboratory [subcontract B634451, subcontract B635445]; Fundação para a Ciência e Tecnologia, Portugal [Grant Number PTDC-FIS-PLA-2940-2014]; and European Research Council [ERC-2015-AdG, Grant Number 695008]. The simulations were performed on Haswell CPUs (Intel Xeon Processor E5-2698 v3) of the Cori system at NERSC.

References

[1] R.A. Fonseca, L.O. Silva, F.S. Tsung, V.K. Decyk, W. Lu, C. Ren, W.B. Mori, S. Deng, S. Lee, T. Katsouleas, J.C. Adam, in: P.M.A. Sloot, A.G. Hoekstra, C.J.K. Tan, J.J. Dongarra (Eds.), Computational Science — ICCS 2002: International Conference Amsterdam, the Netherlands, April 21–24, 2002 Proceedings, Part III, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 342–351, http://dx.doi.org/10.1007/3-540-47789-6_36.

[2] R.A. Fonseca, J. Vieira, F. Fiuza, A. Davidson, F.S. Tsung, W.B. Mori, L.O. Silva, Plasma Phys. Control. Fusion 55 (12) (2013) 124011, http://dx.doi.org/10.1088/0741-3335/55/12/124011.

[3] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov, N. Aunai, J. Dargent, C. Riconda, M. Grech, Comput. Phys. Comm. 222 (2018) 351–373, http://dx.doi.org/10.1016/J.CPC.2017.09.024.

[4] K.J. Bowers, B.J. Albright, L. Yin, B. Bergen, T.J. Kwan, Phys. Plasmas 15 (5) (2008) http://dx.doi.org/10.1063/1.2840133.

[5] D.P. Grote, A. Friedman, J.L. Vay, I. Haber, AIP Conference Proceedings, Vol. 749, 2005, pp. 55–58, http://dx.doi.org/10.1063/1.1893366.

[6] C. Othmer, J. Schüle, Comput. Phys. Comm. (2002) http://dx.doi.org/10.1016/S0010-4655(02)00389-2.

[7] R.D. Ferraro, P.C. Liewer, V.K. Decyk, J. Comput. Phys. 109 (2) (1993) 329–341, http://dx.doi.org/10.1006/JCPH.1993.1221.

[8] P.C. Liewer, V.K. Decyk, J. Comput. Phys. 85 (2) (1989) 302–322, http://dx.doi.org/10.1016/0021-9991(89)90153-8.

[9] E. Saule, E.Ö. Baş, Ü.V. Çatalyürek, J. Parallel Distrib. Comput. 72 (10) (2012) 1201–1214, http://dx.doi.org/10.1016/J.JPDC.2012.05.013.

[10] I. Surmin, A. Bashinov, S. Bastrakov, E. Efimenko, A. Gonoskov, I. Meyerov, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2015, pp. 107–119, http://dx.doi.org/10.1007/978-3-319-21909-7_12.

[11] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, A. Bhattacharjee, J. Comput. Phys. 318 (2016) 305–326, http://dx.doi.org/10.1016/j.jcp.2016.05.013.

[12] V.K. Decyk, T.V. Singh, Comput. Phys. Comm. 185 (3) (2014) 708–719, http://dx.doi.org/10.1016/J.CPC.2013.10.013.

[13] D. Hilbert, Math. Ann. 38 (1891) 459–460, URL: http://eudml.org/doc/157555.

[14] X. Kong, M.C. Huang, C. Ren, V.K. Decyk, J. Comput. Phys. 230 (4) (2011) 1676–1685, http://dx.doi.org/10.1016/j.jcp.2010.11.032.

[15] C.M. Chen, T. Norimatsu, Y. Izawa, T. Yamanaka, S. Nakai, J. Vac. Sci. Technol. A 13 (6) (1995) 2908–2913, http://dx.doi.org/10.1116/1.579612.

[16] J. Lindl, Phys. Plasmas 2 (11) (1995) 3933–4024, http://dx.doi.org/10.1063/1.871025.

[17] R. Betti, O.A. Hurricane, Nat. Phys. 12 (5) (2016) 445–448, http://dx.doi.org/10.1038/NPHYS3736.

[18] T. Tajima, J.M. Dawson, Phys. Rev. Lett. 43 (4) (1979) 267–270, http://dx.doi.org/10.1103/PhysRevLett.43.267.

[19] C. Joshi, E. Adli, W. An, C.E. Clayton, S. Corde, S. Gessner, M.J. Hogan, M. Litos, W. Lu, K.A. Marsh, W.B. Mori, N. Vafaei-Najafabadi, B. O'shea, X. Xu, G. White, V. Yakimenko, Plasma Phys. Control. Fusion 60 (2018) 14, http://dx.doi.org/10.1088/1361-6587/aaa2e3.

[20] P. Chen, J.M. Dawson, R.W. Huff, T. Katsouleas, Phys. Rev. Lett. 54 (7) (1985) 693–696, http://dx.doi.org/10.1103/PhysRevLett.54.693.

[21] W. Lu, M. Tzoufras, C. Joshi, F.S. Tsung, W.B. Mori, J. Vieira, R.A. Fonseca, L.O. Silva, Phys. Rev. ST Accel. Beams 10 (6) (2007) 061301, http://dx.doi.org/10.1103/PhysRevSTAB.10.061301.

[22] H. Suk, N. Barov, J.B. Rosenzweig, E. Esarey, Phys. Rev. Lett. 86 (6) (2001) 1011–1014, http://dx.doi.org/10.1103/PhysRevLett.86.1011.

[23] E. Oz, S. Deng, T. Katsouleas, P. Muggli, C.D. Barnes, I. Blumenfeld, F.J. Decker, P. Emma, M.J. Hogan, R. Ischebeck, R.H. Iverson, N. Kirby, P. Krejcik, C. O'connell, R.H. Siemann, D. Walz, D. Auerbach, C.E. Clayton, C. Huang, D.K. Johnson, C. Joshi, W. Lu, K.A. Marsh, W.B. Mori, M. Zhou, Phys. Rev. Lett. 98 (8) (2007) 084801, http://dx.doi.org/10.1103/PhysRevLett.98.084801.

[24] X.L. Xu, F. Li, W. An, T.N. Dalichaouch, P. Yu, W. Lu, C. Joshi, W.B. Mori, Phys. Rev. Accel. Beams 20 (11) (2017) 111303, http://dx.doi.org/10.1103/PhysRevAccelBeams.20.111303.

[25] T.N. Dalichaouch, X.L. Xu, F. Li, A. Tableman, F.S. Tsung, W. An, W.B. Mori, Phys. Rev. Accel. Beams 23 (2) (2020) 021304, http://dx.doi.org/10.1103/PhysRevAccelBeams.23.021304.

[26] A. Davidson, A. Tableman, W. An, F.S. Tsung, W. Lu, J. Vieira, R.A. Fonseca, L.O. Silva, W.B. Mori, J. Comput. Phys. 281 (2015) 1063–1077, http://dx.doi.org/10.1016/j.jcp.2014.10.064.

[27] V.K. Decyk, Comput. Sci. Eng. 17 (2) (2015) 47–52, http://dx.doi.org/10.1109/MCSE.2014.131.