SAND2022-6391C

Optimizing Distributed Load Balancing for Workloads with Time-Varying Imbalance

1st Jonathan Lifflander
Sandia National Laboratories
Livermore, CA, U.S.A.
jliffla@sandia.gov

2nd Nicole Lemaster Slattengren
Sandia National Laboratories
Livermore, CA, U.S.A.
nlslatt@sandia.gov

3rd Philippe P. Pébaÿ
NexGen Analytics
Sheridan, WY, U.S.A.
philippe.pebay@ng-analytics.com

4th Phil Miller
Intense Computing
New York, NY, U.S.A.
phil.miller@intensecomputing.com

5th Francesco Rizzi
NexGen Analytics
Sheridan, WY, U.S.A.
francesco.rizzi@ng-analytics.com

6th Matthew T. Bettencourt
NexGen Analytics
Sheridan, WY, U.S.A.
matt.bettencourt@ng-analytics.com

***Abstract*—This paper explores dynamic load balancing algorithms used by asynchronous many-task (AMT), or 'task-based', programming models to optimize task placement for scientific applications with dynamic workload imbalances. AMT programming models use overdecomposition of the computational domain instead of graph partitioning methods. Overdecompostion provides a natural mechanism for domain developers to expose concurrency and break their computational domain into pieces that can be remapped to different hardware. This paper explores fully distributed load balancing strategies that have shown great promise for exascale-level computing but are challenging to theoretically reason about and implement effectively. We present a novel theoretical analysis of a gossip-based load balancing protocol and use it to build an efficient implementation with fast convergence rates and high load balancing quality. We demonstrate our algorithm in a next-generation plasma physics application (EMPIRE) that induces time-varying workload imbalance due to spatial non-uniformity in particle density across the domain. Our highly scalable, novel load balancing algorithm, achieves over a 3x speedup (particle work) compared to a bulk-synchronous MPI implementation without load balancing.**

***Index Terms*—dynamic load balancing, overdecomposition, exascale computing, asynchronous many-task (AMT), task-based programming, distributed algorithms**

I. Introduction

As the exascale era emerges, achieving performance-portable scalability is paramount for developing next-generation scientific applications. Novel heterogeneous/hybrid architectures require adaptable and composable programming models to optimally utilize the diversity of hardware resources.

Many scientific application domains engender workload imbalances due to spatial, structural, or temporal non-uniformities in their underlying computational structure. These include adaptive mesh refinement, smooth-particle hydrodynamics, molecular dynamics, and fast multipole methods. Remediating these imbalances directly in the Message Passing Interface (MPI), using a bulk-synchronous programming model, is challenging as it requires reconfiguration of the domain's decomposition onto potentially non-uniform (e.g., NUMA or heterogeneous) hardware resources. Furthermore, after changing the domain mapping onto hardware, an application must then restructure its communication patterns: e.g., rebuilding neighborhood ghosting layers or exchanging messages to coordinate sparse data exchanges.

A potential solution is to use graph partitioning (e.g., Zoltan [1]) to repartition the domain. However, this does not alleviate the complexity of computation/communication pattern reconfiguration in MPI after the partitioner changes the decomposition. Moreover, such schemes cannot be applied in a highly incremental fashion, thereby limiting their feasibility for rapidly time-varying workloads.

In contrast to graph partitioning decomposition strategies, asynchronous many-task (AMT), otherwise known as task-based, programming models use *overdecomposition*. For applications where interactions across the spatial or other domain remain local, decomposing that domain using a factor much greater than the number of computational processes yields domain chunks with dependencies on only a few neighboring chunks, exposing a great deal of asynchrony that was previously hidden. These chunks can be distributed across processors in an arbitrary way because the communication needed to satisfy their dependencies is not expressed in terms of MPI rank. Thus, frequent, incremental rebalancing of time-varying workloads can be performed by the underlying system without the application's detailed involvement. Due to its asynchrony and explicitly-exposed dependencies, AMT engenders a more composable programming model that is less machine-specific and more adaptable to novel, heterogeneous architectures. Further, AMT makes it easier for application developers to decompose their computational domain into multi-dimensional structures that enable more natural reasoning and expression of

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. cf. SAND2021-6350 C

communication patterns than decomposing the domain directly to MPI ranks.

Overdecomposition empowers the runtime system to instrument tasks by measuring computational intensity and recording communication patterns. Subsequently, with these measurements, the runtime system can optimize task placement based on an objective function. For example, the objective of a strategy might be to minimize total running time or inter-node communication. The latter could be achieved by the load balancer instructing the runtime to co-locate tasks that communicate heavily. This paper proposes a novel distributed load balancing algorithm with near-optimal scalability that operates on collected instrumentation to minimize the imbalance across ranks in time spent executing tasks.

This paper is organized as follows: § II discusses existing load balancing strategies; § III presents the novel tasking runtime used for the results we present; § IV describes the Grapevine approach that we extend in this work; § V describes our proposed approach and provides its mathematical foundations; § VI shows results in the context of EMPIRE, a plasma physics application; § VII concludes this article.

II. Related work

Load balancing has been extensively studied in the literature, spanning a wide variety of algorithms. For static, predictable problems, inspector-executor approaches [2] may be applied to carefully map work to processors during an initialization phase. For these problems, and potentially more dynamic ones, graph partitioning algorithms along with their associated parallelizations [3] and sparse graph representation [4] have been a rich area of research resulting in a set of packages: Zoltan [1], Scotch [5], ParMetis [6].

Incremental strategies with better efficacy on dynamic problems, include a suite of load balancers that Charm++ [7] ships spanning centralized, hierarchical [8], and distributed schemes. This paper builds on a gossip-based, distributed computing protocol [9], implemented in Charm++, that was inspired from epidemic algorithms [10]. Gossip-based protocols have been used in distributed computing [11] for highly effective dissemination of information.

Several frameworks and applications use partitions of tasks to blocked mappings or space-filling curves to assign work to processes. These kinds of frameworks include tree-structured AMR [12], [13], multi-block CFD [14], tree-structured N-body codes [15], and dense linear algebra [16]–[19]. These approaches have the advantage of implicitly maintaining communication locality, with the disadvantage that the ordering tightly constrains the possible assignments of objects to processes, hence hindering the load balancing process.

Work stealing is a popular load balancing/scheduling technique with theoretically proven efficiency in shared-memory [20]. It has the advantage of addressing highly dynamic imbalances that arise during a phase, rather than only addressing imbalances across phases, benefiting classes of applications with highly unpredictable loads. Distributed work stealing has shown to be viable [21], including a novel *retentive* algorithm [22], in which the location where work is actually executed becomes the starting point for that work on the next iteration. This can be effective in comparison to persistence-based load balancers in a distributed-memory context on large systems.

III. Programming & Execution Model

The load balancing algorithms described in this paper are developed in the context of the Virtual Transport (vt) tasking library[1], an AMT programming and execution model built to interoperate with MPI (refer to [23] for more detail). This library, developed by the DARMA project at Sandia National Laboratories, provides a transitional runtime for overdecomposing applications, making them more asynchronous and providing dynamic load balancing to improve performance on large distributed-memory systems. Many production scientific applications are built atop large MPI libraries, e.g. Trilinos [24] that provides core capabilities such as distributed data structures, linear and nonlinear solvers, and meshing. These libraries embed many MPI calls to perform communication across nodes. Thus, any proposed AMT runtime must be fully composable and interwoven with MPI.

A. Execution Model

At a low level, control flow and data flow are achieved in vt by either (1) sending active messages that trigger registered handlers on a target rank/registered object or (2) directly transferring data by targeting RDMA handles with get, put, and accumulate operations. Instead of users explicitly calling an API to receive a message, the scheduler drives progress by searching for incoming messages and then reading their content to determine what to execute when they arrive. vt employs distributed termination detection algorithms to sequence tasks and create dependencies for ordering distributed execution.

B. The Principle of Persistence

To enable instrumentation for load balancing, vt provides a mechanism to demarcate an application *phase*, e.g. a timestep or iteration within an application. Load balancing depends on the general notion that computation in previous phases in an application can be used as a predictor of the computation in future phases. In the literature, this has been referred to as the *principle of persistence* [9]. The efficacy of our load balancing algorithms presented herein relies on it, so it must hold to some extent at the phase level for at least part of the time in order to improve performance. We found this to be broadly true for the scientific applications we are studying; for those where it would not, load balancing could be applied within a given phase instead of between two.

C. Measuring Imbalance

In this work, we use the same metric as Menon, et. al [9] for imbalance:

$$\mathcal{I} = \frac{\ell_{\max}}{\ell_{\text{ave}}} - 1 \tag{1}$$

[1]Available BSD-licensed at: `https://github.com/DARMA-tasking/vt`

where $\ell_{\max}$ and ℓ_{ave} respectively refer to the maximum per-rank task load and average per-rank task load over all the ranks. With this definition, a perfect load distribution occurs when $\mathcal{I} = 0$. The key reasoning behind this metric is that the performance attained is always limited by the maximum rank load in a phase due to synchronization at the end of each phase. Thus, throughout this paper we will use this metric for analyzing the quality of load distributions.

IV. Load Balancing Scalability

A. Background

When executing a scientific application that has workload imbalances, the scalability of the application may be limited by the scalability of the load balancer itself. The total cost of performing load balancing is relative to: (a) the timescale of application workload variation; (b) the scalability of the load balancer; (c) and, the number of tasks per node (level of overdecomposition), which is typically a factor in the computational complexity of the load balancing algorithm. There is also a tradeoff between these: the more scalable the load balancer, the more frequently it can be invoked as workloads dynamically vary over time.

Initial research of load balancing algorithms studied centralized schemes that are suitable for low levels of concurrency. Centralized strategies have the advantage of having global knowledge of the input data (knowledge of task durations/communications) on the rank where the redistribution algorithm executes. Thus, these strategies can provide very good distributions by applying greedy algorithms or other purely local heuristics. However, as the amount of concurrency increases, these balancers quickly become an execution time and memory bottleneck.

To ameliorate scalability limits, hierarchical schemes often build trees or form groups of ranks that recursively apply redistribution algorithms on subsets of the ranks and trade tasks at higher levels in the hierarchy. These algorithms nominally are lower bounded by $\Omega(\log(P))$, where P is the number of ranks. However, as scales increase even more (where P is very large), the cost of running even a hierarchical load balancer might limit its efficacy when workload imbalances evolve rapidly.

In contrast, fully distributed load balancers avoid tree-like structures, instead relying on distributed computing protocols (neighborhoods, gossip, randomization) to discover tasks in the system and trade them across ranks independently of other uninvolved ranks. In terms of scalability, distributed algorithms show the most promise for the exascale era, but are difficult to implement and research has shown they generally have limited efficacy due to a lack of information, leading to poor results.

Research by Menon, et al. [9] has shown one of the first fully distributed load balancing algorithms that is more scalable than hierarchical strategies yet achieves nearly comparable (or better) quality in load distribution. In this paper, we build on this approach, evince its limitations by means of a theoretical analysis, and improve it noticeably both in terms of quality of the load distribution produced.

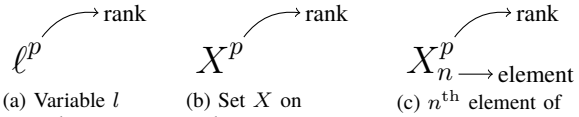
B. The GrapevineLB Algorithm

With the initially described design and implementation, Menon, et al. demonstrate the efficacy of the original algorithm, referred to as GrapevineLB, for adaptive mesh refinement and molecular dynamics on up to 131,072 cores of a BlueGene/Q system. The algorithm is fully distributed, i.e. with neither centralized nor hierarchical synchronization, using partial information about the global state of the system in order to perform the load balancing. It consists of two stages: global information propagation using a lightweight algorithm inspired by epidemic algorithms (the *gossip* phase), and work unit transfer using a randomized algorithm (the *transfer* phase).

To start, ranks perform an all-reduce to collect constant-size statistical data about their load (maximum load, $\ell_{\max}$; average load, ℓ_{ave}) to determine which ranks are overloaded or underloaded. Then, in the *gossip* phase, underloaded ranks randomly send messages to other ranks in a series of k rounds with a branching factor f. As each new message is received, known underloaded ranks are added to a local list that accumulates this knowledge, which is then forwarded when the next round of messages are sent. Theoretical analysis in the paper demonstrates that $\log_f P$ rounds produce global knowledge transfer with high probability during the gossip phase[2]. In the asynchronous implementation, rounds are not synchronized and proceed without barriers, relying on distributed *termination detection* to detect when all causally related gossip messages have been received and processed. Due to the asynchronous implementation, global knowledge transfer is not attained, but we achieve good results without it.

In the subsequent *transfer* phase, with overloaded ranks knowing about a subset of underloaded ranks, the potential re-location of tasks is determined using a probability mass function. The cumulative mass function (CMF) is calculated based on the load availability of each rank compared to the mean load. Because an overloaded rank can transfer tasks without distributed coordination with the underloaded rank, and an underloaded rank may be known by multiple overloaded ranks, the mass function is calculated based on local yet imprecise knowledge of the load of the underloaded rank.

Algorithm 1 specifies the original algorithm for the information propagation stage. The Transfer function in Algorithm 2 describes the transfer stage, where tasks are chosen for migration based on the partial information gathered. Superscripts notate the rank for a given symbol. Capital letters (e.g., X) are sets (unless indexed with a subscript), while variables/constants and in lowercase letters.



(a) Variable l on rank p (b) Set X on rank p (c) n^{th} element of set X on rank p

[2]This may not be desirable, as it may result in lists of size $\mathcal{O}(P)$ being communicated and stored in memory. In future work, we intend to examine load balancing efficacy with more limited information to avoid this potential scalability pitfall. Note that research on random graphs shows that they will tend toward full connectivity even with relatively modest maximum degrees.

Notations Definitions of symbols used in algorithms.

S^p	▷ Set of underloaded ranks known by rank p
$\textsc{Load}^p()$	▷ Map of loads in S^p known by rank p
T^p	▷ Set of tasks currently on rank p
M^p	▷ Set of tasks to migrate off rank p
$\textsc{Target}^p()$	▷ Map of target ranks for tasks in M^p
P	▷ Set of all ranks
f	▷ Gossip fanout factor (constant value)
k	▷ Number of rounds (constant value)
h	▷ Relative imbalance threshold (constant value)

Algorithm 1 The inform/gossip stage to gather knowledge about underloaded rank loads.

1: $S^p \leftarrow \emptyset$
2: $\textsc{Load}^p() \leftarrow \emptyset$
3: $M^p \leftarrow \emptyset$
4: $\textsc{Target}^p() \leftarrow \emptyset$
Require: $|S^p| \equiv |\textsc{Load}^p()|$
5: **function** $\textsc{Inform}(\ell_{\text{ave}}, \ell^p, r = 0)$
6: **if** $\ell^p < \ell_{\text{ave}}$ **then** ▷ If is underloaded
7: $S^p \leftarrow S^p \cup \{p\}$
8: $\textsc{Load}^p() \leftarrow \textsc{Load}^p() \cup \{p \mapsto \ell^p\}$
9: **for** $i \leftarrow 1, f$ **do**
10: $p_r \leftarrow$ random sample from P
11: send(p_r, $\textsc{InformHandler}(S^p, \textsc{Load}^p(), r+1)$)
12: **end for**
13: **end if**
14: **end function**
15: **function** $\textsc{InformHandler}(S^x, \textsc{Load}^x(), r)$
16: $S^p \leftarrow S^p \cup S^x$ ▷ Add new underloaded ranks
17: $\textsc{Load}^p() \leftarrow \textsc{Load}^p() \cup \textsc{Load}^x()$ ▷ Add new loads
18: **if** $r < k$ **then**
19: **for** $i \leftarrow 1, f$ **do**
20: $R \leftarrow P \setminus S^p$ ▷ Avoid known underloaded
21: $p_r \leftarrow$ random sample from R
22: send(p_r, $\textsc{InformHandler}(S^p, \textsc{Load}^p(), r+1)$)
23: **end for**
24: **end if**
25: **end function**

V. Analysis & Proposed Changes

We propose the following algorithmic changes, discussed and motivated in the indicated subsections:

1) Iteratively refine the task assignments and the associated imbalance before performing transfers (§ V-A).
2) Perform multiple trials of the iteration process to increase the odds of avoiding local minima (§ V-A).
3) Re-compute the CMF on line 7 of Algorithm 2, taking advantage of updated rank loads as tasks are considered for transfer (§ V-A).
4) Relax the objective function used on line 11 of Algorithm 2 to avoid getting trapped in local minima (§ V-C).
5) Modify the CMF built on line 7 of Algorithm 2 to be compatible with above-average rank loads that can result from the relaxed objective function (§ V-C).

Algorithm 2 The transfer stage to choose tasks for migration based on partial knowledge gathered in the inform stage.

1: $h \leftarrow$ threshold ▷ Constant value
2: **function** $\textsc{Transfer}(\ell_{\text{ave}}, \ell^p)$
Require: $\sum_{n=1}^{|T^p|}(\textsc{Load}(T_n^p)) \equiv \ell^p$
3: $O^p \leftarrow \textsc{OrderTasks}(T^p, \ell_{\text{ave}}, \ell^p)$ ▷ Traversal order
4: $n \leftarrow 0$ ▷ Index of task to try
5: **if** CMF is original **then** $F \leftarrow \textsc{BuildCMF}(\ell_{\text{ave}})$
6: **while** $\ell^p > h \times \ell_{\text{ave}} \wedge n < |O^p|$ **do** ▷ Overloaded
7: **if** CMF is modified **then** $F \leftarrow \textsc{BuildCMF}(\ell_{\text{ave}})$
8: $o_x \leftarrow O_n^p$
9: $p_x \in S^p$ using F ▷ Pick rank sampling CMF
10: $\ell_x \leftarrow \textsc{Load}_j^p \mid p_j \equiv p_x$ ▷ Known load of rank
11: **if** $\textsc{EvaluateCriterion}(\ell_x, o_x, \ell_{\text{ave}}, \ell^p)$ **then**
12: $\ell_x \leftarrow \ell_x + \textsc{Load}(o_x)$
13: $\ell^p \leftarrow \ell^p - \textsc{Load}(o_x)$
14: $T^p \leftarrow T^p \setminus \{o_x\}$
15: $M^p \leftarrow M^p \cup \{o_x\}$ ▷ Record proposed transfer
16: $\textsc{Target}^p() \leftarrow \textsc{Target}^p() \cup \{o_x \mapsto p_x\}$
17: **end if**
18: $n \leftarrow n + 1$
19: **end while**
20: **end function**
21: **function** $\textsc{BuildCMF}(\ell_{\text{ave}})$
22: **if** CMF is original **then**
23: $\ell_s \leftarrow \ell_{\text{ave}}$
24: **else if** CMF is modified **then** ▷ Described in § V-C
25: $\ell_s \leftarrow \max(\ell_{\text{ave}}, \max(\textsc{Load}^p()))$
26: **end if**
27: $z \leftarrow \sum_{i=1}^{|S^p|} \left(1 - \frac{\textsc{Load}^p(i)}{\ell_s}\right)$
28: $p_i \leftarrow \frac{1}{z}\left(1 - \frac{\textsc{Load}^p(i)}{\ell_s}\right)$
29: $\varphi_j \leftarrow \sum_{i=1}^{j} p_i$
30: $F \leftarrow \{\varphi_i\}_{i=1}^{|S^p|}$
31: **return** F
32: **end function**
33: **function** $\textsc{EvaluateCriterion}(\ell_x, o_x, \ell_{\text{ave}}, \ell^p)$
34: **if** Criterion is original **then**
35: **return** $\ell_x + \textsc{Load}(o_x) < \ell_{\text{ave}}$
36: **else if** Criterion is relaxed **then** ▷ Described in § V-C
37: **return** $\textsc{Load}(o_x) < \ell^p - \ell_x$
38: **end if**
39: **end function**
40: **function** $\textsc{OrderTasks}(T^p, \ell_{\text{ave}}, \ell^p)$
41: **return** T^p ▷ Alternatives are discussed in § V-E
42: **end function**

6) Modify the task transfer traversal order to increase the odds of the necessary transfers being accepted (§ V-E).

A. Iteratively Refining Task Assignments

While the original algorithm has the potential to improve imbalance, it is unlikely to result in an optimal task assignment. Factors contributing to that limitation include:

1) missed opportunities for transfer due to making a poor random choice for the potential recipient rank; and,

2) each overloaded rank working in isolation allowing an underloaded rank to become overloaded by transfers from multiple ranks.

We explore iteratively refining task assignments to further reduce the imbalance, as shown on lines 6–8 of Algorithm 3. Furthermore, we employ multiple trials of this iterative process, as shown on lines 2–5, starting each trial from the state used for the previous timestep to ensure that we do not get trapped in a local minimum. We defer actual task transfers until line 13, when we have exhausted our trials and the iterations within, accepting the set of transfers from lines 9–10 that resulted in the best imbalance.

Algorithm 3 Iterative refinement of task-rank mapping.

1: $T^p_{\text{orig}} \leftarrow T^p$
2: **for** $t \leftarrow 1, n_{\text{trials}}$ **do**
3: $\quad T^p \leftarrow T^p_{\text{orig}}$ ▷ Reset for each trial
4: $\quad M^p \leftarrow \emptyset$
5: $\quad \textsc{Target}^p() \leftarrow \emptyset$
6: $\quad$ **for** $i \leftarrow 1, n_{\text{iters}}$ **do**
7: $\quad\quad \textsc{Inform}(\ell_{\text{ave}}, \ell^p, 0)$
8: $\quad\quad \textsc{Transfer}(\ell_{\text{ave}}, \ell^p)$
9: $\quad\quad$ Evaluate $\mathcal{I}_{\text{proposed}}$ using Eqn. 1
10: $\quad\quad$ Save T^p_{best}, M^p_{best}, $\textsc{Target}^p_{\text{best}}$ for lowest $\mathcal{I}_{\text{proposed}}$
11: $\quad$ **end for**
12: **end for**
13: Execute transfers defined by T^p_{best}, M^p_{best}, $\textsc{Target}^p_{\text{best}}()$

We do not employ the negative acknowledgements proposed by Menon, et al. [9] to prevent transfers originating on multiple overloaded ranks from making an underloaded rank become overloaded. However, we still want to use all available information, including transfers scheduled from the overloaded rank running Algorithm 2, when choosing a potential recipient for the next transfer. For this reason, we choose to recompute the CMF on line 7 of Algorithm 2. In contrast, in the original work, it was computed only once, on line 5.

B. Analysis of Objective Function under Iteration

We now present the results of iteration for one representative test case with our Load Balancing Analysis Framework (LBAF)[3], a Python tool for exploring, testing, and comparing load balancing strategies. We apply $n_{\text{iters}} = 10$ iterations of the original GrapevineLB algorithm, each having $k = 10$ gossiping rounds with an overload threshold of $h = 1.0$ and a fanout factor of $f = 6$, to an initial distribution of 10^4 tasks across only 2^4 out of 2^{12} total ranks, leaving the other ones without tasks. We observe in the following table the rejection rates caused by the criterion in Algorithm 2 on line 35:

[3]Available BSD-licensed at: `https://github.com/DARMA-tasking/LB-analysis-framework`

Iteration (index)	Transfers (count)	Rejected (count)	Rejection Rate (%)	Imbalance ($\mathcal{I}$)
0	-	-	-	280
1	9084	154 931	94.46	187
2	4	1654	99.76	187
3	1	1130	99.91	187
4	7	2682	99.74	185
5	6	2396	99.75	183
6	2	1143	99.83	183
7	1	1041	99.90	183
8	0	882	100.0	183
9	0	882	100.0	182
10	3	1405	99.79	182

These immediately hint at the fact that the decision criterion is too tight. We argue that the main weakness of the GrapevineLB algorithm is this high rejection rate. In fact, we believe this can be explained by observing the condition set forth in the algorithm for each overloaded rank in Algorithm 2 on line 6:

$$6: \quad \textbf{while } (\ell^p > h \times \ell_{\text{ave}} \wedge n < |O^p|)$$

Indeed, this condition enforces strict monotonicity for each of the underloaded ranks; in other words, it uses the "taxicab" norm ($||\cdot||_1$) to minimize in the $||\cdot||_\infty$ sense: this criterion is therefore not adapted to the considered minimization problem. Another way to look at the problem is as an attempt to decrease the load of overloaded ranks while never allowing a single underloaded one to become overloaded—even when this may improve the global imbalance.

As a result, these almost-full rejection rates limit load-balancing to a noticeable decrease of $\mathcal{I}$ during the first iteration of the algorithm. Subsequent iterations of the algorithm provide little benefit because $\mathcal{I}$ remains trapped in a local minimum. Furthermore, increasing k and allowing for higher values of h do not substantially affect the outcome on average (with the exception of the occasional convergence due to a nice original layout).

The condition on line 6 mandates that, after a variable number of iterations, each overloaded rank no longer is, up to a certain relative threshold h. This implies that, in the worst case, after completion of this loop,

$$\ell_{\max} \leq h \times \ell_{\text{ave}} \quad \Longleftrightarrow \quad \mathcal{I}_{\mathcal{D}} < h - 1$$

where $\mathcal{I}_D$ is the load imbalance of the distribution D of tasks across the entire set of ranks. This amounts to saying that the objective function that the algorithm aims to minimize is $F(D) = \mathcal{I}_{\mathcal{D}} - h + 1$, and that a sufficient stopping criterion is $F(D) \geq 0$ (we observe that ℓ_{ave} is by definition a constant as no loss or gain of load may occur globally).

However, $F(D) \geq 0$ is by no means necessary; in fact, if it were, there would be no guarantee that the algorithm would terminate in finite time. This is, however, ensured by the fact that, given an overloaded rank, the criterion of line 35 is tested for all of its tasks, of which there are only a finite number. While this ensures termination of the while-loop in finite time, this does not guarantee that *any* transfer will have occurred at all.

C. GrapevineLB *with Relaxed Objective Function*

To ensure a faster convergence of the algorithm, we propose to change the criterion (the original in Algorithm 2 on line 35) as follows:

Lemma 1:
The following alternate criterion:

$$37: \qquad \textbf{if } \textsc{Load}(o_x) < \ell^p - \ell_x \textbf{ then}$$

ensures that the objective function F monotonically decreases.

Proof. Cf. Appendix A. □

We remark that the new, modified criterion can be equivalently written as:

$$\textbf{if } \ell_x + \textsc{Load}(o_x) < \ell^p \textbf{ then}$$

which is indeed less strict than the the original one, for it allows, in particular, one underloaded rank to land in overloaded territory after a transfer. However, what is ensured is the maximum norm will not increase. In addition, Lemma 1 ensures that, as long as there is at least one object satisfying this criterion on at least one overloaded rank, then the optimization can continue. However, once it is no longer possible to find such a combination, then F may no longer decrease: this new criterion thus also provides a stopping criterion.

While this criterion will provide more opportunities for overload transfers than the original one, one may wonder whether it could not be further relaxed, hereby allowing for even lower rejection rates. This question is quickly answered by the following, with the same notations as in Lemma 1:

Lemma 2:
If p_i is a rank with maximum load in D, and

$$(\exists o_x \in p_i)\ (\exists p_x \in D) \quad \ell = \textsc{Load}(o_x) \geq \ell^p - \ell_x$$

then if o_x is transferred from p_x to p_i, the objective function F does not decrease (and possibly increases).

Proof. Cf. Appendix B. □

As a result of Lemma 1 and Lemma 2, we can now assert the following:

Proposition [Optimal Load-Transfer Criterion]:
The following alternate criterion:

$$37: \qquad \textbf{if } \textsc{Load}(o_x) < \ell^p - \ell_x \textbf{ then}$$

is optimal for the load transfer strategy of Algorithm 2.

Proof:
From Lemma 1 we know that this alternate criterion ensures that monotonicity is sufficient to ensure that F monotonically decreases.

Furthermore, from Lemma 2 we know that if this criterion is not met for at least one particular case, then F will no longer monotonically decrease (and will possibly increase if (o_x, p_x) is such that ℓ'_x is not maximal in D'. Therefore, the alternate criterion, being necessary and sufficient, is optimal for the considered optimization strategy. □.

With this relaxed criterion, it is possible for a prospective transfer to cause an underloaded rank to become overloaded. To accommodate this, we need to update the probability mass function used for selecting the recipient of a transfer. To keep it from becoming negative in such cases, we change the formula as shown on line 25 of Algorithm 2. Allowing ranks that are no longer underloaded to remain candidates for receiving transfers is natural with our relaxed criterion, which allows transfers as long as the recipient would become less overloaded (based on the limited knowledge of the transferring rank) than the transferring rank was just prior to the the transfer.

D. Results with Relaxed Objective Function

We now compare the above results to those obtained when applying the alternate criterion in the LBAF simulator, for the same case as above. The new acceptance verses rejection results are shown in the following table:

Iteration (index)	Transfers (number)	Rejected (number)	Rejection rate (%)	Imbalance ($\mathcal{I}$)
0	-	-	-	280
1	11 292	648	5.43	3.34
2	4044	3603	47.12	1.60
3	2201	3412	60.79	0.873
4	1324	3586	73.03	0.632
5	765	3171	80.56	0.632
6	410	2969	87.87	0.626
7	247	2794	91.88	0.626
8	159	2749	94.53	0.626
9	120	2682	95.72	0.626
10	72	2643	97.35	0.623

In contrast with what was happening with the original criterion on line 35, we see now that the rejection rate is negligible initially, then slowly increases as the global imbalance rapidly decreases. In fact, with already more than acceptable values of $\mathcal{I}$, additional iterations continue to improve the outcome, hereby experimentally validating the preceding theoretical results. This is examplified by comparing the values of $\mathcal{I}$ in both cases, shown in the following table:

Iteration (index)	Criterion 35 ($\mathcal{I}$)	Criterion 37 ($\mathcal{I}$)
0	280	280
1	187	3.34
2	187	1.60
3	187	0.873
4	185	0.632
5	183	0.632
6	183	0.626
7	183	0.626
8	183	0.626
9	182	0.626
10	182	0.623

We note that the modified algorithm has not fully run its course after iteration 10, and continues to improve $\mathcal{I}$, albeit modestly. In contrast, the original algorithm has essentially converged to a very sub-optimal local minimum and is no longer able to improve the overall imbalance after a few steps.

E. Ordering Candidate Objects

The overall gossip-based distributed load balancing algorithm described above efficiently determines which ranks are able to send or receive tasks, and rapidly shifts load to

produce an improved distribution. However, these results are based on considering tasks as candidates for migration in an essentially arbitrary order, by their identifying index or hash table iteration order. Hypothetically, considering them in some order determined by their loads could improve the process further. In this section, we discuss and analyze three different orderings: Migrate Load-Intensive Tasks, Fewest Migrations, and Migrate Most Lightweight Tasks.

1) Migrate Load-Intensive Tasks: One simple ordering would be to try to migrate the most load-intensive possible tasks from overloaded ranks (Algorithm 4). When it succeeds, the algorithm minimizes the number of transfers necessary in a given round to achieve its results. However, this comes at the expense of worst-case acceptance rates, worst-case margins for the transfer criterion in Section §V-C, excessive migration size if size correlates with load, and potentially increased round counts. Thus, we consider this primarily as a straw-man.

Algorithm 4 The algorithm for ordering tasks for selection that picks the most load-intensive tasks first during the transfer phase (see line 3 in Algorithm 2).

```
1: function OrderTasks_Descending(T^p, ℓ_ave, ℓ^p)
2:     c ← lambda (a, b) ↦                    ▷ Sort comparator
3:     { return Load(a) > Load(b) }           ▷ Descending load
4:     return Sort(T^p,c)
5: end function
```

2) Fewest Migrations: If we assume that task migrations are an undesirable necessity in order to achieve load balance, we would want to minimize their number. This may be a consideration due to migrations themselves being expensive, or secondary effects such as lost communication locality leading to increased data movement.

To achieve balance using the fewest possible migrations, we must consider task loads in comparison to the degree of overload on their current rank, as calculated on line 2 of Algorithm 5. Any task with load greater than that excess rank load is a potential candidate to resolve the overload on that rank with a single migration. To minimize the chance of a transfer being rejected, and to minimize potential overload on the recipient rank resulting from the relaxed criterion of Section §V-C, each overloaded rank considers the smallest such task as its first candidate for transfer. We identify this task, if one with a large enough load exists, on line 6. Because we expect it to be easier to place tasks with smaller loads, we then want to consider lighter-weight tasks by descending load, followed by more load-intensive tasks by ascending load. The comparator c for this sort order is implemented on lines 7–11. This order increases the likelihood of transferring the fewest number of tasks to reduce the overload.

3) Migrate Most Lightweight Tasks: If we wish to maximize the odds that tasks proposed for transfer are accepted, then we should select the tasks with the smallest loads as candidates. However, to ensure overall success of the LB process, we want to consider the most load-intensive of such tasks first so that recipients still have the largest possible underload

Algorithm 5 The algorithm for ordering tasks for selection that minimizes the number of migrations during the transfer phase (see line 3 in Algorithm 2).

```
1: function OrderTasks_FewestMigrations(T^p, ℓ_ave, ℓ^p)
2:     ℓ_ex ← ℓ^p − ℓ_ave                     ▷ Excess load on this rank
3:     if max_i T_i^p < ℓ_ex then
4:         return OrderTasks_Descending(T^p, ℓ_ave, ℓ^p)
5:     end if
6:     ℓ_cut ← min_i { T_i^p | T_i^p > ℓ_ex }          ▷ Cutoff load
7:     c ← lambda (a, b) ↦ {                  ▷ Load sort comparator
8:         if Load(a) ≤ ℓ_cut ∧ Load(b) ≤ ℓ_cut
9:             then return Load(a) > Load(b)
10:            else return Load(a) < Load(b)
11:    }
12:    return Sort(T^p,c)
13: end function
```

in which to accomodate them. Thus, we need to determine the marginal task—that is, the most load-intensive of the lightweight tasks that must be migrated for the rank to stop being overloaded. After sorting the tasks by ascending load on line 5 of Algorithm 6, this task can be identified in a single partial sum over the task loads, stopping when the sum first exceeds the overload, as shown on line 6. For the same reason that we choose this task as the first candidate to consider for transfer, we then want to consider lighter-weight tasks by descending load, followed by more load-intensive tasks by ascending load, as implemented in the comparator c_2 on line 7.

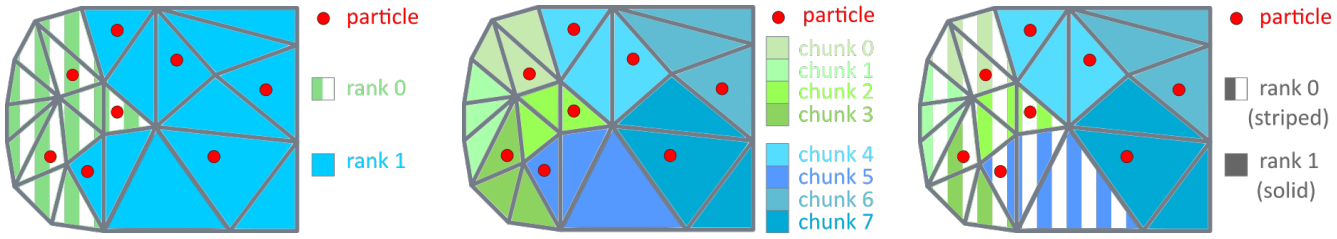
Algorithm 6 The algorithm for ordering tasks for selection that picks the most lightweight tasks first during the transfer phase (see line 3 in Algorithm 2).

```
1: function OrderTasks_Lightest(T^p, ℓ_ave, ℓ^p)
2:     ℓ_ex ← ℓ^p − ℓ_ave                     ▷ Excess load on this rank
3:     c_1 ← lambda (a, b) ↦                  ▷ Sort ascending to start
4:     { return Load(a) < Load(b) }           ▷ Ascending load
5:     S^p ← Sort(T^p,c_1)
6:     ℓ_marg ← min_j { S_j^p | Σ_{i=0}^{j} S_i^p ≥ ℓ_ex }   ▷ Partial sum
7:     c_2 ← lambda (a, b) ↦ {                ▷ Final sort comparator
8:         return if Load(a) ≤ ℓ_marg ∧ Load(b) ≤ ℓ_marg
9:                 then Load(a) > Load(b)
10:                else Load(a) < Load(b)
11:    }
12:    return Sort(S^p,c_2)
13: end function
```

VI. Empirical Results

A. Example Application

EMPIRE is an electromagnetic plasma modeling tool in development at Sandia National Laboratories. EMPIRE utilizes a Finite Element Method (FEM) on unstructured meshes for the elecromagnetic fields and a Lagrangian/particle formulation for the plasma utilizing the Particle-In-Cell (PIC) formulation [25]. EMPIRE was designed to be performance-portable through use of the Kokkos programming model. Solving for



(a) SPMD mesh decomposition (b) Overdecomposition (4 chunks/rank) (c) Overdecomposition with LB

Fig. 1: Example SPMD decomposition and overdecomposition for a simple mesh with particles.

the electromagnetic fields can be easily balanced by a static SPMD decomposition, as illustrated in Figure 1a.

For some target problems, however, the particle operations in EMPIRE's PIC capability present large, highly-variable, dynamic load imbalances. It is this nature that motivated use of the dynamic, fine-grained load balancing capability provided by vt. To avoid the overheads of overdecomposition and the AMT runtime in cases where there is not a substantial load imbalance, dual implementations of PIC as SPMD and with vt have been developed. As a result, EMPIRE's vt-based PIC implementation and vt itself have been partially co-developed in order to ensure that as much code as possible be shared between the two implementations. Furthermore, we also sought to minimize the burden on physics developers when expanding and improving EMPIRE's capabilities.

The conventional approach to load balancing in EMPIRE would be to infrequently re-partition the mesh in order to offset the evolving particle imbalance. The main issues posed by this approach are the following:

- this is an intrinsically *synchronous* process; and
- *large volumes of data* must be migrated to new ranks or recomputed from the new mesh.

With the conventional methodology, the value proposition of re-partitioning at a given timestep must consider the costs of both executing the load-balancing algorithm and, more important, re-configuring the problem in order to proceed with the next timestep (data transposition and meta-data exchange for the new partition). By making the LB step more incremental, its frequency can be adjusted to match the imbalance rate arising from migrating particles and thereby greatly reduce the re-configuration costs by amortizing it and reaping benefits earlier along the way.

In contrast with the conventional approach, we retain the initial SPMD static mesh decomposition and further decompose it by coloring the mesh of each MPI rank as shown in Figure 1b. We create a data strucure corresponding to each chunk, called a *color* in the context of EMPIRE, that contains that sub-mesh and its particles. As a simulation runs, the runtime system instrumentation captures observations of each color's evolving workload, and uses those measurements to reassign colors to different ranks, as shown in Figure 1c, in order to keep the particle workload as evenly-balanced as possible. Because EMPIRE relies on MPI-based SPMD solvers from the Trilinos suite for finite element work, execution transitions between SPMD and vt's AMT runtime are required multiple times per simulation timestep.

This novel, fined-grained, dynamic approach to the LB of particles, enabled by vt, decreases data migration cost, facilitates the overlapping of communication and computation, and avoids the cost of recalculating connectivity that would have been required for dynamically repartitioning the SPMD mesh. In § VI-B, we present the impact that load balancing has on EMPIRE's B-Dot problem, in which the particle load varies dramatically over the course of the run, but at a rate that allows us to successfully apply the principle of persistence.

B. Performance Results

All the empirical results presented for EMPIRE were run on 100 nodes (4 ranks per node, or 400 ranks) of an ARM cluster, equipped with 2.0 GHz Arm Cavium Thunder-X2 processors and 128 GB of RAM per node. The cluster is outfitted with a Mellanox EDR Infiniband interconnect.

Figure 2 presents a summary of the performance of EMPIRE in five different configurations. The total application time is broken down into two parts: Particle update and Non-particle update. The Particle update is the part of EMPIRE that we implemented in vt with tasking. The SPMD (no AMT) configuration provides the pure MPI baseline performance of EMPIRE without tasks. The other configurations all use vt, with an AMT overdecompostion factor of 24, whereby every SPMD rank mesh is split into 24 migratable chunks.

The configuration labeled AMT w/GreedyLB uses a non-scalable, centralized, greedy algorithm as a baseline to compare quality of the other strategies. AMT w/HierLB is a hierarchical, tree-based load balancing strategy (described in [22]) that is less scalable at very large scales compared to distributed algorithms. AMT w/TemperedLB is the novel distributed load balancing algorithm that we present in this paper. Finally, AMT w/GrapevineLB is a configuration of our TemperedLB that matches the original algorithm described in § IV-B.

For Figure 2, the configuration labeled AMT w/TemperedLB uses the Fewest Migrations approach from § V-E2, the ordering which provided the best overall result. The configuration labeled AMT (without LB) includes all the overhead of tasks without actually enabling the load balancer. Figure 2 exhibits that this configuration adds about a 23% overhead. This is due to the cost of creating tasks, extra communication overhead (smaller messages), and smaller kernel invocations. We found

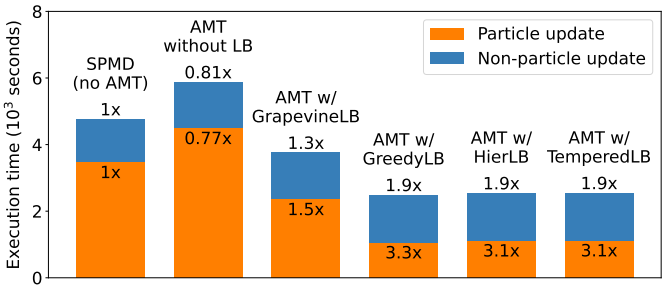


Fig. 2: Overall performance of the EMPIRE application comparing LB strategies. The multipliers are speedups compared to the SPMD baseline. The Particle update is the portion of the application implemented with tasks using our AMT programming model. The AMT w/TemperedLB bar shows the performance of TemperedLB with the Fewest Migrations ordering algorithm, which is used for all plots unless otherwise noted.

Type	t_{n}	t_{p}	t_{lb}	t_{total}
SPMD (no AMT)	1284s	3478s	0s	**4762s**
AMT without LB	1374s	4501s	0s	**5875s**
AMT w/GrapevineLB	1388s	2363s	5s	**3756s**
AMT w/GreedyLB	1419s	1063s	5s	**2488s**
AMT w/HierLB	1416s	1117s	8s	**2541s**
AMT w/TemperedLB	1416s	1118s	11s	**2546s**

Fig. 3: Execution time breakdown: non-particle execution time (t_{n}), particle execution time (t_{p}), LB + migration execution time (t_{lb}), and total execution time (t_{total}).

that although 24 chunks adds significant overhead, it performs the best with load balancing because it gives the runtime more flexibility in migrating tasks. When AMT is enabled with load balancing, we observe a 1.9x speedup over the whole application and $\sim$3x speedup over the part of the application where we added tasks, except for GrapevineLB, which only achieved a speedup of 1.3x and 1.5x, respectively, for the whole application and the part where we added tasks.

Figure 3 displays a table breaking down the execution time graphed in Figure 2. We find for all three load balancing configurations the cost of running the load balancer is small compared to the application time. Our novel load balancing implementation, called TemperedLB, takes slightly longer than the others due to the number of trials (10) and iterations (8) we utilize and the cost of migrations (which dominates t_{lb}) proposed by the load balancer, although fewer trials would have sufficed.

In Figure 4a, we plot the total execution time per timestep for each configuration. For all the empirical results with load balancing enabled, except for AMT w/HierLB, we run the load balancer on the second timestep and then every 100^{th} timestep. In contrast, we run HierLB differently, preferentially migrating the most load-intensive tasks on the second timestep, then preferentially migrating the most lightweight tasks on the fourth timestep and then every 100^{th} as we do in the other configurations. The spikes observed are the extra cost of running the load balancer, resizing RDMA buffers post-LB, and computing application-specific (physics) diagnostics on the same interval.

Figure 4b graphs the maximum and minimum per-rank task load for each configuration with load balancing. In contrast to execution time, task load excludes idle time. The Lower bound (max) curve plots the maximum of the average per-rank task load and the load of the largest task in the system, which provides a lower bound for the best performance achievable. Our TemperedLB performs well compared to HierLB between timesteps 800 and 1100, when the task loads are rapidly evolving. In Figure 4c, we plot the imbalance metric $\mathcal{I}$ computed from the per-rank task loads. We observe that, without LB, $\mathcal{I}$ starts around 7 and then decreases to $\sim$3.3 by the end of the execution. This change is due to the average rank load increasing as the amount of particle work increases.

Figure 4d compares particle update time for the various traversal orders presented in § V-E. The best performing overall was Fewest Migrations (§V-E2), motivating our use of it as our representative TemperedLB run in previous plots.

VII. Concluding Remarks

The main contribution in this paper is a set of major improvements to seminal work on fully distributed load balancing by Menon et al., following a thorough study of the hypotheses underpinning the original algorithm. Specifically, we have identified weaknesses in several aspects of its load transfer phase, which we have addressed by establishing new theoretical results to justify the optimality of our relaxed transfer criterion, proposing new object selection strategies heuristics, and studying these with a LB simulator developed in the context of this work. Furthermore, we have presented our implementation of this vastly improved load balancer (TemperedLB) in the context of vt, our open-source asynchronous tasking library. This implementation has allowed us to empirically demonstrate the performance of our approach by applying it to a plasma physics application, at scale, with large workloads with time-varying imbalance.

We acknowledge that one of our proposed improvements to the ordering of candidates for task migration, the Migrate Most Lightweight Tasks has not demonstrated superior performance to the other proposed methods including the straw-man used for baseline comparison. While this might be merely a reflection of properties specific to the task distribution patterns typical to the target application, the fact that this methodology did not deliver superior results warrants further study, which may give us additional insight into distributed load transfer dynamics. Moreover, we want to apply our improved algorithm, TemperedLB, to other full-scale applications than the one we considered in this work. This will allow us to confirm the empirical performance findings with a broader set of task distribution patterns that we can use to further study the respective performance of the task migration strategies in the simulator. Finally, because the overarching goal of this work is not to reduce or even eliminate load imbalance for its own

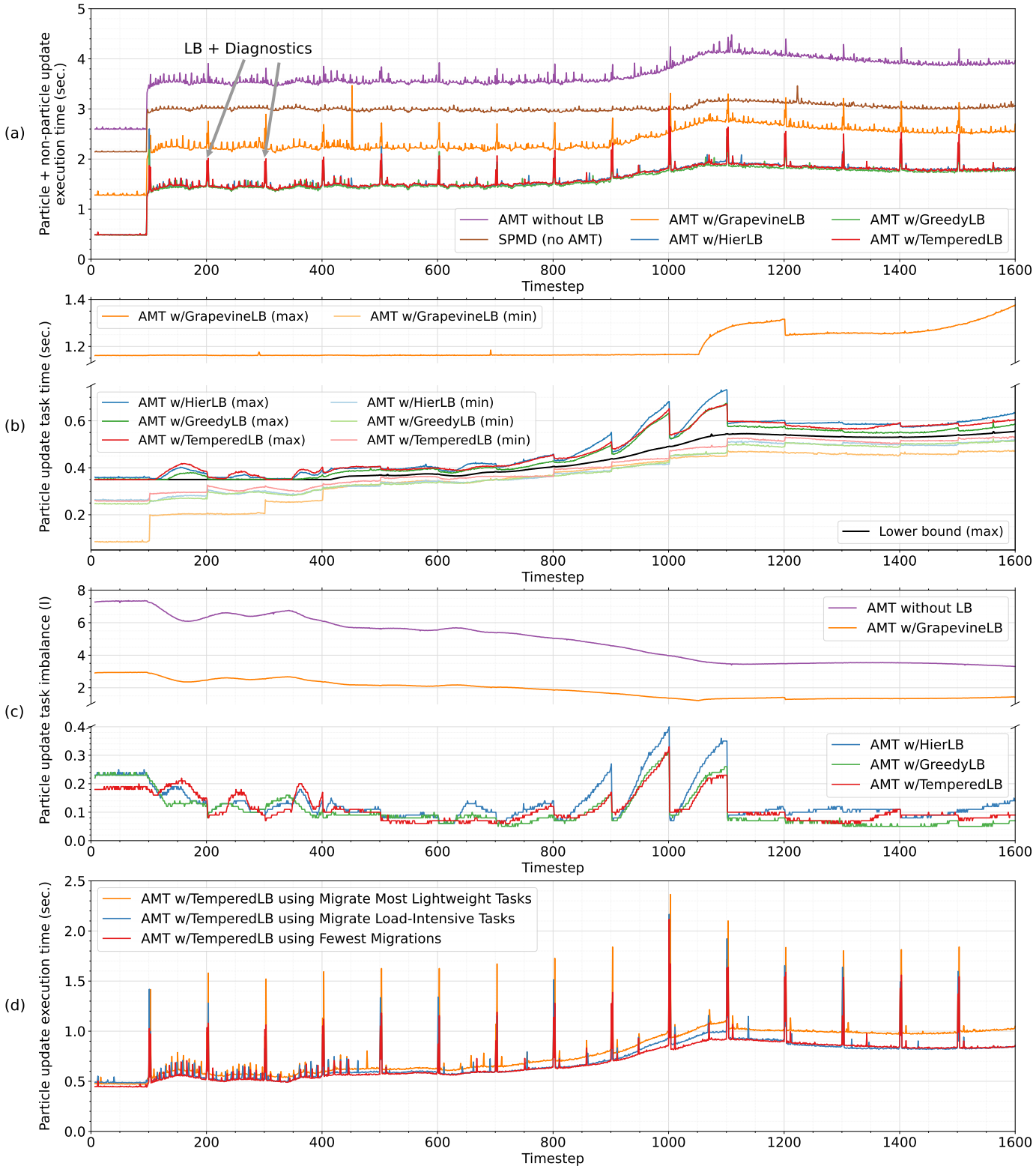


Fig. 4: Comparison of algorithms. We omit the first several steps that show initial transient conditions.
(a): Full step (particle update and non-particle update) time per timestep of EMPIRE.
(b): LB statistics in the particle update over time. Note that task load excludes idle time.
Max: maximum per-rank task load across all ranks;
Min: minimum per-rank task load across all ranks;
Lower bound (max): maximum of ℓ_{ave} and the load of the most load-intensive task.
(c): Imbalance metric $\mathcal{I}$ (described in § III-C) computed on the per-rank task load of particle update.
(d): Particle update time with three LB task traversal ordering algorithms for deciding which tasks to migrate (§ V-E).

sake—but rather to make simulations run faster—our future work will consider inter-task communication costs in addition to task load.

References

[1] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, pp. 1–11, best Algorithms Paper Award.

[2] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems," in *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, 1994, pp. 97–106.

[3] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[4] S. Reinhardt and G. Karypis, "A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[5] C. Chevalier, F. Pellegrini, I. Futurs, and U. B. I, "Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework," in *In Proceedings of Euro-Par 2006, LNCS 4128:243–252*, 2006, pp. 243–252.

[6] G. Karypis, K. Schloegel, and V. Kumar, "Parmetis: Parallel graph partitioning and sparse matrix ordering library," *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[7] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[8] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[9] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503284

[10] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[11] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 41–88, 1999.

[12] G. Harel, J.-B. Lekien, and P. P. Pébaÿ, "Visualization and analysis of large-scale, tree-based, adaptive mesh refinement simulations with arbitrary rectilinear geometry," *arXiv preprint arXiv:1702.04852*, 2017.

[13] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. V. Kalé, and P. Ricker, "Scalable algorithms for distributed-memory adaptive mesh refinement," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012, pp. 100–107.

[14] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale, "A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model," in *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.

[15] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008, pp. 1–12.

[16] B. Hendrickson and E. Womble, "The torus–wrap mapping for dense matrix calculations on massively parallel computers," *Siam J Sci Stat Comp*, Jan 1994.

[17] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, L. Kale, and T. Jones, "Mapping dense LU factorization on multicore supercomputer nodes," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 596–606.

[18] A. Petitet and J. Dongarra, "Algorithmic redistribution methods for block-cyclic decompositions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 12, pp. 1201–1216, 1999.

[19] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms," in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95, 1995, pp. 207–216.

[21] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC*, 2009.

[22] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287103

[23] J. Lifflander, P. Miller, N. L. Slattengren, N. Morales, P. Stickney, and P. P. Pébaÿ, "Design and implementation techniques for an mpi-oriented AMT runtime," in *Workshop on Exascale MPI, ExaMPI@SC 2020, Atlanta, GA, USA, November 13, 2020*. IEEE, 2020, pp. 31–40. [Online]. Available: https://doi.org/10.1109/ExaMPI52011.2020.00009

[24] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[25] M. Bettencourt, D. Brown, K. Cartwright, E. Cyr, C. Glusa, P. Lin, S. Moore, D. McGregor, R. Pawlowski, E. Phillips, N. Roberts, S. Wright, S. Maheswaran, J. Jones, and S. Jarvis, "EMPIRE-PIC: a performance portable unstructured particle-in-cell code," *Communications in Computational Physics*, 2021, accepted.

Appendix

A. Proof of Lemma 1

Consider a strictly overloaded (resp. underloaded) rank p_i (resp. p_x) in an load/rank distribution D, with loads ℓ_i (resp. ℓ_x). We recall that $\ell_{\max}$ and ℓ_{ave} respectively denote the maximum and average loads across all ranks, and that h is the chosen relative imbalance threshold (cf. §IV-B). Consider also an object $o \in p_i$ such that $\ell = \textsc{Load}(o) < \ell_i - \ell_x$ (both sides being strictly positive, by hypothesis on loads); we can thus distinguish the two following disjoint cases:

1) If $\ell \leq \dfrac{\ell_i - \ell_x}{2}$ then $\ell_x + \ell \leq \ell_i - \ell$, in which case $\max(\ell_i - \ell, \ell_x + \ell) = \ell_i - \ell < \ell_i$.
2) If $\ell > \dfrac{\ell_i - \ell_x}{2}$, then $\ell_i - \ell < \ell_x + \ell$, in which case $\max(\ell_i - \ell, \ell_x + \ell) = \ell_x + \ell < \ell_i$.

Therefore, overall, $\max(\ell_i - \ell, \ell_x + \ell) < \ell_i$. Now, recall that

$$F(D) = \frac{\ell_{\max}}{\ell_{\text{ave}}} - h \geq \frac{\ell_i}{\ell_{\text{ave}}} - h,$$

with equality if and only if p_i is such that $\ell_i = \ell_{\max}$ in D (i.e., p_i has maximum load in the original distribution D). Therefore, $\ell < \ell_i - \ell_x$ ensures that

$$\frac{\max(\ell_i - \ell, \ell_x + \ell)}{\ell_{\text{ave}}} - h < \frac{\ell_i}{\ell_{\text{ave}}} - h \leq F(D)$$

Assume now that o is transferred from p_i to p_x, and denote $\ell_i' = \ell_i - \ell$ and $\ell_x' = \ell_x + \ell$ the respective new loads of these ranks. The new maximum load in the new load/rank distribution D' may then be encountered on p_i, p_x (possibly both of these), or neither of these. Therefore, at least one of the following cases is encountered (the two first ones thus not being mutually exclusive):

1) If p_i is such that $\ell_i' = \ell'_{\max}$ in D' (i.e., p_i has maximum load in the new distribution D'), then $\max(\ell_i - \ell, \ell_x + \ell) = \ell_i - \ell = \ell_i'$ and (A) yields:
$$F(D') = \frac{\ell'_{\max}}{\ell_{\text{ave}}} - h = \frac{\ell_i'}{\ell_{\text{ave}}} - h < \frac{\ell_i}{\ell_{\text{ave}}} - h \leq F(D)$$
2) If p_x is such that $\ell_x' = \ell'_{\max}$ in D' (i.e., p_x has maximum load in the new distribution D'), then $\max(\ell_i - \ell, \ell_x + \ell) = \ell_x + \ell = \ell_x'$ and (A) yields:
$$F(D') = \frac{\ell'_{\max}}{\ell_{\text{ave}}} - h = \frac{\ell_x'}{\ell_{\text{ave}}} - h < \frac{\ell_i}{\ell_{\text{ave}}} - h \leq F(D)$$
3) If a rank $p_Y \notin \{p_i, p_x\}$ is such that $\ell_Y' = \ell'_{\max}$ in D' (i.e., neither p_i nor p_x have maximum load in the new distribution D'), then necessarily $\ell_Y' = \ell_Y$ because the transfer did not affect p_Y, and thus necessarily $\ell_Y' \leq \ell_i$ and (A) yields:
$$F(D') = \frac{\ell'_{\max}}{\ell_{\text{ave}}} - h = \frac{\ell_Y'}{\ell_{\text{ave}}} - h \leq \frac{\ell_i}{\ell_{\text{ave}}} - h \leq F(D).$$
Furthermore, in this case, $\dfrac{\ell_Y}{\ell_{\text{ave}}} - h = \dfrac{\ell_i}{\ell_{\text{ave}}} - h$ if and only if p_Y was maximally-overloaded in D; because there is only a finite number of such ranks, it is guaranteed that the inequality becomes strict in a finite number of iterations.

We can therefore conclude that, overall,

$$\ell < \ell_i - \ell_x \Longrightarrow F(D') < F(D).$$

□

B. Proof of Lemma 2

If p_i has maximum load in the object/load distribution D, and one can find $o \in p_i$ and $p_x \in D$, then by definition of F one has, on one hand:

$$F(D) = \frac{\ell_i}{\ell_{\text{ave}}} - h \leq \frac{\ell_x + \ell}{\ell_{\text{ave}}} - h.$$

On the other hand, in the new distribution D' obtained by transferring o from p_i to p_x, one has:

$$\frac{\ell_x + \ell}{\ell_{\text{ave}}} - h = \frac{\ell_x'}{\ell_{\text{ave}}} \leq \frac{\ell'_{\max}}{\ell_{\text{ave}}} - h = F(D').$$

Combining the two above inequalities thus yields $F(D) \leq F(D')$; in other words, F remains constant at best, or increases at worst. Furthermore, if (o, p_x) is such that ℓ_x' is not a maximally-overloaded rank in the new distribution, then the latter inequality is strict, in which case F strictly increases from D to D'. □.