SANDIA REPORT
SAND2020-3967
Unlimited Release
Printed April 2020

DARMA/vt FY20 Mid-Year Status Report

Jonathan Lifflander, Philippe P. Pébaÿ

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

SAND2020-3967
Unlimited Release
Printed April 2020

DARMA/vt FY20 Mid-Year Status Report

Jonathan Lifflander
Sandia National Laboratories
Livermore, CA 94551
U.S.A.

Philippe P. Pébaÿ
NexGen Analytics
Sheridan, WY 82801
U.S.A.

Abstract

The goal of this report is to provide a comprehensive status report of the research & development conducted in the context of the DARMA project by the end of the first quarter of fiscal year 2020. It follows in particular [LBS+19] and [PL19].

Acknowledgments

The authors would like to thank Matt Bettencourt (Sandia) and Phil Miller (NGA) who have reviewed and provided valuable edits to this report.

Contents

1 Introduction **9**

2 An Application-Driven Effort **11**

2.1 EMPIRE-PIC: an Overview 11

2.2 Load-Balancing Challenges 12

2.3 The DARMA/vt Tasking Library 12

2.3.1 Fundamental Tasking Abstractions in DARMA/vt 14

3 New Load-Balancing Results **19**

3.1 A Proposed Measure of Persistence 19

3.2 Effect of Load Caching during Migration 22

3.3 Communication-Aware Load-Balancing 25

3.3.1 Problem Statement 25

3.3.2 Strict Localization 26

3.3.3 Relaxed Localization 28

3.3.4 Experimental Comparison 29

4 Conclusion **31**

References 33

List of Figures

2.1 Visualization showing particle (e^-, p^+) distribution in two different timesteps of an EMPIRE plasma physics simulation across 4 processors: initial state (left) and time-evolved state (right). 11

2.2 An initial decomposition (partition) of a mesh into 2 subsets for 2 processors (no overdecomposition) with particles in some mesh elements: each process is assigned exactly one subset of the partition. 13

2.3 A two-level partition of the above mesh into $k = 2 \times 4 = 8$ chunks for 2 processors. Overdecomposition is an enabling transformation for subsequent load balancing on the smaller chunks. 13

2.4 The task-to-processor mapping after running the load balancer on the overdecomposed, two-level partition of the mesh. 13

2.5 Notional flowchart of the interplay between the two levels of mesh resolution and the two phases of EMPIRE-PIC as executed by the DARMA/vt runtime. 18

3.1 Coefficient of variation over four successive phases computed for three different object times across a 500-phase simulation; note that there is an offset of length 3 between the data points and the actual phase (i.e., values plotted at index 0 correspond to the coefficient computed at phase 3). 20

3.2 3-, 4-, and 10-persistence coefficients of object times computed for an ensemble of 64 objects across a 500-phase simulation, truncated above at 0.1; note that there are offsets of corresponding lengths between the data points and the actual phase (i.e., the value plotted at index 0 corresponds to the 10-persistence computed at phase 10). 21

3.3 Effect on load-balancing of caching the per-processor loads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full ($f^k = 128$) or limited ($f^k = 4$) information. 23

3.4 Effect on load-balancing of caching the per-processor loads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full information ($f^k = 128 = |\mathbf{P}|$): square areas are proportional to aggregate per-processor loads. 24

3.5 Effect on load-balancing of utilizing know recipient processor underloads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full ($f^k = 128 = |\mathbf{P}|$) or limited ($f^k = 4 < |\mathbf{P}|$) information. 25

3.6 Left: load-balancing oblivious to communication, achieving perfectly balanced loads in terms of per-processor work; right: communication-aware load-balancing strictly preserving local (on-rank) communication sub-graphs. Center: color scale of per-processor loads; note that processors are also scaled so that their surface areas are commensurate to their respective loads. 27

3.7 Effect on load-balancing of using strict (left) vs. relaxed (right) localization, in the presence of full information ($f^k = 256 = |\mathbf{P}|$) for an initial case (center) with 1024 objects distributed across 16 out of 64 processors: square areas are proportional to aggregate per-processor loads; communication edges are not shown in the final figures to avoid visual clutter. 30

This page intentionally left blank.

Chapter 1

Introduction

We briefly recall that the main motivation underpinning the DARMA (Distributed Asynchronous Resilient Models and Applications) project is to research and develop novel programming model paradigms for next-generation HPC applications at Sandia, across NNSA laboratories, and beyond, to study and implement intelligent, dynamic, and self-aware distributed runtimes that automatically guide application execution.

Specifically, this has been conducted by investigating how to fundamentally shift the expression and execution of massively concurrent HPC scientific algorithms to:

- be more asynchronous and aware of data dependencies;
- provide resilience to executional aberrations in heterogeneous and/or unpredictable environments, such as system noise and network contention;
- enable runtime reconfigurability (e.g., load-balancing or task remapping) to better support dynamic environments and problem domains which can quickly render predetermined partitionings suboptimal.

In this context, we have articulated our research of such novel programming models around two main tasking concepts: *asynchronous execution*, and *overdecomposition*. Whereas the former is self-explanatory and generally understood as almost synonymous with tasking itself, the latter deserves further elaboration.

While asynchrony provides fundamental benefits itself, a machine-dependent decomposition (or partition) of a scientific application domain reduces flexibility and portability at run-time. Overdecomposition is an approach to partitioning that is conducted on a logical basis natural to the application area (e.g., dimensionality of a matrix to be inverted; dimensionality of the computation space), allowing the application domain expert to program in an abstract, machine-independent fashion, while giving the runtime system a new, valuable degree of freedom to remap and expose communication edges of the computational graph. As a result, the synergy of these two pillars provide a sound basis to allow for building highly portable codes that will scale to exascale, including not-yet-existent platforms.

This page intentionally left blank.

Chapter 2

An Application-Driven Effort

2.1 EMPIRE-PIC: an Overview

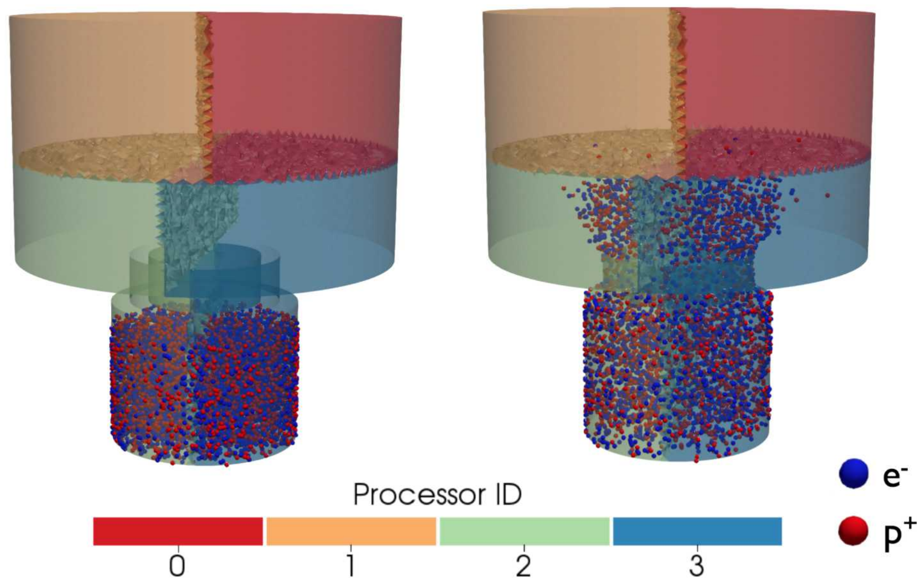

Figure 2.1: Visualization showing particle (e^-, p^+) distribution in two different timesteps of an EMPIRE plasma physics simulation across 4 processors: initial state (left) and time-evolved state (right).

EMPIRE (for Electro-Magnetic Plasma In Realistic Environments) is an Advanced Technology Development & Mitigation (ATDM) plasma physics application that includes a Particle-in-Cell (PIC) algorithm, with the following main characteristics:

- It has initial particle distributions that can be spatially concentrated, creating *heavy load imbalance*, as illustrated in Figure 2.1, left.
- The particles may move rapidly across the domain, hereby inducing *workload variation over time*, as shown in Figure 2.1, right.
- The pre-existing MPI-based EMPIRE code base *does not support load-balancing* (LB).

- *Hybrid PIC/fluid* configurations present a difficult challenge for traditional LB solutions (multi-objective).

2.2 Load-Balancing Challenges

The conventional, to-date approach has been to infrequently re-partition the mesh decomposition in order to offset 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 processors 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 (LB) algorithm and more importantly, re-configuring the problem 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, ours retains the initial MPI static mesh decomposition, but further overdecomposes the initial partition with a two-level scheme that splits the particles into k subsets, as illustrated in Figures 2.2, 2.3, and 2.4. Overdecomposition is a powerful approach, enabling transformation that allows the runtime to load-balance where particles are processed based on the dynamic load profile of each subset in the overdecomposed partition.

This novel, fined-grained, dynamic approach to the LB of particles both decreases data migration cost and facilitates the overlapping of communication and computation. It is enabled by DARMA's virtual transport (DARMA/vt) tasking library, discussed in the next section.

2.3 The DARMA/vt Tasking Library

DARMA/vt is a novel runtime tasking library for C++ that is built for general scientific applications with a focus on Sandia's specific tasking needs. Thus, it is designed and implemented to adhere to the following fundamental principles:

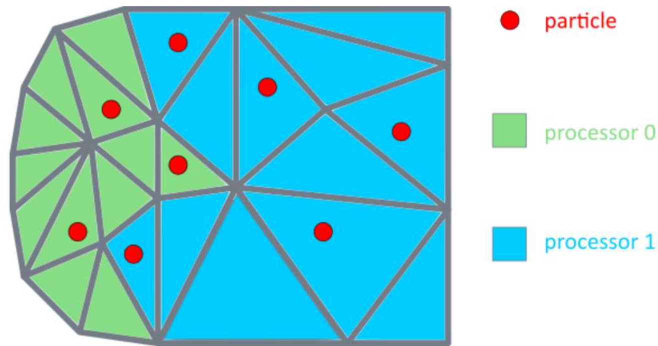

Figure 2.2: An initial decomposition (partition) of a mesh into 2 subsets for 2 processors (no overdecomposition) with particles in some mesh elements: each process is assigned exactly one subset of the partition.

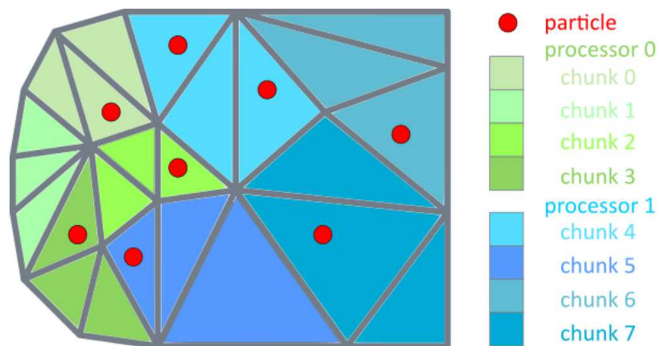

Figure 2.3: A two-level partition of the above mesh into $k = 2 \times 4 = 8$ chunks for 2 processors. Overdecomposition is an enabling transformation for subsequent load balancing on the smaller chunks.

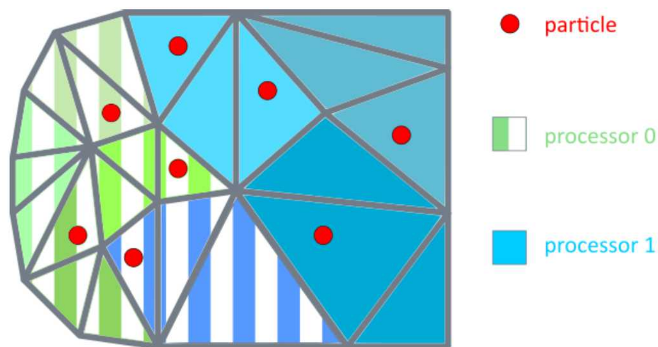

Figure 2.4: The task-to-processor mapping after running the load balancer on the overdecomposed, two-level partition of the mesh.

1. be fully inter-operable with MPI[1];

[1]This principle is extremely important given Sandia's large, existing code bases which cannot reasonably be re-written completely in a new library. MPI interoperability allows the application to share MPI abstrac-

2. enable and favor an incremental adoption model for the C++ "task-ification" of applications;
3. dynamically migrate data and work off-processor;
4. provide scalable[2] load-balancers;
5. integrate communication costs in its load-balancing objective functions;
6. be co-developed and performance tuned with key Sandia applications (e.g., EMPIRE-PIC and NimbleSM, a Sierra/SM mini-app).

In order to satisfy the aforementioned requirements, DARMA/vt abstracts MPI ranks (or parts of them) as distinct C++ objects (either singletons or collections) with efficient data transfer via active messages and/or RDMA (Remote Direct Memory Access). Its interoperable runtime interface easily composes with MPI, on-node threading runtimes, and GPUs (including OpenMP, Kokkos, and CUDA), and provides load balancing on the overdecomposed abstractions through object/task migration across MPI ranks.

In [LBS+19], we demonstrated the preliminary significance of DARMA/vt by enabling dynamic load-balancing of time-varying workloads in the context of EMPIRE simulations, hereby mitigating performance imbalances on heterogeneous architectures.

2.3.1 Fundamental Tasking Abstractions in DARMA/vt

DARMA/vt provides some basic abstractions that interoperate with MPI at the rank level implemented as C++ singletons:

1. A *context* (`vt::theContext()`) that has a one-to-one mapping with a MPI rank and provides information about the current task running (`vt::NodeType()`), number of ranks, threads available on the current rank, MPI communicator being used by vt, and the current vt runtime instance.
2. A *group* (`vt::theGroup()`) that has a projection onto a MPI group (each vt group can provide an underlying MPI group—a superset), representing a scalable subset of the communicator (or vt context). Unlike MPI groups, DARMA/vt groups can be constructed in a rooted or collective manner. Each vt group is backed by a spanning tree over the subset of nodes included. A collective vt group is built with a novel, fully distributed algorithm that constructs a spanning tree with no centralization.
3. An *active messenger* (`vt::theMsg()`) that sends and receives messages (classes that inherit from `vt::Message`) asynchronously and without blocking between vt nodes. It

tions with DARMA/vt, seamlessly and efficiently moving between DARMA/vt and MPI (i.e., time and space sharing).

[2]Fully distributed LB data collection and strategies that ship Exascale-ready

manages distributed events (local or remote tracking of `MPI_Requests`) through events (`vt::theEvent()`) to provide user subscribable notifications of local and remote completion.

4. A *termination detector* (`vt::theTerm()`) that detects and tracks causal relationships between message sends and receives and therein allows user-defined semantic grouping of related tasks. The termination detector is backed by several distributed algorithms based on the extensive research literature studying termination (including Dijkstra-Scholten [DS80] and wave-based [] algorithms). vt allows the user to construct distributed epochs to track sub-computations and thereby sequence groups of tasks based on data dependencies.

5. An *object group* (`vt::theObjGroup()`) that provides a rank-based encapsulation of localized behavior to form semantic groupings of application-specific logic and better re-usability. Object groups are non-migratable collective instances that have a single instantiation per node. They provide a easy transition to encapsulate parts of an MPI rank which can then be reused and given an identifying distributed handle for communication and runtime load balancing.

6. A *scheduler* (`vt::theSched()`) that provides a central scheduler on each node for processing work units: messages, events, or data transfers. The vt scheduler allows fine-grained user prioritization of work units including the task graph expansion criteria: breadth-first, depth-first, or a combination thereof.

7. A *trace* (`vt::theTrace()`) module that optionally records a concurrent trace of the execution sequence and the dependencies between the nodes. It generates Projections [] logs for *a posteriori* performance analysis.

8. A *registry* (`vt::theRegistry()`) module that registers active functions, methods, and objects to invoke/migrate them across address spaces. The registry uses static template initialization order in C++ to create a universal integer handle for handlers across multiple address spaces even in the face of address-space layout randomization (ASLR).

9. Serialization is an optional component (DARMA/checkpoint) that DARMA/vt can be compiled with. It allows any C++ types that have a serialize method/function associated with them to be migrated across nodes. DARMA/checkpoint is a general library that can be adapted for I/O or creating snapshots disk.

Beyond these rank-based constructs, DARMA/vt provides several abstractions for transforming a rank-decomposed application domain to a "task-ified"/overdecomposed domain in C++.

1. A *virtual context* (`vt::theVirtualManager()`) module that aids in constructing lightweight, migratable C++ objects that messages and data can be directed. Virtual contexts provide the foundation for encapsulating data that may be operated on across address spaces.

2. A *virtual context collection* (`vt::theCollection()`) module that enables the creation of n-dimensional, dense or sparse, insertable collections that get mapped to physical nodes. The collection abstraction is the foundation of virtualization for scientific application domains. Instead of an application developer splitting their domain into a one-dimensional rank-linearized (P) chunks, the user can choose the dimensionality of their collection based on the physical domain. For a three-dimensional particle simulation, the user might create a chunked, three-dimensional collection that maps to the physical spatial domain. Such decomposition helps greatly with determining neighborhood or ghosting data transfer.

3. A *callback* (`vt::theCB()`) module provides a higher-level abstraction to creating a general endpoint across components to transfer data. For instance, when developing a new component, it may export data as an output after it runs. This output might need be sent to many sources, depending on other components. Callbacks provide a general way to export data without knowing the exact type of endpoint (broadcast to object group or collection, etc.).

4. Collection chain sets (`vt::messaging::CollectionChainSet`) provide an important facility in which users can sequence events occurring across the distributed system based on data dependencies. The collection chain set works with `vt::messaging::PendingSend` to queue up future sends or other events and then make them conditional on a preceding epoch to sequence them (see termination detection).

5. Under the hood, DARMA/vt must route messages to virtualized endpoints. These endpoints are migratable, implying that their location may change during the execution. Virtual contexts (singletons or collections) may be migrated by the user or by invoking the load balancer. The location manager (`vt::theLocMan()`) manages a cache of known locations of virtualized endpoints. It routes messages, forwarding them when it misses in the cache or the cached location is stale due to migrations. It guarantees that messages will eventually be delivered to the endpoint even under extreme conditions when an object is migrated many times.

6. DARMA/vt provides an index module to create multi-dimensional index classes based on a specified index traits class. It ships with n-dimensional dense index class that is used for creating basic multi-dimensional virtual context collections. With the index traits provided, the user is allowed to define their own index classes with a well-defined scheme. For example, for tree-based AMR, the user could create an index class that uses a bit-string to identify collection elements with a certain level of refinement in that tree.

7. DARMA/vt also provides a mapping module to define default mapping functions for indices used by collections. When a virtual context collection is initially created, it needs a map function to define how the collection will be scalably mapped to the underlying MPI ranks. DARMA/vt ships with n-dimensional mapping schemes for row- or column-major blocked mappings and round robin mappings. By following the provided mapping interface, the user is allowed to create custom mapping schemes that may be useful for their specific application.

8. DARMA/vt ships with distributed, scalable load balancers that can be automatically invoked on their virtual context collections. To load balance, DARMA/vt automatically instruments the data transfer and tasks that virtual contexts execute during a *phase*. A phase is a distinct iteration or timestep of a application demarcated by the user based on their domain. Within a phase, DARMA/vt collects instrumentation across virtual contexts and provides all this data to the load balancers in a highly scalable manner (no centralization of instrumentation data). As the user runs their application, he or she can pick a distributed load balancing to run at each phase with a command line parameter or load balancing specification file. In the future, we plan to add automatic mechanisms to determine the most profitable times to run the load balancer based on the load balancer cost and object migration cost.

In Figure 2.5 we provide a notional diagram of one step of the integrated finite element electro-magnetic solver – iterative particle ODE integrator interplay, showing in particular the alternation between rank-decomposed mesh (used by the SPMD MPI solver) and the overdecomposed mesh (used by DARMA/vt for asynchronous particle movement).

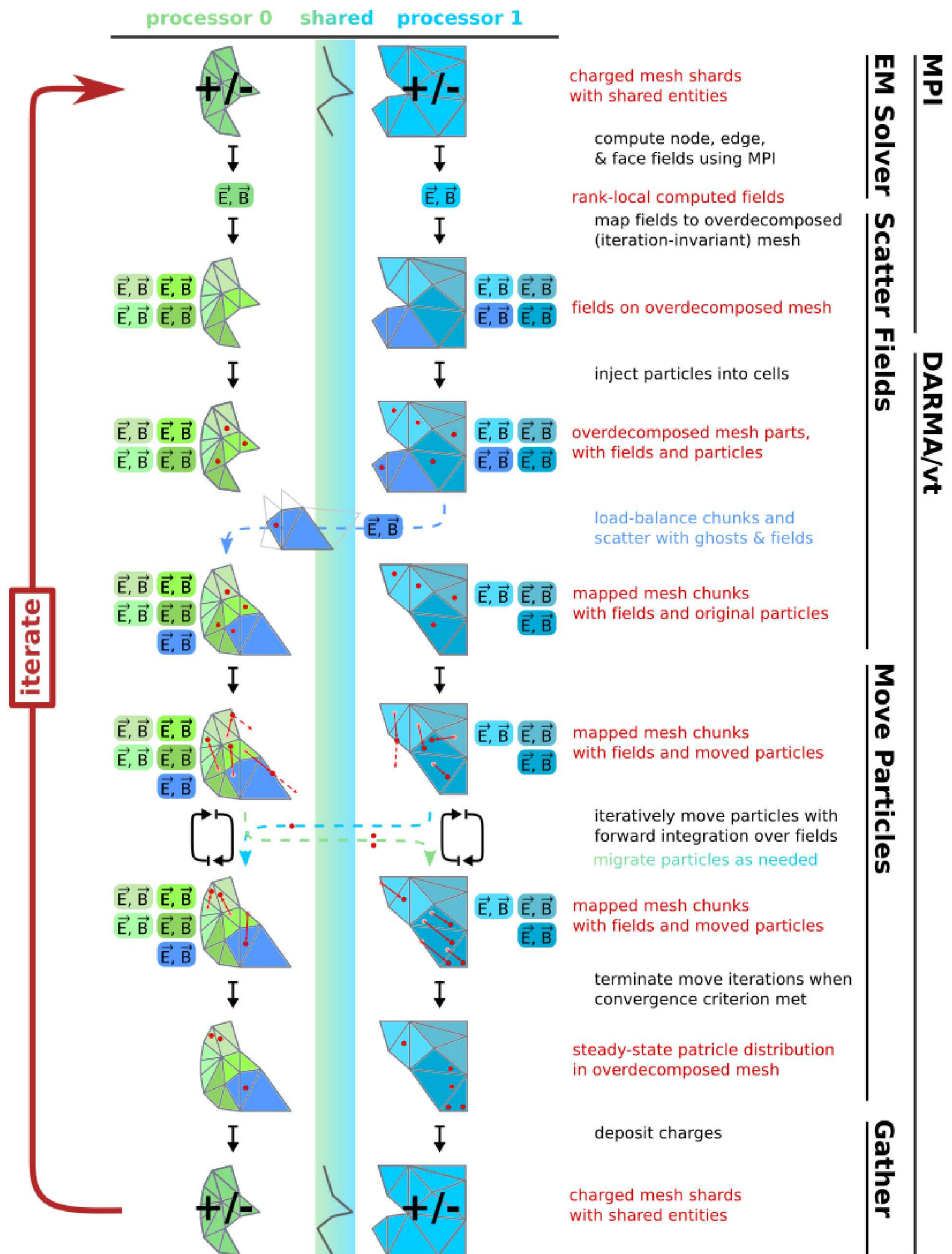

Figure 2.5: Notional flowchart of the interplay between the two levels of mesh resolution and the two phases of EMPIRE-PIC as executed by the DARMA/vt runtime.

Chapter 3

New Load-Balancing Results

3.1 A Proposed Measure of Persistence

One key assumption to statistically-based fully-distributed load-balancing, using Grapevine-like algorithms, is that of *persistence* of object loads as the computation progresses.

Albeit relatively intuitive, this notion does not appear to have received, to-date, a general treatment in the open literature. We are therefore now proposing a general methodology for the quantitative assessment of this concept, in particular in (but not limited to) the context of per-object loads.

We begin by recalling that, given a population of X (in our case with finite cardinality $n = |X|$) with arithmetic mean $\mu(X) \neq 0$, its *coefficient of variation* is defined as follows:

$$c_V(X) = \frac{\sigma_n(X)}{\mu(X)},$$

where $\sigma_n(X)$ denotes the standard deviation of X (uncorrected for bias as we do not assure normality, nor sub-sampling of X).

The main rationale for the use of c_V as a quantitative estimate of persistence is manifold, in particular:

1. X contains values of variables expressed in the same units (in particular, no relative units).
2. It is expected that no constant type zero measurements will be observed.
3. We are not seeking to build confidence intervals.
4. We are not looking for measurement assurance (in which case standard error would be more suitable).

Furthermore, in order to capture persistence over $k \in [\![2; +\infty[$ successive simulation phases (or steps), we compute this the coefficient of variation of the quantity of interest (here, per-object times) across as sliding window of length k, as $c_V(X_{i,-k})$ where $X_{i,-k}$ denotes the

subset of values in X with indices $i-k+1$ to i, where $i \in [\![k-1;+\infty[$ is the phase index of interest (assuming that the initial phase corresponds to $i=0$).

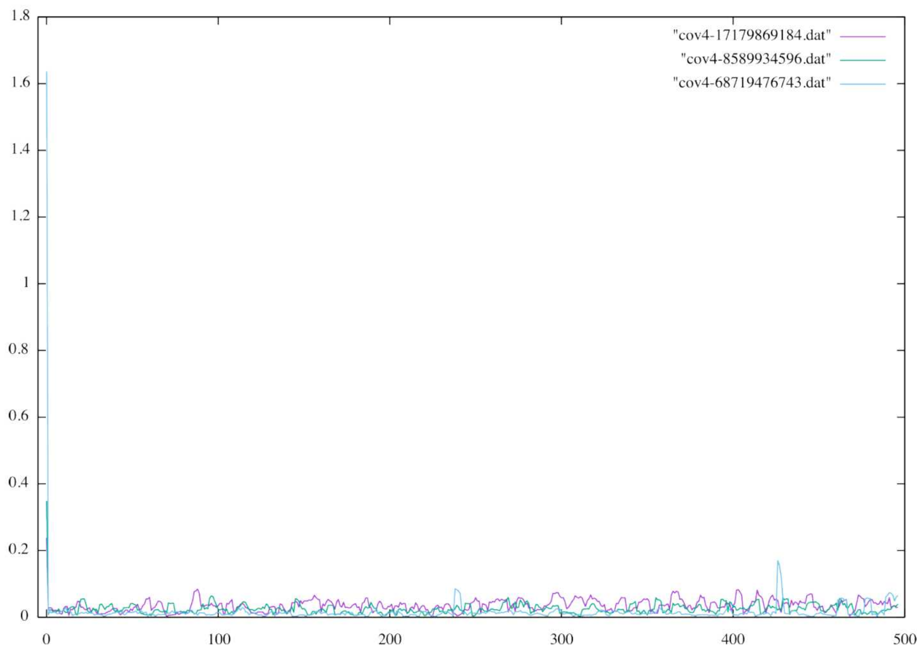

Figure 3.1: Coefficient of variation over four successive phases computed for three different object times across a 500-phase simulation; note that there is an offset of length 3 between the data points and the actual phase (i.e., values plotted at index 0 correspond to the coefficient computed at phase 3).

This methodology is illustrated in Figure 3.1, where $c_V(X_i, 4)$ is computed for the times of three separate objects, at each phase $i > 2$ of a 500-phase simulation: each of these plots illustrates that, after an initial sharp variation due to cache warm-up and other initialization transients, the time-windowed per-object coefficients of variation constantly remain well below 10%.

From these plots one can readily conclude that the initial object time values exhibit too much relative variation with respect to their time-averaged baselines to allow for a successful attempt at load-balancing. In contrast, it appears that during the remainder of the simulation, these objects' respective times only vary mildly with respect to the values observed in the three preceding phases.

However, possibly very many objects participate in the simulation, and it is not feasible to examine each individual time-sliding coefficient of variation. Instead, a practical measurement of overall object time persistence is simply the arithmetic mean of all coefficients, for it is sensitive to extreme values and, being bounded-below (by 0) but not above, this mean will readily respond to each large relative variation. We thus propose, as a measure of overall

per-object time persistence, the following *k-persistence coefficient* of object times:

$$\mathscr{P}_k(\mathbf{O}) = \frac{1}{|\mathbf{O}|} \sum_{O \in \mathbf{O}} c_V(\ell_{i,-k}(O)),$$

where $\ell_{i,-k}(O)$ denotes the subset of times of an object O from phases $i-k+1$ to i.

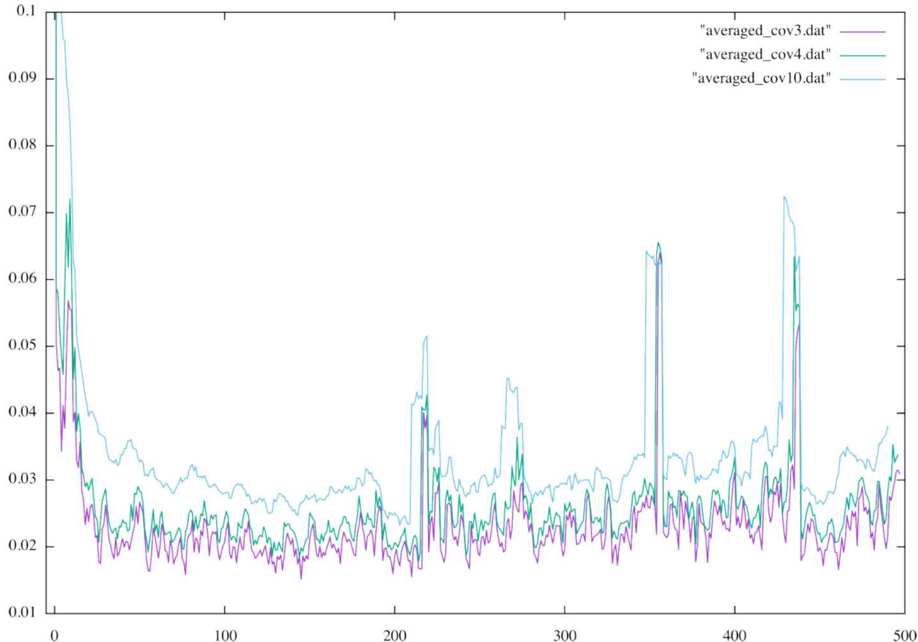

Figure 3.2: 3-, 4-, and 10-persistence coefficients of object times computed for an ensemble of 64 objects across a 500-phase simulation, truncated above at 0.1; note that there are offsets of corresponding lengths between the data points and the actual phase (i.e., the value plotted at index 0 corresponds to the 10-persistence computed at phase 10).

An application of this k-persistence coefficient is illustrated in Figure 3.2 for values of k in $\{3; , 4; 10\}$ where $\mathbf{O}$ contains a total of 64 objects, including those whose individual time coefficient of variations are illustrated in Figure 3.1.

We observe that as the length of the time-window increases, the averaged relative variation increases but its own variability decreases. However, overall, these different values of k yield qualitatively similar information, as it is important to note that the vertical axis is capped at 10%, in order to emphasize the low overall relative variation past the first few phases: even the spikes in Figure 3.2 remain below the rate of 0.08.

Based on these preliminary observations, we suggest using our newly-defined k-persistence metric as an experimental tool to guide persistence-based load-balancing.

3.2 Effect of Load Caching during Migration

In [PL19], we had proposed a novel load-balancing algorithm, with a noticeable improvement with respect to pre-existing work (cf. [MK13]). Both approaches use an information phase to propagate information about underloaded processors across their entire collection, followed by a migration phase where then overloaded processors attempt to offload some of their work onto the underloaded ones. Both phases use (pseudo-) random selection of "targets", incorporating a statistical bias in the migration phase in order to primarily target the most underloaded processors. We note that these types of methods are often referred to as *gossiping* algorithms, as a result of their reliance on random propagation of information only decided at the local level, without recourse to centralized control or steering.

This succession of information and migration phases can be iteratively applied, and it is in the setting of our `NodeGossiper` Python simulator that we demonstrated an increase of performance with the use of [MK13]. Meanwhile, once implemented in DARMA/vt, as `GossipLB`, and applied a real EMPIRE-PIC case, we noticed that the same algorithm was achieving sub-optimal performance when compared to the possible load-balance improvements demonstrated with the `NodeGossiper`. For instance, in a case with 1024 objects with the following statistical properties of their individual loads (or times) ℓ:

$\|\ell\|$	$\min \ell$	μ_ℓ	$\max \ell$	σ_ℓ	$\mathscr{I}_\ell$
1024	0.000982	0.0494	0.617	0.0499	11.5

initially mapped to 128 processors, resulting in the per-processor load distribution $\mathscr{L}$ as follows:

$\|\mathscr{L}\|$	$\min \mathscr{L}$	$\mu_{\mathscr{L}}$	$\max \mathscr{L}$	$\sigma_{\mathscr{L}}$	$\mathscr{I}_{\mathscr{L}}$
128	0.00870	0.395	2.61	0.0499	5.60.

After $i = 10$ iterations of the `NodeGossiper` with $f = 128$ and $k = 1$ (for perfect information with f^k equal to the cardinality of the processor set), the following almost optimal load-balance:

$\|\mathscr{L}\|$	$\min \mathscr{L}$	$\mu_{\mathscr{L}}$	$\max \mathscr{L}$	$\sigma_{\mathscr{L}}$	$\mathscr{I}_{\mathscr{L}}$
128	0.373	0.395	0.617	0.0259	0.562

is achieved. It is interesting to note that similar results are obtained, almost irrespective of the chosen values for the fanout factor f and number of gossiping rounds k, in just a few iterations. However, in the real DARMA/vt run, the `GossipLB` was only able to achieve a load imbalance oscillating about $\mathscr{I} \approx 1,6$, found by `GossipLB`. a clear under-performance as compared to the results of the `NodeGossiper`.

As we suspected that this discrepancy was due to the fact that, in the DARMA/vt context, object migration occurs in an asynchronous fashion, the aggregated loads on recipient processors as it is known by the sender is *not* recomputed after each object migration from the latter to the former: rather the computed load information regarding the underloaded (and thus, receiving) processors as known initially to the overloaded (and thus, sender) ones is only that which results from the gossiping phase, and remains constant throughout the migration phase. As a result, any criterion that uses recipient load as part of its input parameters (which is the case for both the original [MK13] and modified [PL19] algorithm will potentially be impacted by this *information asymmetry*.

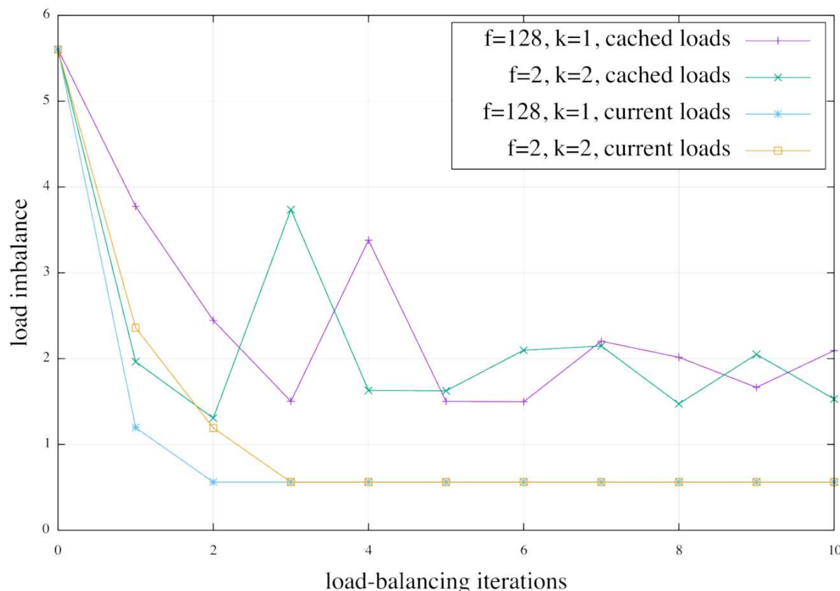

Figure 3.3: Effect on load-balancing of caching the per-processor loads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full ($f^k = 128$) or limited ($f^k = 4$) information.

In order to better analyze this phenomenon as well as validate the results of DARMA/vt's `GossipLB`, we have implemented the optional caching of recipient processor loads in the `NodeGossiper`, so that these values are only current after the end of the gossiping phase as long as no migration has occurred. And indeed with such caching turned on, the `NodeGossiper` matches almost exactly the behavior of `GossipLB`, as illustrated by the green and purple lines in Figure 3.3 and a load imbalance oscillating about $\mathscr{I} \approx 1,6$ but not being able to improve further. This is in contract with the optimal load-balancing obtained when caching is off, illustrated by the cyan and yellow lines in Figure 3.3 which both rapidly converge, almost irrespective of the k and i input parameters, towards $\mathscr{I} = 0,562$.

Figure 3.4 provides what may be an even more intuitively understandable visual representation of the initial distribution and of the load distribution results in the presence of complete information, allowing for a more intuitive comparison of the obtained mapping after ten iterations of the `NodeGosspier`, with or without caching of the recipient processor loads.

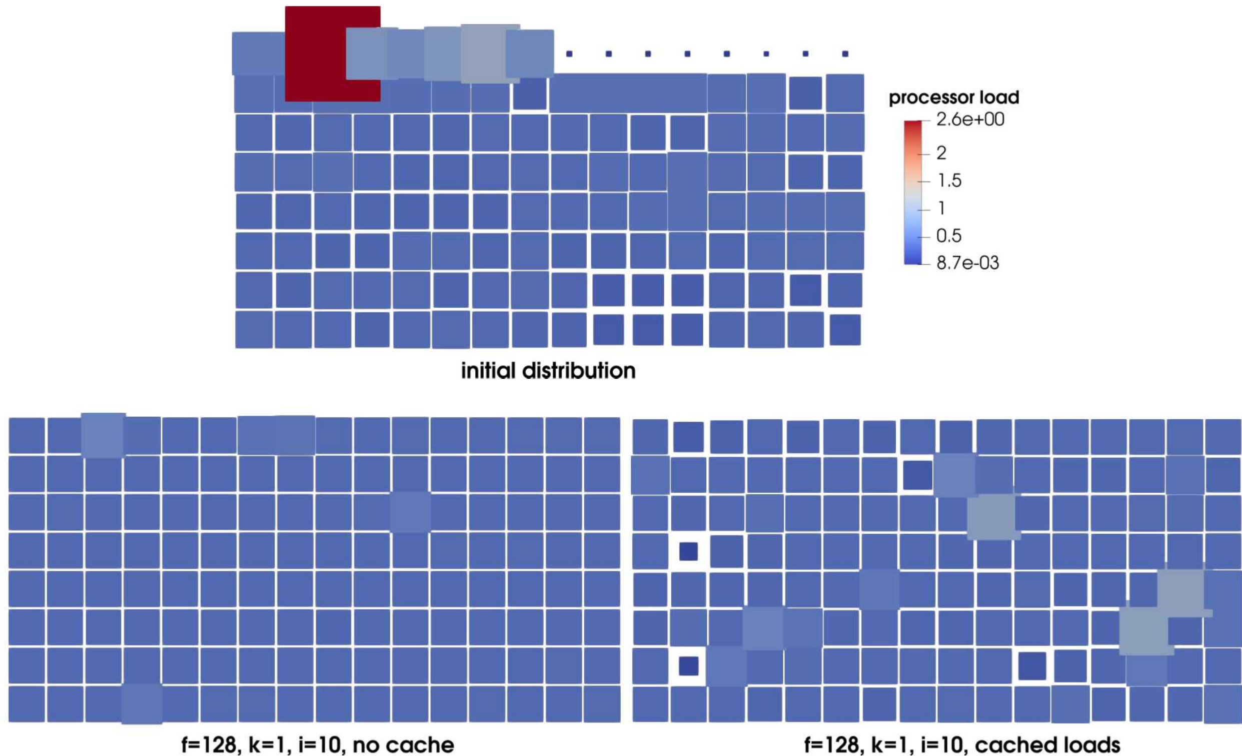

Figure 3.4: Effect on load-balancing of caching the per-processor loads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full information ($f^k = 128 = |\mathbf{P}|$): square areas are proportional to aggregate per-processor loads.

This caching option allowed us to confirm the validity of the sub-optimal load imbalance, found by `GossipLB` for this case, for the reason that we initially suspected, namely information asynchrony. In order to resolve this problem, cause by information asynchrony between sender and recipient, we propose to further amend the original Grapevine algorithm, by updating the CMF used to sample target processors during the transfer phase in order to incorporate the change of information caused by each object migration. Specifically, the computing of this CMF (F_j) in steps `1` and `2` of Algorithm 2 shall be transferred to the inside of the **`while`** loop (lines `3-11`). As a result, when an underloaded processor X is picked (`5`) and the transfer criterion (original `6` or our own modified `6'`) is satisfied, the correspondingly new load of $X = P_j \in S$ is indeed taken into account in the re-computing of its picking probability p_j and, therefore, in F_j. One additional improvement in order to speed up this re-computing is to altogether drop from S those underloaded processors that are no longer underloaded or, better yet, which no longer satisfy our improved transfer criterion `6'` with respect to the considered sender processor[1].

The results obtained of the modified criterion using this new improvement, which we call the *post-migration CMF update*, are shown for the same case as before in Figure 3.5: we can

[1]We note that this is by nature a *processor-local* criterion which therefore cannot be used during the gossiping phase, which uses a *global* criterion to cut through underloaded and overloaded processors.

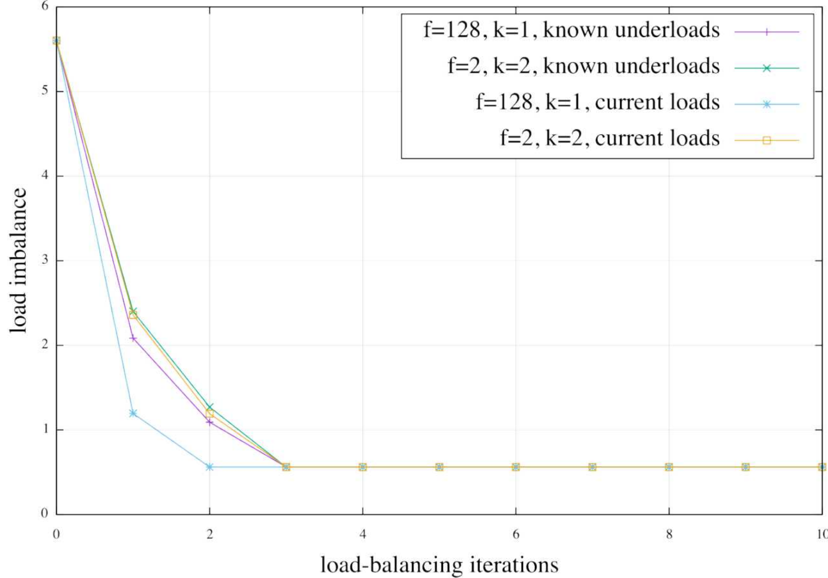

Figure 3.5: Effect on load-balancing of utilizing know recipient processor underloads during each migration iteration, as opposed to re-computing these after each migration, in the presence of full ($f^k = 128 = |\mathbf{P}|$) or limited ($f^k = 4 < |\mathbf{P}|$) information.

observe that the results of the fully-synchronous load-balancing are almost entirely retrieved in terms of rate of convergence, and fully retrieved in terms of consistency and stability.

3.3 Communication-Aware Load-Balancing

In order to achieve better performance with actual cases, it is necessary to take into account the cost of inter-processor communication. Meanwhile, the fully distributed approach discussed above also allows for the use of communication-aware optimization criteria, instead of the purely load-based ones mentioned in 3.2. This approach, which iteratively refines workloads with incremental changes, can preserve localized communication graph and thus optimize load balance by trading off communication for work.

3.3.1 Problem Statement

Consider the simple, academic case a pair of two objects $\mathbf{O} = \{O_1; O_2\}$ is to be distributed across a set of processors $\mathbf{P} = \{p_1; p_2\}$, and $load(O_1) = load(O_2) = \ell$ (in the sense of time needed to perform these tasks), and denote as follows the 4 possible object/processor mappings: $m_{ij} = \{O_1 \to p_i; O_2 \to p_j\}$. Both m_{12} and m_{21} have null imbalance, whereas for both m_{11} and m_{22} $\mathscr{I} = 1$. In other words, in terms of object/processor balance for each of the 4 possible distributions, exactly 2 are optimal and occur when exactly one object in $\mathbf{O}$

is assigned to each processor in $\mathbf{P}$.

Meanwhile, consider now the additional piece of information whereby some data must be communicated between O_1 and O_2 before they may proceed to performing their respective pieces of work with equal time ℓ. For the sake of simplicity, assume also that this data transfer is instantaneous when both objects reside on the same processor, whereas it takes an amount of time $\tau > 0$ when they do not. When factoring in this new parameter, we see that the total time T that is required to perform the entire work (i.e., executing all objects to completion) takes on the following values for each of the 4 possible configurations:

$$T(m_{11}) = T(m_{22}) = 2\ell \quad ; \quad T(m_{12}) = T(m_{21}) = \ell + \tau.$$

Evidently, depending on the relative values of ℓ and τ, the best use of the available resources is not necessarily an optimal object/processor distribution: specifically, these two possible optima coincide in our example if, and only if, $\tau < \ell$. From this simple, idealized example we can thus readily observe that taking into account inter-object communication costs can have determinant impact on what is considered an optimal distribution, in order to minimize the overall compute time.

3.3.2 Strict Localization

One first approach in order to factor in communication costs is to consider that $\tau = \infty$, i.e., to impose an absolute penalty on the cost of inter-processor communication; the main interest of this approach is to explore the opposite extreme of the load/communication trade-offs, as our work so far has only focused on pure load-balancing.

Recall that, for each overloaded processor $p \in \mathbf{P}$, the transfer phase of the Grapevine algorithm (i.e.. Algorithm 2 in [MK13]) iterates all objects O_i assigned to overloaded processor p_o, picks a potential target processor p_t using a previously computed PMF. We define the *in-* (respectively *out-*) *communicator of* an object O_i, as the subset of $\mathbf{O}$ that contains all objects that have an incoming (respectively outgoing) communication edge with O_i, i.e.

$$c_+(O_i) = \left\{ O \in \mathbf{O}, \exists \overrightarrow{OO_i} \in \mathbf{E} \right\},$$
$$c_-(O_i) = \left\{ O \in \mathbf{O}, \exists \overrightarrow{O_iO} \in \mathbf{E} \right\},$$

where $\mathbf{E}$ denotes the set of all communication edges defined between the objects in $\mathbf{O}$. The *communicator of* O_i is thus defined as

$$c(O_i) = c_-(O_i) \cup c_+(O_i).$$

In this context, the strict localization criterion allows for the transfer of O_i from p_o to p_t when the following condition is met:

$$6_{\texttt{strict_loc}}: \qquad \textbf{if } c(O_i) \cap p_o = \varnothing \textbf{ then} \tag{3.3.1}$$

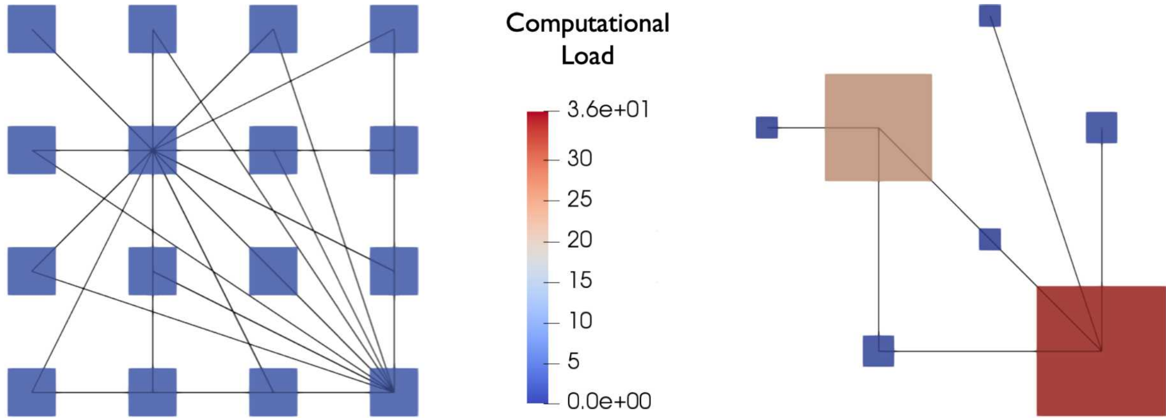

Figure 3.6: Left: load-balancing oblivious to communication, achieving perfectly balanced loads in terms of per-processor work; right: communication-aware load-balancing strictly preserving local (on-rank) communication sub-graphs. Center: color scale of per-processor loads; note that processors are also scaled so that their surface areas are commensurate to their respective loads.

where inclusion is understood here in the sense of a collection of processors all assigned to p_o. It is important to note, in particular, that this criterion does not make use of any information regarding the proposed target processor, p_t.

This is illustrated in Figure 3.6, which compares the load and communication distribution (*object map*) prescribed by our method with the same input data, depending on whether a "load-only" (left, $\tau = 0$) or a "strict localizer" (right, $\tau = \infty$) criterion is used. In this case, the initial setting consisted of 64 objects of identical unit load are uniformly pseudo-randomly distributed on only 2 processors out of of a total of 16. In addition, unit-size communication inter-object communication links are pseudo-randomly sampled from a binomial distribution with average out-degree of 2 from each object. In this figure, only the final results are shown for comparison purposes. One shall note that while the processors are represented as squares whose surface areas are proportional to their respective aggregated loads, graph edges are in contrast all of identical width and only represent the existence of at inter-processor communications reflecting communications between the objects they own.

While in the "load-only" case an optimal load balance is achieved as expected because communication edges are assumed to have zero associated times, the "strict localizer" allows fewer for fewer object transfers, as inter-process communication is then considered infinitely costly. As can be seen in rightmost graph, the two initially overloaded processors remain so at the end of the `NodeGossiper` iterations (convergence was indeed achieved), with only some underloaded processors having received objects that were not initially communicating locally. Furthermore, one object o_1 transferred away from one of the 2 initially (over)loaded processors (call it p_1) can necessarily communicate with either no other, or an object p_2 located or the only other of the two initially loaded processors (call it p_1). In both cases, o_1 may only have been transferred to one of the underloaded processors, i.e. one of the 14

initially unloaded ones (call it p_u). Because o_1 was *not* communicating with other objects on P_1 (otherwise, it would not have been transferred away from it), this transfer *create* a communication edge between p_1 and p_u. As a result of this, one can see that the edges shown in Figure 3.6, right, result from pre-existing communication edges with objects not initially present on the processors to which these edges are connected. This results in a very "stiff" pattern where load imbalance cannot decrease much: however, because $\tau = \infty$, this is to be expected as compute time is indeed negligible compared to communication time.

While strict localization might be applicable for some problems, the current DARMA/vt implementation in EMPIRE has communication patterns that render it too strict to be useful. Specifically, in EMPIRE-PIC, as depicted in the flow chart 2.5, execution each PIC timestep transitions from SMPD/MPI-only to DARMA/vt. This transition is due to the solver being implemented in pure SMPD MPI inside Trilinos. After the solver completes its runs, the E and B fields are transferred to the appropriate over-decomposed mesh colors (sub-chunks in the two-level decomposition). Initially, before the load balancer transfers any objects, the MPI-only-to-DARMA/vt transfer is local—each over-decomposed mesh color is mapped to its "home" MPI rank. By transferring these mesh colors to another physical rank, DARMA/vt effectively balances the load profile across ranks when spatial particle concentrations are uneven.

However, this implementation implies that any migration away from the "home" (or initial starting point) MPI-only rank will always imply an additional non-local data transfer. The E and B fields will thus need to be transferred from the SMPD/MPI-only home rank when it arrives after the solver finishes to where the migrated mesh color resides. Thus, any strict localization algorithm will reject any potential migration because there is a local data edge from mesh color object to MPI-only object that will become remote after migration occurs to any other rank.

3.3.3 Relaxed Localization

We now explore more relaxed version of a localization-based transfer criterion. In contrast with strict localization introduced above, the *relaxed localizer* criterion is defined as follows:

$$\mathsf{6}_{\texttt{relaxed_loc}}: \quad \textbf{if} \sum_{O \in c(O_i) \cap p_t} w(O, O_i) \geq \sum_{O \in c(O_i) \cap p_o} w(O, O_i) \textbf{ then} \tag{3.3.2}$$

where $w(O, O_i)$ denotes the sum of the weights of communication edges between objects O and O_i (both ways). In other words, the relaxed localizer allows for the transfer of an object only when this results in a more localized communicator, in terms of aggregated communication weights. We thus prove the following material conditional

Proposition 3.3.1. *The following material conditional holds:*

$$\mathsf{6}_{\texttt{strict_loc}} \implies \mathsf{6}_{\texttt{relaxed_loc}} \tag{3.3.3}$$

but its converse statement is false.

Proof. If the premise of (3.3.1) is true, then

$$\sum_{O \in c(O_i) \cap p_o} w(O, O_i) = \sum_{O \in \varnothing} w(O, O_i) = 0.$$

Because all communication edge weights are non-negative by definition, the left-hand side of the premise of (3.3.2) is necessarily greater than or equal to 0, wherefrom the implication ensues.

Conversely, consider a case where O_i has a exactly (non-zero) communication edges: one of weight 1 with $O_j \in p_o$ and the other with $O_k \in p_t$ communication edge, and formed with an object O_j assigned to p_t, i.e. $c(O_i) \cap p_o = \{O_j j$ and $c(O_i) \cap p_t = \{O_k\}$; therefore,

$$\sum_{O \in c(O_i) \cap p_o} w(O, O_i) = w(O_j, O_i) = 1$$

while

$$\sum_{O \in c(O_i) \cap p_t} w(O, O_i) = w(O_k, O_i) = 2,$$

hereby ensuring that the premise of (3.3.2) is satisfied while that of (3.3.1) is not. From this counter-example, we see that the converse statement of (3.3.3) is false. □

3.3.4 Experimental Comparison

In order to better understand the potential improvements provided by relaxing the localization criterion, we conducted a broad number of experiments with various configurations, and provide here that illustrates typical findings that allowed us to conclude that, for realistic cases, the relaxed criterion approach appears valid. The experimental comparison provided here uses a synthetic case generated with 1024 objects, with times sampled from $U(0, 1)$, with the following statistics observed:

$\lvert\ell\rvert$	$\min \ell$	μ_ℓ	$\max \ell$	σ_ℓ	$\mathscr{I}_\ell$
1024	0.000348	0.0492	0.999	0.287	1.03

.

In addition, inter-object communication edges were created with for each object a number of outgoing edges sampled from a binomial distribution with parameters 0.5 and 4 (i.e., with between 0 and 4 outgoing edges and therefore an average out-degree of 2). For each such edge, a communication weight was drawn from a log-normal distribution with mean and variance both equal to 1 (i.e., $\log \mathscr{N}(-0.346574; 0.832555)$, yielding a set of 2060 edge weights $\mathscr{W}$ with the following statistics:

$\lvert\mathscr{W}\rvert$	$\min \mathscr{W}$	$\mu_{\mathscr{W}}$	$\max \mathscr{W}$	$\sigma_{\mathscr{W}}$	$\mathscr{I}_{\mathscr{W}}$
2060	0.0397	1.01	9.45	0.976	8.39.

Finally, these 1024 objects were pseudo-randomly distributed across 16 out of 64 processors, resulting in the per-processor load distribution $\mathscr{L}$ with the following statistics:

$\lvert\mathscr{L}\rvert$	$\min\mathscr{L}$	$\mu_{\mathscr{L}}$	$\max\mathscr{L}$	$\sigma_{\mathscr{L}}$	$\mathscr{I}_{\mathscr{L}}$
64	0	7.87	38.1	13.9	3.84.

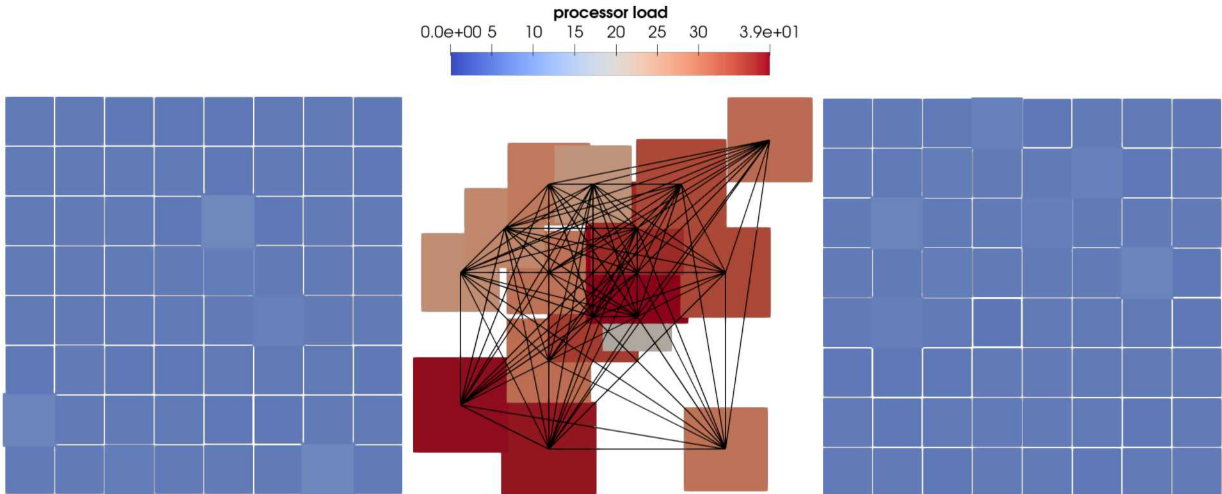

Figure 3.7: Effect on load-balancing of using strict (left) vs. relaxed (right) localization, in the presence of full information ($f^k = 256 = |\mathbf{P}|$) for an initial case (center) with 1024 objects distributed across 16 out of 64 processors: square areas are proportional to aggregate per-processor loads; communication edges are not shown in the final figures to avoid visual clutter.

This initial state is illustrated in Figure 3.7, center, with square areas are proportional to $\mathscr{L}$ values and existing edges are shown but not their respective weights for the sake of legibility.

After $i = 40$ iterations of the `NodeGossiper` with $f^k = 256 > 128 = |\mathbf{P}|$ to ensure statistically perfect information, we obtained the results illustrated in Figure 3.7, left, with the strict localizer, and in Figure 3.7, right, with the relaxed localizer: by visual inspection we can see that fewer broad patches are left after the load-balancing phase. When attempting to perform a qualitative assessment, one realizes that results vary substantially across runs, because the transfer criterion (communication-based) does not guarantee that the objective function (load-based) decreases between iterations: in 200 experiments (100 for each criterion) we have observed final values of $\mathscr{I}_{\mathscr{L}}$ in the $[0.256; 0.922]$ range which, albeit much better than the original one, does not stabilize towards an optimal balance. However what we observes is that the relaxed localization criterion has a higher average transfer acceptance rate than the strict one. From this we conclude that a relaxed localizing criterion indeed allows for more transfer than a strictly localizing approach, but that it must be combined with a load-based metric in order to ensure monotonic decrease of the objective function.

Chapter 4

Conclusion

In this report, we have recalled the main motivations of the DARMA project, and why the novel programming model paradigm it presents has the potential to drastically change that way we way HPC applications are developed and utilized at Sandia and beyond.

We have then focused on the EMPIRE family of applications to explain why it has been selected as a primary application to conduct DARMA's research and development, due to the specific challenges posed by these applications. In particular, this work is conducted in a co-development context between the DARMA runtime library and, specifically, the EMPIRE-PIC application, so that the latter immediately benefits from the former, while the former is guided by actual findings made by the latter on realistic cases.

This has driven us to focus with particular emphasis on the issue of dynamic load-balancing, made necessary by the nature of the physics simulated in EMPIRE-PIC, and made possible by DARMA/vt's tasking library. We have explained the fundamental underpinnings of our approach, and shown in particular why overdecomposition not only allows applications prone to both transient and major load imbalances to continue to progress in theory, but also in practice thanks to DARMA/vt's fundamental tasking abstractions. For this reason, we have provided the first in-depth description of these abstractions, hereby making this report an important intermediate reference point for both documentation of existing implementations and future work.

Because load-balancing is such a crucial aspect of applications targeted by DARMA/vt, we have dedicated a chapter of this report to our novel results in this area, specifically regarding: the assessment of load persistence to drive the spawning of load-balancing phases; the effect of incomplete/inexact information in task-based distributed setting; and communication aware load-balancing. For each of these aspects, we have provided both theoretical and experimental results, hereby providing a way forward for further improvements of DARMA/vt's load-balancing capabilities.

Future work will, in particular, focus on the utilization at full-scale of these new techniques, and will expand the scope of DARMA/vt to broader domain areas.

This page intentionally left blank.

References

[DS80] E. W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[LBS+19] J. Lifflander, M. Bettencourt, N. Slattengren, G. Templet, P. Miller, P. Pébaÿ, M. Perrinel, and F. Rizzi. DARMA-EMPIRE integration and performance assessment – interim report. Sandia Report SAND2019-1134, http://prod.sandia.gov/techlib/access-control.cgi/2019/191134.pdf, January 2019.

[MK13] Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 1–11, New York, NY, USA, 2013. ACM.

[PL19] P. Pébaÿ and J. Lifflander. Some results about distributed load balancing. Sandia Report SAND2019-5139, http://prod.sandia.gov/techlib/access-control.cgi/2019/195139.pdf, May 2019.

DISTRIBUTION:

1	MS	9018	Central Technical Files, 08944
1	MS	0899	Technical Library, 09536
1	MS	N/A	Jonathan Lifflander, 08753
1	MS	N/A	Robert L. Clay, 08753
1	MS	9159	Philippe Pébay, 08753

Sandia National Laboratories