In-Situ Assessment of Device-Side Compute Work for Dynamic Load Balancing in a GPU-Accelerated PIC Code

Michael E. Rowan
Kevin N. Gott
Jack Deslippe
mrowan@lbl.gov
kngott@lbl.gov
jrdeslippe@lbl.gov
Lawrence Berkeley National Laboratory, NERSC
Berkeley, California, USA

Axel Huebl
Maxence Thévenet*
Remi Lehe
Jean-Luc Vay
axelhuebl@lbl.gov
maxence.thevenet@desy.de
rlehe@lbl.gov
jlvay@lbl.gov
Lawrence Berkeley National Laboratory, ATAP
Berkeley, California, USA

ABSTRACT

Maintaining computational load balance is important to the performant behavior of codes which operate under a distributed computing model. This is especially true for GPU architectures, which can suffer from memory oversubscription if improperly load balanced. We present enhancements to traditional load balancing approaches and explicitly target GPU architectures, exploring the resulting performance. A key component of our enhancements is the introduction of several GPU-amenable strategies for assessing compute work. These strategies are implemented and benchmarked to find the most optimal data collection methodology for in-situ assessment of GPU compute work. For the fully kinetic particle-in-cell code WarpX, which supports MPI+CUDA parallelism, we investigate the performance of the improved dynamic load balancing via a strong scaling-based performance model and show that, for a laser-ion acceleration test problem run with up to 6144 GPUs on Summit, the enhanced dynamic load balancing achieves from 62%–74% (88% when running on 6 GPUs) of the theoretically predicted maximum speedup; for the 96-GPU case, we find that dynamic load balancing improves performance relative to baselines without load balancing (3.8× speedup) and with static load balancing (1.2× speedup). Our results provide important insights into dynamic load balancing and performance assessment, and are particularly relevant in the context of distributed memory applications ran on GPUs.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; **Parallel algorithms**; • **Applied computing** → **Physics**.

---

*Now with Deutsches Elektronen Synchrotron (DESY), Germany

KEYWORDS

Dynamic Load Balancing, Particle-In-Cell (PIC), AMReX, High-Performance Computing, GPU, CUDA

1 INTRODUCTION

GPU-accelerated machines entered the TOP500 rankings just over a decade ago [18]. GPUs offer an effective path to high computational throughput, and GPU accelerated machines are particularly of interest, as they are central to the US Exascale, and EuroHPC Pre-Exascale initiatives. HPC software must be adapted to take full advantage of heterogeneous systems as parallelism continues to increase and we approach an era of exascale computing in leadership-class supercomputers. Maintaining equal distribution of computational load (*dynamic load balancing*) is crucial for distributed memory applications to make efficient use of hardware with continually increasing parallel-compute capabilities [10, 13, 19].

In this paper, we present and investigate GPU-targeted enhancements for in-situ dynamic load balancing. These techniques are demonstrated in the particle-in-cell (PIC) framework WarpX [21], which, as a particle-mesh simulation code, models fields via an Eulerian description and particles via a discretization in Lagrangian markers. In WarpX, multi-node parallelism is achieved through spatial domain decomposition, which is supported through the block-structured framework AMReX [24]. WarpX's particles are used to model kinetic phenomena and often cause localized spikes in memory consumption and compute demand. It is thus likely, that herein studied scenarios exemplifying realistic geometries of contemporary research, using highly mobile particles are applicable to other distributed memory codes. In particular, WarpX models plasma physics problems, including laser-plasma acceleration and particle acceleration in astrophysical contexts [21].

The outline of this paper is as follows: In Sec. 2, we introduce load balancing essentials for particle-mesh codes and present our

dynamic load balancing improvements for GPU systems. In Sec. 3, we introduce laser-ion acceleration as a scientifically relevant test problem with which to explore the performance of our improvements to WarpX's dynamic load balancing. Next, in Sec. 4, we discuss the performance of WarpX's enhanced dynamic load balancing in the context of a strong scaling-based performance model, and present weak scaling tests of dynamic load balancing performance. We conclude in Sec. 6, with a summary of the generality of the proposed approach, advantages relative to related works, and discussion of future work.

2 IN-SITU DEVICE-SIDE DYNAMIC LOAD BALANCING

In this section, we describe novel methods we have devised for measuring device-side compute work at runtime and the dynamic load balancing algorithms used with these measurements.

In Sec. 2.1, we introduce general features of distributed memory particle-mesh codes that are necessary for a discussion of load balancing and introduce the terminology used in the software framework AMReX. In Sec. 2.2, we present details of our enhancements to dynamic load balancing when using GPUs, with a focus on the challenges of load balancing in a GPU-based code and the GPU-specific aspects of the load balance algorithm.

2.1 Overview of the Domain Decomposition in Particle-Mesh Codes

A common motif in parallel particle-mesh codes is to partition the simulation domain into separate sub-domains, typically contiguous spatial sub-domains. These sub-domains are interchanged between compute elements at runtime to achieve an even distribution of compute work across computational resources. A load-balancing algorithm is implemented to achieve this even distribution. An example domain consisting of 16 (rectilinear) cells decomposed into 4 sub-domains (delimited by solid lines; each sub-domain contains 4 cells) is shown in panel (a) of Fig. 1. A sub-domain usually contains the field data that describes its contiguous space, as well as associated particle data for the particles that lie within the bounds of that sub-domain. For the example in panel (a) of Fig. 1, the boxes
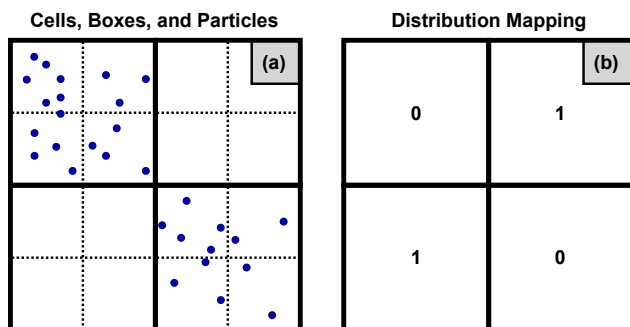

Figure 1: Example domain with 16 cells (delimited by dotted lines) and 4 sub-domains (delimited by solid lines; each sub-domain contains 4 cells) (a); blue circles represent particles, and are associated with the sub-domain with which they overlap. A distribution mapping indicates which MPI rank manages each box (b); this example demonstrates load imbalance because rank 0 manages 30 particles, whereas rank 1 manages no particles.

in the upper left and lower right have 18 and 12 associated particles, respectively.

Within the MPI parallelism model, which is ubiquitous in distributed memory codes, it is natural to assign each sub-domain to one exclusive MPI rank; for MPI+X parallelism (where X corresponds to an accelerator platform, e.g., CUDA), the ranks correspond to unique GPUs.

Our study is designed as a general improvement in AMReX, a performance-portable, block-structured framework which provides infrastructure for mesh-based simulations, with capability to support particles [24]. For the remainder of this paper, we will use AMReX terminology, as it is consistent with our study and the naming convention is highly descriptive. In AMReX, a 'sub-domain' is a rectilinear region of cells, defined by a *box*. A *distribution mapping* describes the current MPI process ownership of field and particle data associated with each box; since in our study we use one MPI rank per GPU, the distribution mapping also maps the GPU ownership of each box. This consists of a box-sized vector containing the MPI rank (equivalently, GPU ID) to which each box is assigned. For the example in panel (a) of Fig. 1, one possible distribution mapping is shown in panel (b); the boxes (including both their cells and particles) in the upper left and lower right belong to rank 0 (also GPU 0), and the boxes in the upper right and lower left belong to rank 1 (also GPU 1).

Our load-balancing strategy is tested in WarpX, a particle-mesh application, implemented on top of AMReX. Cells are updated in self-consistent, explicit time steps solving Maxwell's equations. Throughout all cells, representative particle markers (weighted particles) move according to the Lorentz-force and in turn modify fields through deposited, charged currents [3]. When ran on systems with accelerators, locally assigned boxes of particles and structured field (cell) data are stored persistently in device memory. Implementing advanced, numerical schemes for those operations, single-source kernels consistently accelerate the whole application.

Load balancing in WarpX consists of a global update of the distribution mapping across all MPI ranks, and redistribution of particle and cell data in accordance with the new distribution mapping (this is further discussed in the next section, Sec. 2.2). The example configuration in Fig. 1 demonstrates particle load imbalance; while ranks 0 and 1 each manage 8 cells, rank 0 manages 30 particles and rank 1 manages no particles.

2.2 Load Balancing Strategy

A basic outline for dynamic load balancing in a distributed memory particle-mesh code (representative of what we use in the present work) is sketched in Lis. 2.1; this resides within the main time-stepping loop in WarpX.

The frequency of the load balancing calculation is controlled by a user-selected load balance interval (line 1). Computational costs corresponding to each box are gathered from all GPUs to the root process, so that the root process has a global view of the current GPU ownership and cost for each box over the full simulation domain. With this information, the root process computes a load balanced distribution mapping and the corresponding *load balance efficiency E* for the old and new mappings (lines 9–11).

Listing 2.1: Dynamic load balancing routine

```
 1  if (step % loadBalanceInterval == 0) {
 2      // With measured cost per GPU, compute
 3      // new `distribution mapping` (DM) and
 4      // write current and proposed load
 5      // balance efficiencies; the new DM may
 6      // be computed according to `knapsack`
 7      // or `SFC` policies
 8      float currEff = 0.0, propEff = 0.0;
 9      DistMapping newDM = makeNewDM(costs,
10                                      currEff,
11                                      propEff);
12
13      // Load balanced distribution mapping
14      // is now up-to-date only on root;
15      // global update only if there is
16      // enough improvement
17      bool globUpdateDM = false;
18      if (myRank == root) {
19          globUpdateDM = (propEff
20                              > 1.1*currEff);
21      }
22
23      bcast(&globUpdateDM, 1, root);
24      if (globUpdateDM) {
25          bcast(&newDM[0],
26                  newDM.size(),
27                  root);
28          // Remake grids and redistribute
29          // particles according to new DM
30          updateDistributionMapping(newDM);
31      }
32  }
```

This load balance efficiency is defined as:

$$E \equiv c_{\mathrm{avg}}/c_{\mathrm{max}} \qquad (1)$$

where c_{avg} and c_{max} are the average and maximum costs, respectively, over all GPUs. The cost per GPU is taken as the sum of costs over all boxes managed by it; an efficiency of 1 indicates a perfectly balanced distribution. By default, the new distribution mapping will not be broadcast to all GPUs (line 17), but if the proposed efficiency exceeds the current efficiency by a user-selected amount (lines 18–21), the new distribution is communicated to all GPUs and updated (lines 23–31). Update of the distribution mapping includes shuffling ownership of boxes as well as redistributing particles to GPUs (line 30), and when ran is the most expensive step of the load balancing routine (this is the case for simulations we present in Sec. 3.3; redistribution of box and particle data typically constitutes $\gtrsim$99.7% of the time required to load balance). Therefore, only redistributing in cases that will yield a substantial improvement is critical to achieving an optimal load balancing implementation. We control this with a hyperparameter for the required improvement to the load balance efficiency E, i.e. a value that must be met or exceeded in order for the proposed distribution mapping to be communicated and updated (the performance impact of this hyperparameter is discussed in Sec. 3.3).

In distributing costs equally over all GPUs (i.e., computing a new distribution mapping; Lis. 2.1, lines 9–11), a variety of strategies have been developed [2, 10, 13, 19]. Two of the most common algorithms are *knapsack*, which assigns sub-domains to GPUs such that the corresponding computational costs are spread as equally as possible over all GPUs, and *space-filling curve*, in which sub-domains are enumerated with a Morton Z-order space-filling curve

and the resultant ordering is partitioned so that costs are distributed as equally as possible among GPUs. For our simulations of laser-ion acceleration, the performance impact of these algorithms when running on GPUs is discussed in Sec. 3.3.

Measuring computational cost offers a unique challenge with GPUs, as GPUs leverage asynchronous compute [12]. For this reason, timing sections of code with CPU timers will not yield a useful load balancing metric [8]. To address this challenge, we implemented three different GPU-amenable cost measurement strategies to estimate the compute work associated with a box, (*Heuristic*, *GPU clock*, and *CUPTI*), which we summarize below:

- Heuristic: Compute work is estimated as a weighted linear sum of the number of particles and cells per box. Note that the optimal choice of weights may vary depending on hardware, choice of field solver, and the interpolation order of particle shapes.
- GPU clock: The device `clock()` function is used to measure thread execution time, which is accumulated using GPU atomic add operations; the procedure is shown schematically in Fig. 2 (a). To mitigate latency, thread times are accumulated in shared memory before their sum is transferred to global memory. With this GPU clock-based timer, we measure the thread-summed execution time of a compute-intensive kernel, which serves as a proxy for the compute work associated with a box. For the laser-ion acceleration problem discussed in Sec. 3, current deposition typically accounts for 50% of the total walltime, with the remaining walltime dominated by communication routines, thus we take the time spent in current deposition as representative of the compute work that should be considered during load balance.
- CUPTI: With the CUDA Profiling Tools Interface (CUPTI) API [6], we constructed a timer capable of accessing kernel execution times on-the-fly; the CUPTI-based timing strategy is summarized in Fig. 2 (panel (b)). CUPTI enables collection of *kernel activity records*— data structures that contain kernel information including the absolute start and end times of the kernel. Registered callback functions, which handle the request and delivery of buffers used to store activity records, are activated by GPU activity; when the buffer return is complete, the activity records stored therein can be used to compute kernel duration. With this CUPTI-based kernel timer, we measure the duration of the current deposition kernel and use it as a proxy for the computational cost associated with a box.

The heuristic method has been used frequently in particle-mesh codes which run on CPU [4, 7, 9, 13, 15, 17, 23], and transfers readily to particle-mesh codes on GPU [5, 10, 19], given the user has tuned appropriately the particle and cell weights. However, as mentioned above, the optimal choice of weights can vary depending on algorithmic choices and hardware, which limits the general applicability of heuristic cost measurement. The need for careful tuning by the user can be eliminated with an on-the-fly measurement technique such as GPU clock or CUPTI mentioned above; to the authors' knowledge, the present work represents the first implementation and application of such techniques for dynamic load balancing in a particle-mesh code running at scale on GPUs.
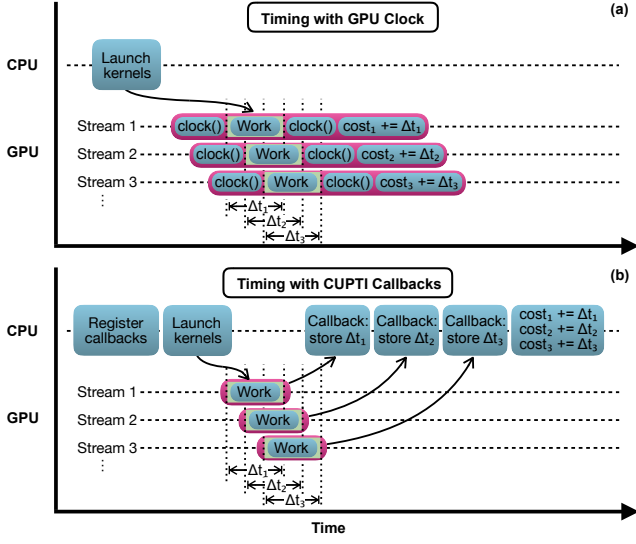
Figure 2: Schematic summary of GPU clock (a) and CUPTI (b) methods for assessing GPU compute work. The diagram in (a) is only representative, in that the actual measured quantity in our implementation is summed thread execution time, rather than kernel execution time; with CUPTI (b), we do measure kernel times as a proxy for GPU compute.
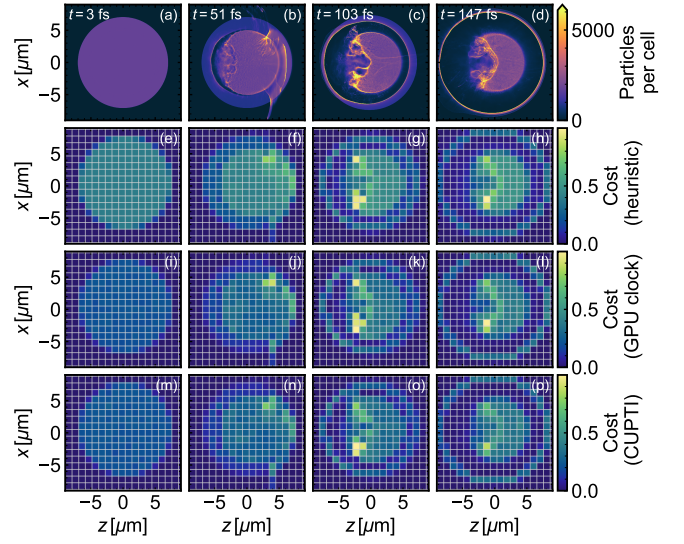

Figure 3: Time evolution of laser-ion acceleration problem (described in Sec. 3.1) for number of particles per cell (a)–(d), heuristic cost (e)–(h), GPU clock-measured cost (i)–(l), and CUPTI-measured cost (m)–(p); the computational cost assignment strategies are described in Sec. 2. The snapshots are shown at $t = \{3, 51, 103, 147\}$ fs for columns 1–4, respectively. Gray gridlines in panels (e)–(p) denote the domain decomposition into boxes (see Sec. 2.1).

3 DYNAMIC LOAD BALANCING TEST PROBLEM: 2D LASER-ION ACCELERATION

In this section, we discuss the performance of WarpX's improved dynamic load balancing in simulations of laser-ion acceleration physics. In Sec. 3.1, we describe the problem setup and initial conditions of our simulations. In Sec. 3.2, we show the time evolution of load balancing in our test problem (i.e., the time evolution of cost per computational grid, and the time dependence of load balance efficiency). Lastly, in Sec. 3.3, we present the load balancing performance dependence on several algorithm choices and numerical parameters which control aspects of the load balancing. The simulations presented in this section were run on the Oak Ridge Leadership Computing Facility (OLCF) Summit system (which consists of IBM AC922 server nodes, two IBM Power9 CPUs and six NVIDIA V100 GPUs per node).

3.1 Laser-Ion Acceleration: Problem Setup

Laser-ion acceleration was chosen to study our dynamic load balancing strategy because of its substantial spatial variations of both fields and particle distributions over the dynamic timescales. Our simulations of laser-ion acceleration are *2D3V*, that is to say the simulation plane (zx) is two-dimensional in space, yet we track all three components of the electromagnetic field and particle momenta. The problem setup we describe here is prototypical for scenarios detailed in Refs. [11, 14].

The initial conditions of our problem are shown in Fig. 3, panel (a). The panel shows a subset of the full computational domain, $-15\ \mu m \leq x \leq 15\ \mu m$ and $-10\ \mu m \leq z \leq 20\ \mu m$.

A dense target of electrons and protons initially fills a circular region of radius 5 μm (core) + 2 μm (slope) centered at $x = z = 0\ \mu$m. The initial number density of electrons and protons in the target is $n_{i0} = n_{e0} \equiv n_0 = 8.7 \times 10^{27}$ m^{-3} (5× the critical plasma density),

and cuts off exponentially near the edge of the target over a scale length $L = 50$ nm. Ions are initialized at rest, whereas electrons are initialized with Gaussian-distributed momenta; the standard deviations of the electron momenta distributions along x and z are both $0.01\ m_e c$ (m_e is the electron mass and c is the speed of light). An ultrashort laser pulse with a Gaussian profile propagates from $z = -9\ \mu$m along z toward the target starting at $t = 0$ s. Physical parameters of the laser pulse are as follows: the peak amplitude of the laser field is 10^{14} V/m ($a_0 = 25$), the laser's polarization vector is along x, the wavelength of the laser pulse is $\lambda_0 = 800$ nm, the laser beam profile *waist* (i.e., the lateral distance to the laser peak amplitude at which the field amplitude decays by a factor of e) is 4 μm, the laser pulse duration (i.e., the time required for the laser pulse peak amplitude to decay by a factor of e) is 10 fs, and the laser pulse is initialized over 30 fs and focused at the center of the target. The simulations (unless otherwise noted) are evolved up to 150 fs, which corresponds to around 4100 simulation timesteps; this timescale is long enough to cover the highly kinetic part of the simulated laser-matter interaction physics, and as a result is an appropriate timescale for testing dynamic load balancing.

In addition to the geometry and physical parameters of our setup, detailed above, we specify several numerical parameters. The simulation domain $(L_z, L_x) = (30\ \mu m, 30\ \mu m)$ (unless otherwise noted) is resolved with $(N_z, N_x) = (1920, 1920)$ cells; the cell size is $dz = dx = 0.0195\ \lambda_0 = 0.274\ c/\omega_{pe}$, where c/ω_{pe} is the electron skin depth, and $\omega_{pe} \equiv \sqrt{n_0 q_e^2/(\epsilon_0 m_e)}$ is the electron plasma frequency; q_e is the electron fundamental charge and ϵ_0 is the permittivity of free space. For parallel computation of the problem, the domain is decomposed into boxes of size $M_z = M_x = 64$ cells. To ensure numerical stability of the finite-difference PIC solver, the time resolution of our simulations is $\Delta t = 0.193\ \omega_{pe}^{-1}$, which is

less than that required by the Courant-Friedrichs-Lewy condition by a factor of 0.999. Lastly, we use $n_{\mathrm{ppc0}} = 900$ macro-particles per cell for each particle species, and third-order particle shapes. With this choice of numerical parameters, the overall time spent in compute-dominated routines is typically about half of the simulation's walltime. For our fiducial simulation parameters (detailed above), we use 16 nodes (96 GPUs). For parallel communications, we use the Message Passing Interface (IBM Spectrum MPI, version 10.3), with one MPI rank per GPU [1].

For completeness, the software, environment, WarpX input files, output data, and analysis tools used to produce and analyze the simulations presented in this paper are archived in Ref. [16].

3.2 Load Balance: Time Evolution

As the ultrahigh-intensity laser pulse interacts with the dense particle target, electrons respond quickly relative to the more massive hydrogen ions, and penetrate through the solid target. The different response times of electrons and hydrogen ions to the incident laser pulse generates strong electric fields, which can in turn accelerate ions to high energies. Throughout this process, a strong kick of the target front by the incident laser results in substantial changes to the spatial density profile as particles are transported through the target; the time evolution for the range 3–147 fs of the spatial profile of particles per cell is shown in Fig. 3 (panels (a)–(d)). The macro-particle number is intentionally kept constant in the lower-density, exponential plasma slope around the target, visible in later steps as a 'ring,' for adequate modeling of laser-absorption. Relative to the initial number of particles per cell in the target ($2 \times n_{\mathrm{ppc0}} = 1800$ particles per cell), the density can increase by a factor of 25 during the simulation (this is true at $t = 51$ fs, shown in panel (b), but note that the color bar range is truncated).

As discussed in Sec. 2.2, particle number density correlates positively with the true compute work (*cost*) associated with a box of the domain. Time variation in the spatial profile of particle number density then implies that the true computational costs, as well as the estimated approximations for each box via the heuristic, GPU clock, and CUPTI proxy schemes (discussed in Sec. 2.2), will change as the simulation progresses. The time evolution of the computational cost per box is shown in Fig. 3, rows 2–4, for the three different cost assignment schemes; heuristic (panels (e)–(h)), GPU clock (panels (i)–(l)), and CUPTI (panels (m)–(p)); gray gridlines delimit boxes which comprise the simulation domain. Costs along each row have been normalized to the maximum cost per box over the four temporal. Comparing the snapshots of costs between schemes confirms they are consistent with one another.

As discussed in Sec. 2.2, different policies are possible when determining the updated mapping from GPU ownership to boxes; we explore two commonly used policies: knapsack and Morton Z-order space-filling curve (from here on, SFC). In Fig. 4, we show visualizations of distribution mappings at $t = 103$ fs, computed according to either knapsack or space-filling curve methods (panels (a) and (b), respectively). 96 different colors (shown in the colorbar) correspond to the 96 different GPUs on which this simulation was run (for additional clarity, white GPU ID numbers are printed in the center of each box). For comparison to the underlying particle distribution, a transparent overlay of the number of particles per
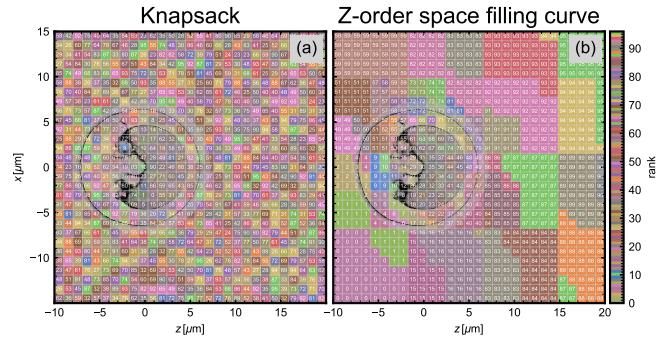

Figure 4: Comparison of distribution mappings that result from knapsack (a) and space-filling curve (b) policies (described in Sec. 2), for the laser-ion acceleration problem described in Sec. 3.1; a transparent overlay shows the number of particles per cell at physical time $t = 103$ fs (this corresponds to panel (c) of Fig. 3).

cell at $t = 103$ fs (this corresponds to panel (c) in Fig. 3) has been added to the figure.

The visualizations of distribution mappings match intuition about the knapsack (panel (a)) and SFC (panel (b)) algorithms. Knapsack calculates its distributed load by grouping boxes as efficiently as possible without any consideration of the spatial location of the boxes; as a result, GPU ownership of boxes appears to be scattered randomly. For SFC, boxes are grouped with the constraint that GPU ownership is contiguous along a Z-order curve threading the boxes; as a result, the distribution mapping computed according to the SFC algorithm shows relatively large patches placed on a single GPU in regions with relatively few particles per cell, and smaller patches in more densely packed regions. These small groupings appear roughly in the circular region of radius 7 μm centered at $(z, x) = (0 \ \mu m, 0 \ \mu m)$. Even though the distribution mappings resulting from the knapsack and SFC policies appear to be strikingly different, the load balance efficiencies (see Eq. (1)) attained at this snapshot ($t = 103$ fs) are similar; about 61% and 56% for knapsack and SFC, respectively. With the spatial constraint of the SFC algorithm, the load balance efficiency that is possible with SFC can be no greater than that obtained with knapsack; comparison between SFC and knapsack is further discussed in Sec. 3.3.

We show in Fig. 5 a comparison of the time evolution of load balance efficiency (see Eq. (1)) for a simulation with no load balancing (dot-dashed green curve), static load balancing (i.e., the load balancing routine is called once early on in the simulation; dotted red curve), and dynamic load balancing (i.e., the load balancing routine is called periodically as the simulation progresses; solid blue curve); the gray region above the line $y = 1$ is not achievable because the load balance efficiency is, by construction, a number on the interval $[0, 1]$ (see Eq. (1)). For the run with dynamic load balancing, the load balancing routine is called once every 10 timesteps (note, as described in Sec. 2.2, this does not necessarily mean that the distribution mapping is updated every 10 timesteps; a proposed distribution mapping is computed at each load balancing step, and is adopted only if it would improve the current load balance efficiency by a prescribed amount. In our fiducial simulations and the dynamic load balancing case shown in Fig. 5, the proposed distribution is adopted only if the load balance efficiency is at least
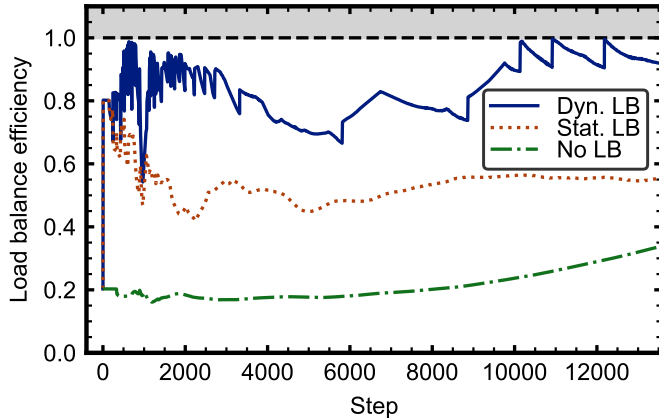
Figure 5: Time evolution of load balance efficiency for simulations with dynamic load balancing (solid blue), with static load balancing (dotted red), and without load balancing (dot-dashed green), for laser-ion acceleration problem similar to that described in Sec. 3.1 (these simulations run on 4 nodes instead of our fiducial 16 nodes, and are time-evolved up to physical time $t = 1000$ fs, as opposed to our fiducial 150 fs). The dynamic and static load balancing runs use knapsack distribution mapping.

a 10% improvement relative to the current load balance efficiency). The distribution mappings for the runs with static and dynamic load balancing are computed according to the knapsack policy.

The simulations shown in Fig. 5 are similar to the fiducial simulation described in Sec. 3.1, but the simulations run for longer physical time (up to $t = 1000$ fs) and have been weak scaled from 16 nodes (96 GPUs) to 4 nodes (24 GPUs). Correspondingly, the grid size is smaller, $(N_z, N_x) = (960, 960)$; at this resolution, the physical duration of the simulation corresponds to about 13600 timesteps.

It is important to highlight some critical results from Fig. 5. Static load balancing shows a significant improvement in overall load balance efficiency (and correspondingly, the simulation's walltime) compared to the baseline case without load balancing. The average load balance efficiency over the duration of the simulation is 53% with static load balancing and 21% without; static load balancing results in a 2.1× speedup, which is similar to the ratio of average load balance efficiencies. Dynamic load balancing can yield even greater load balance efficiency throughout the duration of the simulation; with dynamic load balancing, the average load balance efficiency is 84%, and yields a 2.9× speedup relative to the baseline without load balancing, 1.3× speedup relative to static load balancing.

Another critical result from Fig. 5 is that the dynamic load balancing routine (blue curve), which for this case is called every 10 timesteps, updates the distribution mapping only when it will result in a sufficient improvement to the current load balance efficiency (as described in Sec. 3.3, our tuned threshold is 10%). This feature can be seen clearly at, e.g., step = 10000, where the load balance efficiency increases from 91% to 100%. Early on in the simulation (step $\lesssim$ 3000), the laser-ion acceleration problem shows rapid changes in the spatial profile of particle number density (and correspondingly, the spatial profile of computational cost), but at later times in the simulation (step $\gtrsim$ 3000), the changes occur over longer timescales; these temporal changes are well captured by our dynamic routine.

By requiring that the updated distribution mapping improves the current efficiency by a threshold amount, our implementation avoids the penalty of costly communication of data among ranks (this communication time, when present, dominates the residual time spent in the load balancing routine) when doing so would not substantially improve the load balance.

3.3 Parameter Dependence of Dynamic Load Balancing Performance

The performance of typical load balancing implementation depends on several numerical parameters and choices of algorithms, as discussed in Sec. 2.2. Since algorithmic hyperparameters are hard to tune for domain scientists that model a concrete dynamic setup, we investigate multiple approaches both with respect to load balance efficiency, performance as well as minimization of the need for manual user intervention.

In Fig. 6, we present the performance dependence with respect to these parameters and choices of algorithm for our fiducial 16 node simulation; in particular, we show the performance dependence on the choice of computational cost assignment method (heuristic, GPU clock, or CUPTI), the policy used to compute the distribution mapping (knapsack or SFC), the average number of boxes per GPU (150, 38, 9, or 2; for our fiducial domain size $(N_z, N_x) = (1920, 1920)$, this is equivalent to varying the box size $M_x = 16, 32, 64,$ or 128 cells, respectively), the load balance interval (i.e., the inverse of the frequency with which we call the load balancing routine: once every 1, 3, 10, 30, 100, or 300 steps), and the load balance efficiency improvement threshold (i.e., the improvement to load balance efficiency required for a proposed distribution mapping to be communicated and updated: 5%, 10%, or 15%). For each parameter or algorithm scan, we fix those not under examination to the optimal selection from among those shown in panel (a) (for example, for each simulation represented by the first group of bars showing dependence on cost assignment method, we use knapsack, 9 boxes per GPU, load balance interval of 10 steps, and load balance efficiency improvement threshold of 10%).

The height of each bar indicates the simulation's walltime in seconds, excluding initialization time (≈ 1 s). Error bars show the spread between minimum and maximum across MPI ranks of the time spent in the most compute-intensive kernel (which includes current deposition and particle push routines); smaller error bars indicate smaller spread between minimum and maximum across MPI ranks of the time spent in current deposition and particle push, and thus correspond to a more balanced load, whereas larger error bars indicate a larger spread, and correspond to a less balanced load. Note that the hatched bars in Fig. 6 represent the same simulation; this simulation has the highest speedup relative to baseline, with the following hyperparameters: heuristic cost assignment, knapsack distribution mapping, 9 boxes per GPU, load balance interval of 10 steps, and load balance efficiency improvement threshold of 10%.

The first group of bars (blue) in panel (a) of Fig. 6 shows a comparison between the different cost assignment schemes described in Sec. 2.2, namely heuristic, GPU clock, and CUPTI. Heuristics
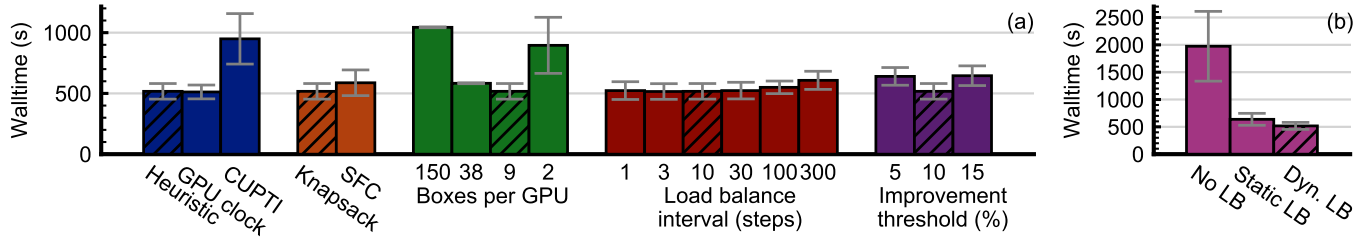
Figure 6: For the laser-ion acceleration problem described in Sec. 3.1, parameter and algorithm dependence of walltime (a), and comparison of a case with no load balancing, with static load balancing, and with dynamic load balancing (b).

need to be tuned by the application user,[1] while the latter two approaches are free of user-facing hyperparameters for costs.

As discussed in Sec. 3.2, the three schemes give similar spatial distributions of computational cost (and as a result, similar measured load balance efficiencies), yet the performance comparison here shows that with the overhead of CUPTI cost collection, the simulation runs about 2× slower than with either heuristic or GPU clock cost collection. This discrepancy is partly explained by an overhead associated with CUPTI instrumentation; we find that simply enabling collection of CUPTI activity records and registering a callback function to handle the request and delivery of buffers in which to store activity records (see Sec. 2.2) increases this simulation's walltime by 30%; the residual 70% increase in walltime (relative to the heuristic or GPU clock cases) is accounted for mostly by latency in our implementation's data movement of costs to global memory. Since the CUPTI instrumentation introduces a non-negligible, unmeasured overhead, it is possible that load balancing without knowledge of these unmeasured costs decreases the actualized (as opposed to measured) load balance efficiency.

The second group of bars (orange) in Fig. 6 (panel (a)) shows a comparison between the knapsack and SFC policies (described in Sec. 2.2; see also panels (a) and (b), respectively, of Fig. 5 for visualizations of sample distribution mappings that result from either policy). Both the knapsack and SFC simulations presented here employ heuristic cost assignment, but we have tuned the particle and cell weights (see Sec. 2.2) to give the most favorable comparison with knapsack;[2] at best, we find that the walltime with SFC is about comparable to that obtained with the knapsack algorithm. The lesser walltime obtained with knapsack, relative to that obtained with SFC, is perhaps counterintuitive; by construction, the SFC algorithm increases the correlation length with respect to GPU ownership of boxes over the simulation domain (as in panel (b) of Fig. 5, note the relatively large unicolored patches), and thus

reduces overall costs associated with communications; while the geometric constraint of the SFC algorithm may result in lower load balance efficiency relative to knapsack, the effect of improved communications with SFC may outweigh the incurred penalty to load balance efficiency, leading to a net reduction in simulation walltime. This scenario requires that communications account for a significant fraction of the simulation's walltime, which is indeed true in our case: as discussed in Sec. 3.1, compute-intensive kernels account for 50% of the measured walltime, and the remainder is predominantly accounted for by communication routines. Still, relative to knapsack, we see no speedup with SFC.

To partially account for this, we note that in AMReX's communication routines, a patch of boxes posts non-blocking MPI send operations to move data that must be communicated to a different GPU, then performs local work (such as local copy operations required by interior boxes), and lastly blocks until all MPI requests have completed. Although SFC has the potential to improve intra-GPU communication, the overall time is still bound by the inter-GPU data transfer required by boxes at patch edges. This may partially account for the comparable performance of knapsack and SFC.

The third group of bars (green) in Fig. 6 (panel (a)) shows the effect of varying the average number of boxes per GPU[3] (for a fixed domain size, this is equivalent to varying box size), which we vary over 150, 38, 9, and 2 (these correspond, respectively to box sizes $M_x = 16, 32, 64$, or 128 cells). Increasing the average number of boxes per GPU (decreasing the box size) produces a trade-off between the overhead associated with managing more boxes (for example, decreasing the box size on a fixed domain increases the total number of guard cells and communication becomes more costly), and the improved load balancing that is possible with a smaller box size (note that smaller boxes enable more fine-grained pixelization of the spatial profile of cost, and thus greater load balance efficiency relative to larger boxes); the improved load balancing can be inferred from the shrinking error bars with decreasing box size (equivalently, increasing average number of boxes per GPU). Even though the load balance efficiency is greater with an average of, e.g., 150 boxes (of size $M_x = 16$) than with 2 boxes (of size $M_x = 128$ cells) per GPU, the walltime is greater (896 s with 150 boxes per GPU as opposed to 1040 s with 2 boxes per GPU). This is because the overhead associated with a greater number of boxes (of smaller size) outweighs the performance benefit of improved load balance efficiency. We find that an average of 9 boxes per GPU (box

---

[1] On the OLCF Summit system using finite-difference time-domain field solve and third-order particle shapes, we measured in WarpX particle and cell weights of 0.75 and 0.25, which (unless otherwise stated) we used in the tests presented here. These weights were calibrated based on benchmark tests, one with a relatively large number of particles per cell (27 particles per cell) distributed uniformly over the simulation domain, and another with no particles (only cells); together, these tests yielded estimates of the walltime corresponding to a single particle and cell, and in turn relative weightings for a particle and cell.

[2] By inspection, we determined (for heuristic cost assignment) that particle and cell weights of 0.02 and 0.98, respectively, yield roughly optimal performance with the SFC algorithm; we note that the AMReX implementation of the knapsack algorithm includes the option to limit the maximum number of boxes per GPU (our default is 1.5 times the average number of boxes per GPU), but this option is not implemented for SFC. The relatively large cell weight and small particle weight that we find are optimal with SFC emulates the constraint on the maximum number of boxes per GPU.

[3] Note that the number of boxes per GPU, as opposed to the *average* number of boxes per GPU, is not fixed; this can of course vary as GPU ownership of boxes is shuffled due to load balancing.

size $M_x = 64$ cells) produces a close to optimal trade-off between overhead and improved load balancing capability.

The fourth group of bars (red) in panel (a) of Fig. 6 shows the performance dependence on load balance interval, which we vary over 1, 3, 10, 30, 100, and 300 steps (load balance interval of 10, for example, means the load balancing routine is called once every 10 steps). As discussed in Sec. 3.2, there is little penalty to walltime when calling the load balancing routine frequently because the costly operation of communicating and updating the distribution mapping on all ranks is done only when doing so would improve the load balance efficiency more than a minimal threshold.

This improvement threshold is shown in the last group of bars (purple) in panel (a) of Fig. 6, which we vary over 5%, 10%, and 15%. If the improvement threshold is too low, communications become more costly as the distribution mapping is updated more frequently; on the other hand, if the improvement threshold is too high, the distribution mapping is not updated frequently enough to reap the performance benefit of load balancing. We find that an improvement threshold of 10% offers an optimal trade-off between these competing effects.

For the simulations we present here, the time required to gather costs from all ranks (which is needed to determine whether to update the distribution mapping) is, at most, no greater than 2.3% of the total walltime (which is achieved with a load balance interval of 1). Since this is a relatively small fraction of the total walltime, the load balancing routine may be called frequently to ensure that load balance is maintained, without incurring a significant penalty to walltime. Empirically, we find little difference in walltime for load balance intervals in the range 1–30 steps, with an increasing trend for load balance intervals $\gtrsim 30$ steps (this is due to the lower load balance efficiency on average over the duration of the simulation).

To summarize, we achieve best performance with the following selection of parameters and algorithms: GPU clock cost assignment, knapsack policy for computing the distribution mapping, an average of 9 boxes per GPU (equivalently, a box size $M_x = 64$ cells), and calling the load balancing routine once every 10 steps (similar performance is achieved with heuristic in place of GPU clock cost measurement, however this requires the user to tune particle and cell weights, which can vary depending on hardware an algorithmic choices). With these selections, load balancing accounts for (at most, across all ranks) 4–6% of the walltime in our test problem.

In panel (b) of Fig. 6, we compare the walltimes of a baseline case without load balancing, a case with static load balancing (i.e., the load is balanced once toward the beginning of the simulation), and a case with dynamic load balancing where the parameters and algorithms are tuned for roughly optimal performance (as determined from the tests shown in panel (a); the static load balancing case has the same parameter and algorithm selections as the dynamic load balancing case, apart from the load balance interval). The relatively large error bars for the baseline case without load balancing indicate a severe load imbalance. Static load balancing results in a 2.9× speedup relative to the baseline case without load balancing; dynamic load balancing yields even greater performance improvement, with a 3.8× speedup relative to the baseline case without load balancing (1.2× speedup compared to static load balancing). This shows that the spatial profile of costs varies substantially with time,

and that dynamic, as opposed to static, load balancing is essential to improved performance.

4 PERFORMANCE ASSESSMENT AND SCALING OF LOAD BALANCING

In this section, we present a performance model, calibrated with strong scaling measurements of our code, and use it to assess the performance of our improvements to WarpX's load balancing in the laser-ion acceleration problem described in Sec. 3.1. We present the weak scaling of our load balancing routine from 1 up to 1024 nodes (6 – 6144 GPUs). As in Sec. 3, these simulations were run on the OLCF Summit system (IBM AC922 server nodes; 2 IBM Power9 CPUs and 6 NVIDIA V100 GPUs per node).

For a given code, how much of a performance improvement may be anticipated with load balancing? To answer this question, we consider the load balancing operation in the context of strong scaling, i.e. increasing compute resources for a problem of fixed size. With measurements of strong scaling for a given code, one may model the performance response as a function of available compute resources; ideally, performance improves linearly with increasing compute resources, and with a constant of proportionality equal to unity. However, in realistic codes, factors such as communications and overhead may modify the relationship and lead to less than ideal scaling. A strong scaling model can capture these nonideal effects and be used to predict the performance improvement that would result from a given increase in compute resources.

To apply strong scaling in understanding the performance improvement that is possible through load balancing, we consider a simulation with a computational load that is initially imbalanced (i.e., there is at least one compute element assigned a greater computational load than another); we call this initial maximum compute work (over all compute elements) $c_{\text{max}0}$. The performance (e.g., walltime) of this hypothetical simulation is limited by the initial load imbalance $c_{\text{max}0}$, and may be improved by distributing compute work evenly over available resources. After perfect load balancing, the compute work is equally distributed over all N compute elements, $c_0 = c_1 = \cdots = c_N = c_{\text{avg}0}$, which becomes the new performance limiter (we define $c_{\text{avg}0}$ as the average at initialization of $c_0, c_1, \cdots, c_N$). Load balancing is similar to strong scaling: more compute resources are assigned to the work $c_{\text{max}0}$, and the workload of the compute element initially assigned that work decreases from $c_{\text{max}0}$ to $c_{\text{avg}0}$; with strong scaling, increasing the compute resources while holding the problem size fixed effectively decreases the compute work assigned to each compute element.

The ratio $c_{\text{max}0}/c_{\text{avg}0}$ can therefore be calibrated (weighted) against a code's characteristic strong-scaling efficiency, which we fit to an exponential model for walltime: $t_{\text{wall}} \approx n_{\text{nodes}}^{-x}$. The parameter x is a value between 1 (ideal) and 0. The maximum speedup, S, from perfect load balancing can then be expressed as:

$$S = \left(\frac{c_{\text{max}0}}{c_{\text{avg}0}} \right)^x = \left(\frac{1}{E_0} \right)^x, \tag{2}$$

where E_0 is the initial load balance efficiency of the simulation (see Eq. (1)). For example, for 16 nodes, we measure load a imbalance ratio to average cost of $c_{\text{max}0}/c_{\text{avg}0} = 6.2$. Perfectly load-balanced on
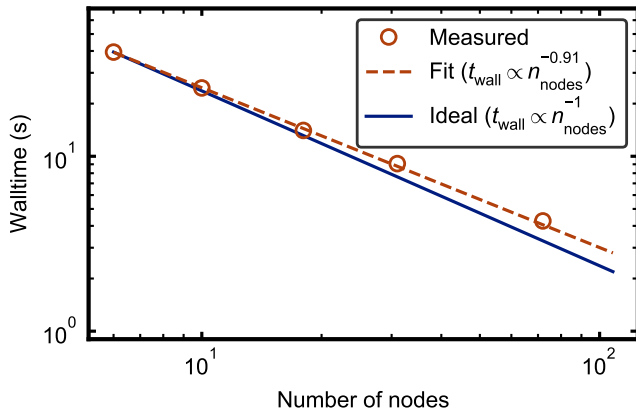
Figure 7: Strong scaling for a WarpX baseline problem (domain filled uniformly with plasma). Circles show measurements and the solid curve shows ideal scaling of walltime with number of compute nodes. The dashed curve shows a fit to the measured walltimes, which we use, along with the level of initial load imbalance, to predict the maximum possible speedup possible with perfect load balancing.
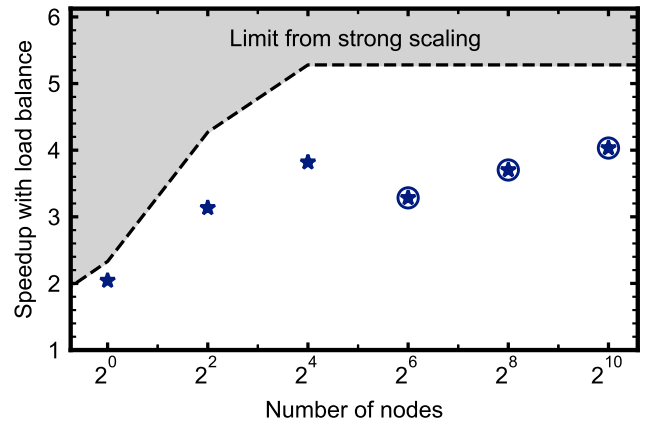

Figure 8: For the laser-ion acceleration test problem described in Sec. 3.1, weak scaling (from 1 up to 1024 nodes, i.e., 6 up to 6144 GPUs) of speedup factor, relative to baseline case without load balancing. The dashed line shows maximum possible speedup, as predicted from strong scaling and the initial level of load imbalance. Circled points indicate cases in which the no-load balancing baseline exceeded GPU memory prior to completion of the simulation (43%, 9%, and 8% completion for 64, 256, and 1024 nodes, respectively); for these cases, the simulations with dynamic load balancing ran to completion without exceeding GPU memory capacity.

the same amount of compute resources, this is the factor the slowest compute element is strong-scaled. Calibrated against WarpX characteristic strong scaling, in 2D3V $x = 0.91$ and 3D3V $x = 0.88$, the best speedup a load balance algorithm can achieve with this starting condition and assuming a dynamically sustained imbalance is 5×. In practice, realization of this maximum speedup may be limited by further factors, such as the cost of performing a load balance operation and more complex communication patterns.[4]

In Fig. 7, we show strong scaling measurements (red circles) for WarpX, i.e., walltime as a function of compute nodes; the problem setup is a domain of size $(N_z, N_x) = (3072, 3072)$ cells, filled uniformly with 550 particles per cell. In the tests, we vary the number of nodes over 6, 10, 18, 31, and 72 (36, 60, 108, 186, and 432 GPUs, respectively). The dashed line shows a log-log fit to our measurements, yielding a performance model for walltime, $t_{wall} \propto n_{nodes}^{-0.91}$ (for comparison with the ideal strong scaling, we show also the curve $t_{wall} \propto n_{nodes}^{-1}$ as a solid line).

To assess the performance of WarpX's updated load balancing, we performed a weak scaling test (from 1 up to 1024 nodes, i.e., 6 up to 6144 GPUs) of the laser-ion acceleration problem with and without load balancing (described in Sec. 3.1), and compared the measured speedup (i.e., the ratio of walltime with load balancing to walltime without load balancing; blue points in Fig. 8) to the ideal limit computed from strong scaling and the initial load imbalance (dashed line in Fig. 8). For 64, 256, and 1024 nodes, our simulations without load balancing exceeded a GPU's memory (16 GB on the NVIDIA V100 GPUs we use) before reaching the prescribed final timestep, leading to extremely degraded performance (this is indicated by the circled points in Fig. 8);

for these cases the speedup is computed only with walltimes (of the load balanced and load imbalanced cases) up to the timestep that

the load imbalanced case exceeds a GPU's memory.[5] For 64, 256, and 1024 nodes, the simulations without load balancing reached 43%, 9%, and 8% completion, respectively, before exceeding a GPU's memory; this demonstrates that, especially when a code uses memory-limited GPUs, load balancing is not just crucial to performance: it enables simulations that are not possible otherwise for a given amount of compute resources! With static load balancing, we tested the cases with 64 and 256 nodes and they ran to completion without exceeding GPU memory capacity, but for problems which develop severe load imbalance as the simulation progresses, dynamic load balancing may be necessary to prevent the performance-breaking effect of exceeding GPU memory capacity.

For number of nodes in the range 4–1024, we see, relative to the baseline case without load balancing, a 3–4× speedup with dynamic load balancing (which is likely an underestimate for the crashing runs with 64, 256, and 1024 nodes); compared to the ideal maximum speedup computed from strong scaling, these measurements attain from 62% to 74% of the predicted limit (when running on 6 GPUs, we observe a 2× speedup, which corresponds to 88% of the predicted maximum speedup).

This study has demonstrated that adaptive, run time based GPU timers yield a substantial performance gain for WarpX PIC simulations with temporally variable particle distributions. GPU-based simulations of next generation particle accelerators and astrophysical plasmas can now readily take advantage of these features to ensure an efficient and performant transition to anticipated exascale supercomputers.

---

[4]In our case, decreasing the box size M_x, as discussed in Sec. 3.3, leads to an overall performance penalty, in spite of the improved load balancing with smaller box size (see the third group of bars (green) in panel (a) of Fig. 6).

[5]We also ran the 64 and 256 node cases (both with and without load balancing) with fewer particles per cell, such that the cases without load balance ran to completion without exceeding GPU memory; the measured speedup factors are similar to those shown by the circled points in Fig. 8, which indicates that those measurements, though limited to the period before exceeding GPU memory, are representative of the speedup possible over the full prescribed time range.

5 RELATED WORK

State-of-the-art particle-in-cell codes [4, 5, 7, 9, 13, 17] typically support either static load balancing or dynamic load balancing targeting CPUs. To the knowledge of the authors, those particle-mesh codes that do support dynamic load balancing on GPUs rely on cost functions based on node-local application data as a surrogate for computing run time [10, 19]. The heuristic approach, however, is not necessarily an accurate reflection of true computing run time [22]. To gain the full performance benefit of dynamic load balancing, without excessive tuning of hyperparameters by the user, computational costs can be measured accurately at runtime. An adoption of dynamic load balancing strategies for accelerator hardware in exascale-workflows is therefore needed, including a reduction of user-facing hyperparameters, which are hard and expensive to tune in realistic scenarios.

Dietrich et al. [8] demonstrated instrumentation of CUDA kernels as a way to measure kernel run times for post-run analysis. One of the herein presented implementations for cost assessment which is free of user-facing hyperparameters, GPU clock, employs a similar approach, yet provides measurement of costs in-situ as the application is running, thus enabling on-the-fly load balancing based on measured computational costs (see Sec. 2.2). Furthermore, the runtime overhead introduced with this technique is small enough that it does not negate the performance benefit of automatic runtime load balancing [20]. The GPU clock strategy for cost measurement is potentially portable, given the accelerator framework supports a clock method (or similar) to assess kernel run times.

6 SUMMARY AND DISCUSSION

In this work, we present enhancements to dynamic load balancing for particle and mesh-based simulations targeting GPU architectures. As a component of these improvements, we introduce several GPU-applicable strategies for measuring the relative computational costs of sub-domains of compute work. While as application developers we can find optimal hyperparameters for a specific setup for heuristic cost-functions, we demonstrate that a measurement of the actual kernel run time can be established. Especially, we implemented a potentially vendor-neutral, in-situ, in-kernel measurement of run time based on a GPU clock that shows negligible load-balancing overhead in practice. Contrarily, a measurement approach based on Nvidia CUPTI added significant run time overhead and is not vendor-neutral.

We demonstrate our methods in the fully kinetic particle-in-cell code WarpX and explore its performance. For the scientifically relevant test case of laser-ion acceleration, we explore the performance dependence of load balancing on several numerical parameters and algorithm choices that enter into our routine, including cost assignment strategy (heuristic cost assessment based on a weighted linear sum of the number of particles and cells, GPU clock timing to assess summed thread execution time, and CUPTI-based timing to assess kernel execution time using NVIDIA's CUPTI API), load balance strategy (knapsack or SFC), the number of boxes per GPU (equivalently, on a fixed domain, the box size in cells), the frequency with which we call the load balancing routine, and the load balance efficiency improvement required to communicate and update a proposed distribution mapping. For the laser-ion acceleration that is

the focus of the present work, we measure a 3.8× speedup relative to the baseline without dynamic load balancing and a 1.2× speedup compared to the static load balancing baseline.

To assess the performance of WarpX's updated dynamic load balancing, we introduce a performance model based on strong scaling measurements of WarpX and link performance improvement through strong scaling to the initial level of load imbalance in our simulations, thereby predicting a theoretical maximum speedup factor that is achievable through load balancing. We present the weak scaling (from 24 up to 6144 GPUs on Summit) of our dynamic load balancing performance relative to the baseline case without load balancing, and find that dynamic load balancing improves performance, with achieved improvement typically 62% to 74% of the predicted maximum (88% when running on 6 GPUs); in particular, several of our simulations demonstrate that dynamic load balancing is de-facto a prerequisite for productive usage of distributed, locally limited GPU memory at scale.

In the present work, we focused on load balancing according to the cost of GPU compute. Incorporating communication costs into our load balancing routine will be a topic of future investigation, and may bring the performance of WarpX's load balancing closer to the theoretical limit of our presented performance model. Further exploration of communication costs may also help to demystify the relative performance of knapsack and SFC update for the distribution mapping, which remains an open question.

REFERENCES

[1] 1993. *Document for a standard message-passing interface.* Technical Report CS-93-214. Message Passing Interface Forum.
[2] Michael Bader. 2012. *Space-Filling Curves: An Introduction with Applications in Scientific Computing.* Springer Publishing Company, Incorporated.
[3] C K Birdsall and A B Langdon. 1991. *Plasma Physics via Computer Simulation.* IOP Publishing. https://doi.org/10.1201/9781315275048
[4] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas* 15, 5 (2008), 055703. https://doi.org/10.1063/1.2840133 arXiv:https://doi.org/10.1063/1.2840133

[5] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. 2013. Radiative signature of the relativistic Kelvin-Helmholtz Instability. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2504564

[6] NVIDIA Corporation. [n. d.]. API Reference Guide for CUPTI. https://docs.nvidia.com/cuda/cupti/index.html.

[7] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov, N. Aunai, J. Dargent, C. Riconda, and M. Grech. 2018. Smilei : A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications* 222 (2018), 351 – 373. https://doi.org/10.1016/j.cpc.2017.09.024

[8] R. Dietrich, F. Schmitt, R. Widera, and M. Bussmann. 2012. Phase-Based Profiling in GPGPU Kernels. In *2012 41st International Conference on Parallel Processing Workshops*. 414–423. https://doi.org/10.1109/ICPPW.2012.59

[9] 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, and J. C. Adam. 2002. OSIRIS: A Three-Dimensional, Fully Relativistic Particle in Cell Code for Modeling Plasma Based Accelerators. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–351.

[10] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee. 2016. The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing. *J. Comput. Phys.* 318 (2016). Issue 305. https://doi.org/10.1016/j.jcp.2016.05.013

[11] Axel Huebl, Martin Rehwald, Lieselotte Obst-Huebl, Tim Ziegler, Marco Garten, René Widera, Karl Zeil, Thomas E. Cowan, Michael Bussmann, Ulrich Schramm, and Thomas Kluge. 2020. Spectral control via multi-species effects in PW-class laser-ion acceleration. *Plasma Physics and Controlled Fusion* 62, 12, Article 124003 (Dec. 2020), 124003 pages. https://doi.org/10.1088/1361-6587/abbe33 arXiv:physics.plasm-ph/1903.06428

[12] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. 2011. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *2011 International Conference on Parallel Processing*. 176–185. https://doi.org/10.1109/ICPP.2011.71

[13] Kyle G. Miller, Roman P. Lee, Adam Tableman, Anton Helm, Ricardo A. Fonseca, Viktor K. Decyk, and Warren B. Mori. 2021. Dynamic load balancing with enhanced shared-memory parallelism for particle-in-cell codes. *Computer Physics Communications* 259 (2021), 107633. https://doi.org/10.1016/j.cpc.2020.107633

[14] Lieselotte Obst, Sebastian Göde, Martin Rehwald, Florian-Emanuel Brack, João Branco, Stefan Bock, Michael Bussmann, Thomas E. Cowan, Chandra B. Curry, Frederico Fiuza, Maxence Gauthier, René Gebhardt, Uwe Helbig, Axel Huebl, Uwe Hübner, Arie Irman, Lev Kazak, Jongjin B. Kim, Thomas Kluge, Stephan Kraft, Markus Loeser, Josefine Metzkes, Rohini Mishra, Christian Rödel, Hans-Peter Schlenvoigt, Mathias Siebold, Josef Tiggesbäumker, Steffen Wolter, Tim Ziegler, Ulrich Schramm, Siegfried H. Glenzer, and Karl Zeil. 2017. Efficient laser-driven proton acceleration from cylindrical and planar cryogenic hydrogen jets. *Scientific Reports* 7, Article 10248 (Aug. 2017), 10248 pages. https://doi.org/10.1038/s41598-017-10589-3

[15] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Tom Arsenlis, and Nancy M. Amato. 2014. Load Balancing N-Body Simulations with Highly Non-Uniform Density. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. Association for Computing Machinery, New York, NY, USA, 113–122. https://doi.org/10.1145/2597652.2597659

[16] M. Rowan, A. Huebl, K. Gott, J. Deslippe, M. Thévenet, R. Lehe, and J.-L. Vay. 2021. Supplementary material for "In-Situ Assessment of Device-Side Compute Work for Dynamic Load Balancing in a GPU-Accelerated PIC Code". https://doi.org/10.5281/zenodo.4708449

[17] A. Spitkovsky. 2005. *AIP Conference Proceedings* 801 (2005), 345.

[18] TOP500. 2020. November 2020 | TOP500 Supercomputer Sites. Retrieved December 9, 2020 from www.top500.org

[19] S. Tsuzuki and T. Aoki. 2016. Effective Dynamic Load Balance using Space-Filling Curves for Large-Scale SPH Simulations on GPU-rich Supercomputers. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 1–8. https://doi.org/10.1109/ScalA.2016.005

[20] Rob F. Van der Wijngaart, Evangelos Georganas, Timothy G. Mattson, and Andrew Wissink. 2017. A New Parallel Research Kernel to Expand Research on Dynamic Load-Balancing Capabilities. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 256–274.

[21] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D.P. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thévenet, and W. Zhang. 2018. Warp-X: A new exascale computing platform for beam–plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 909 (2018), 476 – 479. https://doi.org/10.1016/j.nima.2018.01.035 3rd European Advanced Accelerator Concepts workshop (EAAC2017).

[22] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. 2001. Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*. Association for Computing Machinery, New York, NY, USA, 6. https://doi.org/10.1145/582034.582040

[23] Mo Zeyao and Zhang Baolin. 2001. Multilevel averaging weight method for dynamic load imbalance problems. *International Journal of Computer Mathematics* 76, 4 (2001), 463–477. https://doi.org/10.1080/00207160108805040 arXiv:https://doi.org/10.1080/00207160108805040

[24] Weiqun Zhang, Andrew Myers, Kevin Gott, Ann Almgren, and John Bell. 2020. AMReX: Block-Structured Adaptive Mesh Refinement for Multiphysics Applications. *arXiv e-prints*, Article arXiv:2009.12009 (Sept. 2020), arXiv:2009.12009 pages. arXiv:cs.MS/2009.12009