VECTOR LOAD BALANCING FOR HIGH-PERFORMANCE PARALLEL APPLICATIONS

BY

RONAK BUCH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

ABSTRACT

Load balancing, modifying the distribution of work across a system to optimize resource usage, is vital for achieving scalability and high performance for dynamic parallel applications. However, as applications and systems grow increasingly complex, established methods of measuring and balancing load are becoming less effective. Traditionally, load balancing only considered the total amount of execution time, stored as a scalar value, as a metric for load, but this fundamentally cannot capture other performance-limiting execution properties such as phase-based iteration structures due to dependencies between tasks, the simultaneous division of work across host CPUs and accelerator devices, or the impact of other resource constraints such as memory footprint. This inability to accurately characterize the nature of "load" causes balancing strategies to make poor placement decisions, leading to suboptimal performance.

In this thesis, we propose *vector load balancing*, in which load is a vector of values rather than a scalar, and analyze its potential to improve the quality of load balancing for these complex applications and systems. We discuss the mathematical underpinnings of vector load balancing and challenges in measuring and utilizing vector loads, developing several new load balancing algorithms designed specifically for use with vector loads. We demonstrate both simulated and practical speedups across several different applications and programming models using these new techniques. Finally, we develop optimization techniques to improve the performance of vector load balancing strategies to make them viable at scale.

ACKNOWLEDGMENTS

El libro no es un ente incomunicado: es una relación, es un eje de innumerables relaciones.

— Jorge Luis Borges

First and foremost, to my advisor, Sanjay Kalé – your talent and tireless focus on both pioneering research and ensuring that the fruits of that labor do not merely produce papers, but are realized in software that provides practical benefits for actual people define the epitome of what an academic should be. More importantly, thank you for the generosity you show toward all of your students, consistent willingness to always put us first, and all of the opportunities, resources, support, advice, and conversations; I have been very lucky to have you as an advisor.

To my committee – Luke Olson, David Padua, and Antonio Peña – thank you for all the time you have taken from your busy schedules to help me, and for all of the guidance, diverse perspectives, and valuable feedback.

I owe a debt of gratitude to all of my collaborators. To all of the PPL members I have had the pleasure of working with – Eric Bohm, Esteban Meneses, Osman Sarood, Abhishek Gupta, Lukasz Wesolowski, Ehsan Totoni, Yanhua Sun, Nikhil Jain, Phil Miller, Jonathan Lifflander, Harshitha Menon, Xiang Ni, Akhil Langer, Bilge Acun, Michael Robson, Eric Mikida, Sam White, Juan Galvez, Seonmyeong Bak, Kavitha Chandrasekar, Jaemin Choi, Matthias Diener, and Justin Szaday, as well as Nitin Bhat and Evan Ramos from Charmworks – and all of the other Master's students, undergraduates, staff, postdocs, visiting researchers, co-authors, and external collaborators, it has been a distinct pleasure knowing and working with you all. And of course, all of this has only been possible due to the work done by the generations of alumni that came before, all of the names I know so well from past papers and commit logs; PPL is what it is today due to the immense legacy that they have left in their wake. A special thanks to Nicole Slattengren from Sandia for helping with VT integration and runs.

Thank you to Wikipedia, all the public libraries I have frequented over the years, and all other champions of free and open access to knowledge; you have given me so much. Thank you to all of the contributors to open source software; from text editing to compiling code to plotting to typesetting, literally every aspect of this thesis depends on tools built by and for the community. To all of the teachers I have ever had, I am grateful for everything you have taught me and obliging me with answers to my incessant questions. To my high school computer science teacher, David McKain, your humor, humility, and enthusiasm truly opened the door to the world of computer science for me, and I would not have gone down this path without you.

To all of the new friends from Champaign-Urbana and all of the old friends from before, thank you for everything. It is an honor to know each and every one of you, and you have done more for me than you know.

To Golden Harbor and Sidney Dairy Barn, thank you for all of the delicious food and for playing host to so many memorable outings.

To you, the reader, thank you for your interest, and I hope you take away something valuable from this work. Please take what you can and apply it to your own problems, nothing would give me more joy.

And finally, to my family – your unwavering and boundless support has been instrumental to everything I have ever done and I cannot thank you enough.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

High performance computing (HPC) is a vital tool in modern scientific research. The ability to perform simulations of phenomena from subatomic to galactic scales and to observe in detail the impacts of varying different parameters using supercomputers has allowed science to progress along avenues where experimental or other methods are not feasible.

While supercomputers are indispensable for science, there are many challenges in using them effectively. One of the largest of these is being able to actually utilize the vast raw processing power of systems to do useful work. For many applications, a critical factor in achieving high performance is the necessity of addressing *load imbalance*, a disparity in work across processors, causing some processors to be idle while waiting for others to complete some task. By definition, when load imbalance exists, a system is not being fully utilized.

The challenge of addressing load imbalance is not one unique to the computational domain. It is innate to any coordinating team of workers and thus a core part of the field of operations research, the study of using data and analytic methods to make better, more efficient management decisions. Queueing theory and makespan minimization problems such as job shop scheduling, among others, are variants reaching toward the same fundamental goal: *optimizing the scheduling and assignment of work in systems to achieve high performance.*

However, these optimization problems are not easy to solve; in general they are NP-hard. In practice, this means that they are often addressed using approximations and heuristics, otherwise they would be intractable problems. This is particularly acute for the case of high-performance parallel applications, the area of this proposed thesis, as executing a program with tens of thousands of processors and millions of tasks is commonplace.

These issues are exacerbated by the complexities of HPC applications. Many problem domains and algorithms have innate dynamism, which result in dynamically changing computations exhibiting load imbalance. Modern multi-scale, multi-resolution techniques have exacerbated this dynamism, creating ever larger load differences and introducing several distinct phases into their algorithms. Additionally, modern systems are becoming increasingly complex and heterogeneous. The number and types of cores in a node are increasing, and this induces further load imbalance due to the different performance characteristics of different types of processors and variability in the components. Futhermore, there are often other properties that constrain a computational job, such as the memory capacity or power usage of a node and contention for shared resources, such as last-level caches or network communication. It can be very difficult to determine which of these many factors actually affect performance in practice, a set which may change depending on the application, input data, machine, network topology, etc. Because of this dynamism and complexity, the pattern of load may be continuously changing across an execution, necessitating periodic rebalancing throughout the run, so load balancing must also be fast, lest it add significant overhead to an application. Finally, the time-stepped iterative nature of many HPC applications necessitates synchronization across processors at every step, exacerbating the impact of load imbalance since every processor has to idly wait for the most heavily laden one to finish before advancing at the end of every iteration.

To this end, an emerging issue is that existing load balancing techniques do not have the sophistication to address all of the intricacies of these modern applications and systems. In particular, distilling "load" down to a single scalar value per object does not capture the richness of the performance landscape. Without insight into the individual aforementioned software and hardware considerations, load balancers often do not have enough information to adequately analyze or make decisions about the performance of an application.

One potential solution to address this issue is *vector load balancing*, in which load is no longer a single scalar value, but instead a vector of values, each representing a different aspect of the execution. Via these data, load balancing algorithms can take a holistic approach rather than a myopic one, considering load as a cohesive whole encompassing these individual measurements. Thus, much in the same way that diets are assessed not only by calories, but also fat, carbohydrates, protein, vitamins, and minerals, application performance and load balance should also be assessed by considering the individual components that comprise the overall picture.

In this dissertation, we identify and characterize several situations where the use of a scalar to represent load is fundamentally incapable of representing the true nature of the load imbalance of an application. We show that load vectors are able to faithfully capture the details of such situations and we demonstrate that vector load balancing strategies can use these vectors to produce higher quality load balancing results. We construct a solution to the issues created by the multi-dimensional nature of "load" via vector load balancing and apply it to parallel applications. This entails exploring and elaborating the multi-dimensional nature of "load" and its relationship to properties of interest, such as execution time and execution feasibility, constructing the vector measurement and storage infrastructure, a runtime system interface, new vector-based LB strategies, and finally using all of that to demonstrate benefit for applications.

CHAPTER 2: BACKGROUND

2.1 LOAD BALANCING

Load balancing in the context of HPC encompasses a wide variety of different techniques with the common aim of improving performance by balancing the workload across the available resources of the system, ranging from fine grained workstealing of parallel loop iterations to domain-specific methods for partitioning, such as orthogonal recursive bisection for particle-based codes. In this thesis, we focus on coarse-grained periodic dynamic load balancing for use with distributed-memory parallelism. By "coarse-grained", we mean that we assume the application is decomposed into objects that encapsulate the computational work and/or data of the program and we balance on the level of these objects, mapping them to the processing elements (PEs) of the system. Until the objects are remapped at the next invocation of a load balancing strategy, all of the work associated with an object executes on the PE it is mapped to. By "dynamic", we mean that load characteristics of the application may evolve over time, and by "periodic", we mean that load balancing will be invoked repeatedly across the course of the execution of the application to address the dynamically varying load. The interval between invocations is usually on the order of tens to hundreds of iterations, depending on the particulars of the application. While load balancing more frequently may allow a faster response to changes in load, there are overheads associated with load balancing. The main overheads are the cost of running the strategy to determine the new, balanced mapping itself, and the cost of migrating the objects from one PE to another to apply that new mapping, which can be quite expensive in the distributed-memory context we focus on since it requires serializing objects and transferring them over the network. Thus, load balancing too frequently can increase time to solution when these accumulated overheads exceed the benefit of rebalancing.

In the remainder of this thesis, when we refer to load balancing, we mean this particular type of load balancing unless otherwise specified.

2.2 SCALAR LOAD BALANCING

In general, the goal of load balancing is to minimize the maximum load assigned to a processor in the job. Or, mathematically, let P be the set of processors, O the set of objects, $l_o \in \mathbb{R}_{\geq 0}$ the load of object $o \in O$, and $M : O \to P$ a function mapping objects to processors.

Then, the goal of load balancing is to find a mapping M to minimize the following objective function:

$$\arg\min_{M}\left(\underbrace{\max_{p\in P}}_{\text{Maximum load on a single PE}}\left(\overbrace{\sum_{\forall o\in O:M(o)=p} l_o}^{\text{Load on PE } p}\right)\right) \tag{2.1}$$

Note that load *balancing* is semantically a bit of a misnomer in this formulation; balance or equality are not the explicit goals, instead the aim is to minimize the maximum load on any processor in the job. This is because the most overloaded processor is usually what determines overall application performance, as it will be the last to complete iterations or to arrive to barriers.

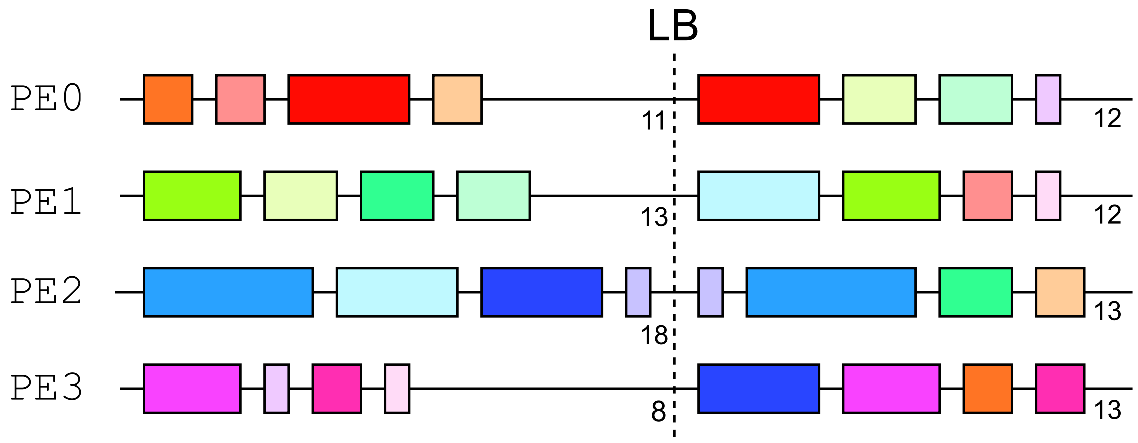


Figure 2.1: Application Timeline with Load Balancing

Figure 2.1 illustrates how load balancing can improve the performance of a parallel application. This figure depicts an execution timeline for a hypothetical application running on four PEs. PEs are arranged on the vertical axis, and the horizontal axis shows the progression of time. The boxes sitting atop the PE lines represent the execution of a task on that PE, with the color of the box indicating the ID of the object owning that task. The measured load of each PE is written under the corresponding line at the end of each iteration. In this case, the load of a PE is equal to the sum of the loads of the objects on that PE, and the load of an object is equal to the total time spent in the execution of its tasks since the invocation of the load balancer.

The figures shows two iterations of the application, and load balancing is invoked between the two, indicated by the vertical dashed line. In the first iteration, the maximally loaded

PE is PE2, with a load of 18. Visually, even though all four PEs have execute four tasks, we see that PE2 has three relatively long tasks, the first three shown on the timeline colored in shades of blue, while the other PEs have shorter tasks on average. At the tail end of the first iteration, the other PEs remain idle for a long time while they wait at the synchronization point for PE2 to finish its assigned work.

When the load balancer runs, it sees that imbalance exists in the job and it remaps the objects across the PEs to reduce the maximum load assigned to any single PE. In the second iteration, which runs after the load balancer has applied this new mapping, there are now two PEs, 2 and 3, tied as the maximally loaded PEs, each with a load of 13, only 72% of the maximum load before load balancing. We see that the three lengthy blue colored tasks that were previously all on PE2 are now spread across PE1, PE2, and PE3, leading to a more balanced distribution of load.

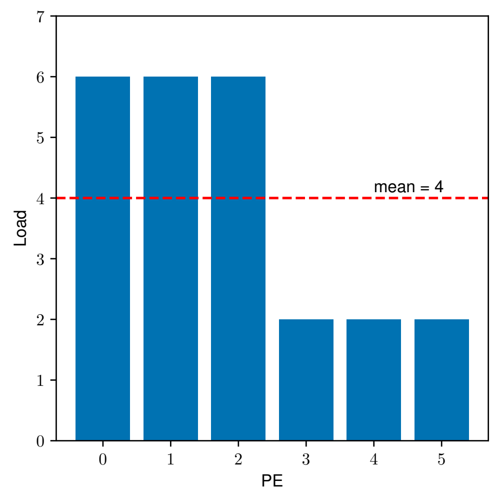
2.2.1 Assessing Load Balance Quality

Importantly, when evaluating the quality of a load balancer, it may be tempting to use variance as a metric to determine how "balanced" a given mapping is, since variance directly measures the degree of dispersion of values from the mean; so a mapping with a smaller variance should be more "balanced" than a mapping with a larger variance. However, when comparing mappings, a smaller variance does not necessarily co-occur with a smaller maximum load. Instead, we use the maximum to average load ratio ($Max : Avg$), as this correlates directly with the aforementioned optimization objective. For example, consider the two cases presented in Figure 2.2.

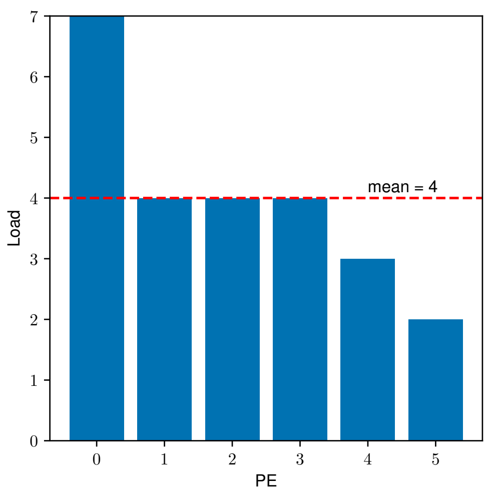
Both Mapping A (Figure 2.2a) and Mapping B (Figure 2.2b) distribute 24 units of load across 6 PEs, with a mean load of 4 units per PE. Since the maximum load on any processor in Mapping A is 6 and the maximum load in Mapping B is 7, Mapping A is the better mapping by our optimization criterion.

Mapping A has a variance of 4.8, while Mapping B has a lower variance of 2.8; thus, if we were to use variance as our quality metric, Mapping B erroneously beats Mapping A. However, by $Max : Avg$, Mapping B rightfully loses, with a ratio of 1.75 as opposed to 1.5 for Mapping A. Thus, in general, we use $Max : Avg$ as the metric to evaluate the quality of load balancing results for the remainder of this thesis.

Returning to the example shown in Figure 2.1, the $Max : Avg$ ratio decreases from $18/(50/4) = 1.44$ before load balancing to $13/(50/4) = 1.04$ after load balancing, quantifying



(a) Mapping A
$\sigma^2 = 4.8$
$Max : Avg = 1.5$



(b) Mapping B
$\sigma^2 = 2.8$
$Max : Avg = 1.75$

Figure 2.2: Comparison of Two Mappings of 24 Units of Load

the improvement in the quality of the mapping.

2.3 SCALAR LOAD BALANCING TECHNIQUES

The simplest case of load balancing is that of static problems, or problems where load does not change at all during execution. In this case, a single partitioning of the data at the beginning of execution is most often used. Since the cost of partitioning happens only once, partitioning performance and load measurement are not paramount concerns, so it differs greatly from the dynamic case.

Dynamic problems present a far more interesting problem, as the load distribution can change wildly and unpredictably across the execution of the problem.

The same partitioning strategy as used in the static case can also be used in the dynamic one, except it must now be repeated throughout the execution; so both partition quality and partition performance become important for useful load balancing. Additionally, depending on the application, both computational balance and communication locality may be important. Due to these factors, partitioners such as METIS [1] and Zoltan [2] are commonly used for load balancing parallel applications. These packages offer high performance balancing by using scalable techniques such as graph coarsening and recursive geometric bisection. Object graphs where vertices correspond to objects and edges correspond to communication or interactions between objects may also be provided to these systems, allowing them to make placement decisions that take communication into account by considering the cost of the edge cut sets. These systems can only make partitioning *decisions*; the user application is responsible for measuring load and passing it into the partitioner and actually applying the output decision by repartitioning or migrating the data of the program.

Corbalan *et al.* take the opposite approach for MPI + OpenMP applications [3]. Instead of repartitioning the data to balance load, they move computational power between MPI ranks, decreasing the number of OpenMP worker threads used on underloaded ranks and increasing the number of threads on overloaded ranks.

Another scheme for load balancing is *work stealing*. In work stealing schemes, each processor has a queue of work items. When load is imbalanced, some processors will finish all of the items in their queue before others have. When this happens, idle processors with empty queues will steal work items from the queues of other processors, preventing idle time so long as some available work remains. This scheme is used in Cilk [4] and Multilisp [5]. Since it is automatically triggered and requires no explicit calls to balance load, work stealing is very responsive and easy to use, but it can suffer from high overheads via context switches, poor locality, and contention. Further, it is typically only used for within-node load balancing due to locality considerations.

Task-based parallel programming frameworks offer another perspective on load balancing. Systems such as StarPU [6] and OmpSs [7] manage programs expressed as a set of tasks, each with some dependencies. The runtimes construct these into a task graph where edges correspond to dependencies, and tasks then get scheduled across the processors of a job as their incoming dependencies are fulfilled. Additionally, both of these systems have advanced runtimes that can analyze the properties of their tasks to do things like place tasks close to their required data and dynamically select different hardware targets based on the performance characteristics of the task.

2.4 CHARM++

The load balancing strategies in this thesis are designed to be used with Charm++ and extend its load balancing framework, unless otherwise noted. However, the ideas and strategies themselves are not dependent on Charm++ and may be used with other programming systems so long as they provide the necessary knobs and levers. In this section, we describe Charm++ and the capabilities it offers.

Charm++ [8], [9] is an adaptive, asynchronous, task-based parallel programming framework. The core entities of Charm++ are objects, called *chares*, which send messages to each other to trigger tasks on the host processor of the receiving object. Charm++ has three particular properties that enable it to perform load balancing: an active runtime system, migratable objects, and overdecomposition.

2.4.1 Active Runtime System

When a Charm++ application executes, it runs in the context of the Charm++ runtime system (RTS). The RTS runs a scheduler that decides which task to execute next, and control always returns to the RTS after a task completes. While this of course adds some overhead to execution, it also allows the RTS to introspect, make measurements, and change the configuration or layout of the application in search of higher performance. Namely, for load balancing, this allows the system to automatically measure the load of objects and to use that information to *identify* imbalance and *plan* how to correct it via load balancing strategies. The output of a load balancing strategy is a mapping of objects to processors created by solving or approximating some optimization problem.

2.4.2 Migratable Objects

Creating a new mapping that balances load is only useful if that mapping can actually be applied. Charm++ allows for the application of new mappings via migratable objects. In general, communication in Charm++ is object-centric, meaning that messages are addressed to and delivered to objects, regardless of where they are located in the system, a characteristic akin to the actor model. This is enabled by location management services in the RTS. In contrast, in other systems such as MPI, communication is processor-centric, meaning there is no way to directly address an object. Further, these Charm++ objects have serialization and deserialization routines, meaning they can be packaged up and moved between processors or nodes. Taken together, this means that when imbalance is identified, Charm++ has the means to *migrate* objects between nodes and processors to improve the situation.

2.4.3 Overdecomposition

In many parallel programming paradigms, data are decomposed into a number of chunks or objects equal to the number of processors in the job. This minimizes messaging and scheduling overhead while ensuring that every processor has some work to do. However, as addressed earlier, these chunks may not actually represent an equal amount of work, especially for dynamically evolving problems. Further, even if these chunks were migratable, only having one chunk per processor means that the size of objects is too coarse to actually address the differences in load; migrations result in either a mere permutation of the original one-to-one mapping or, by the pigeonhole principle, processors with multiple chunks while others are empty. Charm++ instead encourages overdecomposition, creating many more objects than the number of processors. This gives the runtime system scope to move objects between processors while still ensuring that all processors have work. Also, objects are usually fine grained enough that they can be distributed in a way to give the processors roughly equal load. While overdecomposition can result in higher overheads, these are often hidden by overlapping computation and communication. All in all, overdecomposition provides sufficient *granularity* to balance load between processors.

2.4.4 Load Balancing Framework

Building on these core tenets, Charm++ features a well-established load balancing framework [10]–[12]. The existing framework supports automatically instrumenting chares to measure their load and communication pattern and volume, as well as measuring intrinsic PE load. The runtime manages collecting these load statistics, passes them into strategies when they are invoked, and coordinates the migrations of objects to conform to the new mapping.

There are two main modes for invoking load balancers: periodic and `AtSync`. In the periodic mode, load balancing is called repeatedly according to a user specified period, running asynchronously with the application as it continues to execute. In `AtSync` mode, the application calls the `AtSync` function from every object, which triggers load balancing after all objects have checked in. The load balancing framework then resumes the objects after load balancing and migrations are complete. In general, most applications use `AtSync` mode since load statistics are more accurate, transient chare data does not need to be migrated as chares are at a known point in their iteration lifecycle, and performance is more predictable.

There are several different load balancing strategies, each providing different performance-quality tradeoffs, with some offering special additional considerations, such as minimizing the number of object migrations or minimizing internode communication. Applications are also able to write their own custom load balancing strategies to take specific properties of their application into account. Strategies may be written with different modes of execution, such as centralized, hierarchical, and distributed.

Load balancing in Charm++ has been used to great effect in a variety of applications, such as weather forecasting [13], crack propagation through finite element models [14], molecular dynamics [15], and cosmology [16].

CHAPTER 3: VECTOR LOAD BALANCING

In this chapter, we describe the mathematics and theory involved in the extension of the load balancing problem to vector load balancing, different possible objective functions and applications of vector load balancing, and the strategies we have developed to balance load in practice.

3.1 VECTOR LOAD BALANCING

Vector load balancing extends the load balancing problem formulated above in Equation (2.1), adding a dimension d and making the load $\overrightarrow{l_o} \in \mathbb{R}^d_{\geq 0}$ of each object into a vector of size d. $(\overrightarrow{l_o})_i$ refers to the ith component of o's load vector. The optimization problem then becomes a multi-objective optimization problem, in which the goal is to map each object to a PE while minimize the maximum PE load in each dimension simultaneously. In general, these problems do not have a single solution that minimizes each dimension at once, but instead a set of multiple solutions such that each element in the set cannot be improved in any dimension without making some other dimension worse. This set of solutions is called the Pareto frontier of the problem, and this property of the solutions where each cannot be improved without trading off another dimension is called Pareto optimality. The elements of the Pareto frontier are each "optimal" in some abstract sense, given the constraints of the problem, but the load balancer needs to select between them based on how they affect the performance of the target application.

An immediate difficulty that arises is how to compare mappings with vector loads. How do we know which of two mappings is better? In the scalar world, we are able to simply take the maximum load over all PEs and use that as the comparison metric for the mapping. In the vector world, things are more complicated; we can construct a maximum load vector analogous to the scalar version by taking the maximum load in each dimension over all PEs, but can we use this to compare mappings? The most obvious to do so is to do a componentwise comparison since this isolates comparisons to a single dimension at a time. However, while in the scalar case, this is a total order, in the vector case, it is merely a partial order, for example, two solutions from the Pareto front cannot be ordered relative to each other. This is unsuitable for our purposes, as we need some way to compare mappings in order to select one solution. In order to do this, we apply another function to the results in order to obtain a totally ordered metric. Namely, we focus on the following two objective functions:

1. Minimizing the **sum** of the maximum load on a processor in the job across each dimension. Or, mathematically:

$$\underset{M}{\arg\min} \overset{\text{Sum across dimensions}}{\overbrace{\sum_{1\le i\le d}}} \left(\underset{\text{Maximum load in dimension } i \text{ on a single PE}}{\underbrace{\max_{p\in P}}} \left(\overset{\text{Load on PE } p \text{ in dimension } i}{\overbrace{\sum_{\forall o\in O:M(o)=p} (\vec{l_o})_i}} \right)\right) \tag{3.1}$$

2. Minimizing the **maximum** of the maximum load on a PE in the job across any dimension. Or, mathematically:

$$\underset{M}{\arg\min} \overset{\text{Maximum across dimensions}}{\overbrace{\max_{1\le i\le d}}} \left(\underset{\text{Maximum load in dimension } i \text{ on a single PE}}{\underbrace{\max_{p\in P}}} \left(\overset{\text{Load on PE } p \text{ in dimension } i}{\overbrace{\sum_{\forall o\in O:M(o)=p} (\vec{l_o})_i}} \right)\right) \tag{3.2}$$

The validity of these objective functions varies depending on the underlying execution pattern of the application. We describe load balancing scenarios where Equation (3.1) accurately models the resulting performance in Chapter 4 and scenarios for Equation (3.2) in Section 5.2 of Chapter 5.

Correspondingly, the $Max : Avg$ ratio for a mapping when using Equation (3.1) is given by:

$$\frac{\overset{\text{Maximum load under } \textbf{sum} \text{ (Equation (3.1))}}{\overbrace{\sum_{1\le i\le d}\left(\max_{p\in P}\left(\sum_{\forall o\in O:M(o)=p}(\vec{l_o})_i\right)\right)}}}{\underset{\text{Average load under } \textbf{sum}}{\underbrace{\dfrac{\sum_{1\le i\le d}\left(\sum_{o\in O}(\vec{l_o})_i\right)}{|P|}}}} \tag{3.3}$$

The $Max : Avg$ ratio for a mapping when using Equation (3.2) is given by:

$$\frac{\overset{\text{Maximum load under }\textbf{max}\text{ (Equation (3.2))}}{\max_{1\le i\le d}\left(\max_{p\in P}\left(\sum_{\forall o\in O:M(o)=p}(\vec{l_o})_i\right)\right)}}{\underset{\text{Average load under }\textbf{max}}{\dfrac{\max_{1\le i\le d}\left(\sum_{o\in O}(\vec{l_o})_i\right)}{|P|}}} \tag{3.4}$$

For the sake of completeness, we generalize these functions. First, we define some preliminaries: Take $\vec{l_M}$ to be the load of a mapping M, a vector where each dimension equals the maximum load in that dimension on any single PE under the mapping M, and take $\vec{l_O}$ to be the load of the set of objects O, a vector where each dimension equals the sum of every objects's load in that dimension:

$$\vec{l_M} = \left\langle \max_{p\in P}\left(\sum_{\forall o\in O:M(o)=p}(\vec{l_o})_1\right), \dots, \max_{p\in P}\left(\sum_{\forall o\in O:M(o)=p}(\vec{l_o})_d\right)\right\rangle \tag{3.5}$$

$$\vec{l_O} = \left\langle \sum_{o\in O}(\vec{l_o})_1, \dots, \sum_{o\in O}(\vec{l_o})_d \right\rangle \tag{3.6}$$

Then, because $\vec{l_o} \in \mathbb{R}^d_{\ge 0} \Rightarrow \vec{l_M}, \vec{l_O} \in \mathbb{R}^d_{\ge 0}$, we can restate the objective functions given in Equations (3.1) and (3.2) using vector norms as:

$$\arg\min_M \left\|\vec{l_M}\right\|_1 \tag{3.7}$$

$$\arg\min_M \left\|\vec{l_M}\right\|_\infty \tag{3.8}$$

And we can similarly restate the $Max : Avg$ ratio functions given in Equations (3.3) and (3.4):

$$|P|\frac{\left\|\vec{l_M}\right\|_1}{\left\|\vec{l_O}\right\|_1} \tag{3.9}$$

$$|P|\frac{\left\|\vec{l_M}\right\|_\infty}{\left\|\vec{l_O}\right\|_\infty} \tag{3.10}$$

Finally, we fully generalize the objective function for any k-norm, $k \in \mathbb{R}_{\geq 1}$ (we use k instead of the more traditional p to avoid confusion with processors):

$$\arg\min_{M} \left\| \overrightarrow{l_M} \right\|_k \tag{3.11}$$

And the associated $Max : Avg$ ratio function:

$$|P| \frac{\left\| \overrightarrow{l_M} \right\|_k}{\left\| \overrightarrow{l_O} \right\|_k} \tag{3.12}$$

Notice that Equations (3.1) and (3.2) are each equivalent to Equation (3.11) at the extreme values of $k = 1$ and $k = \infty$, respectively, as are Equations (3.3) and (3.4) to Equation (3.12).

In some situations, there may be other optimization goals depending on the application or system, such as minimizing a function of some dimensions while constraining other dimensions to some threshold. Such situations are explored in Chapter 6.

3.1.1 Related Work

While scalar load balancing has been very widely studied, research into vector load balancing is much more nascent. What work there is spans several fields, from general mathematics to specific cases in the database community, the scientific computing community, and particularly the theory community, providing abstract algorithms or bounds for quality or runtime.

The earliest work in the area appears to be that of Hong in 1992 [17], in which an IO-bound query task and a CPU-bound query task in a database system are executed simultaneously to maximize utilization. This scheme is simple yet effective, only requiring the solution of simple linear systems to find the balance point. Since the execution engine of the query system can decide how to execute a given query, this is also very adaptable; the degree of parallelism of each task can be selected by the runtime based on the current state of the system.

Garofalakis *et al.* [18], [19] take a further step in the vein of Hong's work, using a general multi-dimensional optimization scheme to serve multimedia database queries, with a goal of serving as many requests as possible subject to some response time and hardware bandwidth constraints. Their scheme can be used with tasks of arbitrary dimension. Their proposed solution uses greedy list scheduling, ordering tasks by their maximum load component and iteratively assigning each to a feasible site such that makespan is minimized.

While these database query optimization schemes do share the same basic principles as the vector load balancing approach that is the focus of this thesis focuses on, they are not general enough, nor do they offer the performance or scalability that load balancing for HPC requires.

A theoretic definition for what they call the *vector scheduling* problem is provided by Chekuri *et al.* in [20]. Here, they provide several algorithms and tighten the known optimal bounds for approximation schemes for the problem for different dimensions d. While their algorithmic techniques and analysis are useful for the matter at hand, the time complexity of their methods is too large to be practical for load balancing parallel applications, for which improvements in balance quality must not come at the cost of excessive overhead, since load balancing is typically performed with some frequency throughout the execution of a program. Epstein *et al.* provide a more general framework for vector assignment problems in [21], developing a method that allows for balancing with even a non-monotonic objective function. There are several other theoretical works providing approximation bounds or formal algorithms for vector load balancing, particularly focused on online algorithms, such as [22]–[24], or problems of specific dimension, such as [25]. These theoretical contributions are valuable, but in this thesis we are interested in the practical applications and impact of vector load balancing.

The earliest practical applications of vector load balancing in HPC style applications appears to be heat diffusion style strategies targeted toward phase based applications described in [26] and [27]. In these works, the phase structure of several physics applications results in poor load balance when using scalar balancing techniques. To address this, they add vector support to their load diffusion implementations to compute the magnitude of each dimension of load to send to each neighbor, and then use some heuristics to approximate the multidimensional subset sum problem to decide which tasks to actually send to neighbors. This technique is effective in both papers and diffusion offers a clever, scalable solution to balancing vector loads, but the task selection techniques used are lacking in robust accuracy and neighborhood oriented LB tends to produce poor results on large machines because of the limited available scope of load information.

The load balancing problem can be phrased as a graph partitioning problem, splitting a fully disconnected graph where weighted vertices represent object loads, or $G = (V = O, E = \emptyset)$, into $|P|$ mutually exclusive subgraphs while balancing the total sum of the vertex weights of each partition. When communication costs are significant, this can be adapted to promote intra-node communication by adding edges weighted by the communication volume between two objects to the graph and partitioning with consideration of both balancing vertex load and minimizing the edge cut. As graph partitioning is a very common operation in scientific computing, there are several high-performance software packages to perform this task such as METIS [1], Scotch [28], and Zoltan [29]. Some of these tools support giving vector loads to vertices, for example, the multi-criteria graph partitioning algorithms in METIS are described in [30] and those of Scotch in [31]. These methods offer the performance and quality needed for HPC load balancing and are both a point of comparison for new load balancing algorithms and an essential component of the end-to-end LB systems implemented for this thesis.

Devine *et al.* apply *multicriteria load balancing* to geometric partitioners [32], used in cases where objects have a position in some coordinate space and partitions are disjoint regions of that space. These schemes are useful in many physical simulations because the locality of objects in “application space” is maintained in “processor space”, and nearby objects often interact with each other. In this paper, they use recursive coordinate bisection to perform the partitioning, determining the bisection point by comparing norms of the sums of objects in each partition. They also attempted to use arbitrary cost vectors to combine the load vectors of a partition into a single metric, but abandoned that in favor of using norms for the sake of performance and feasibility, as it changes the problem from optimizing a potentially non-convex function to doing so for a monotonic one. Results were promising, slightly worse quality than using a graph partitioner, but with lower runtime for the partitioning step.

3.2 VECTOR LOAD BALANCING STRATEGIES

3.2.1 Greedy

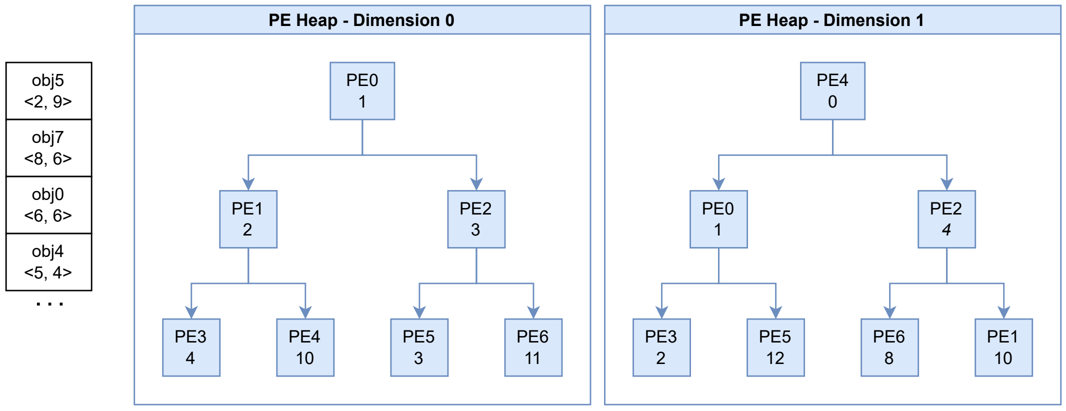
The vector greedy strategy takes the standard scalar greedy balancing strategy and adds dimension awareness. It creates d separate min-heaps, one for each dimension, and each heap contains every available PE, with heap i keyed on the ith dimension of the load vector of the PE. Objects are sorted in descending order by the maximum of their load vectors. The difference from the scalar strategy is that after the the current maximum object is popped from the object heap, the balancer determines which dimension contains its maximum load value, then places the object on the min PE corresponding to *that dimension*, and then removes and readds the selected PE from all of the PE heaps (since the entirety of the selected PE’s load vector may have changed). This requires a specially designed heap that allows for removing nodes from any position in the heap, as the selected PE is likely not at the top of the heaps for other dimensions.

The operation of the vector greedy strategy for a two-dimensional balancing problem is graphically illustrated in Figure 3.1. The figure begins *in medias res* and shows the selection and placement procedure for a single object. On the left side of each subfigure is the sorted list of objects, each element showing the object ID and load vector. The right side shows the current state of each per-dimension min-heap of PEs, each element showing the PE ID and its load in the corresponding dimension. Figure 3.1a shows the existing state of the problem, with several objects already having been placed on the PEs. In Figure 3.1b, the algorithm selects the current largest object, `obj5`, from the front of the list. In Figure 3.1c, the strategy determines the target dimension of the load vector of the selected object by comparing the load in each dimension to the mean load in that dimension across all objects; the dimension with the largest load relative to the average is the target. Here the load vector of `obj5` is $\langle 2, 9 \rangle$ and we assume that both dimensions have the same mean load, so the target dimension is one (zero-indexed). Then, in Figure 3.1d, the algorithm finds a PE for placement by selecting the PE with the minimum load in this dimension by selecting the top of the PE min-heap for that dimension, here `PE4` with a load of 0 in the target dimension. Once a PE is selected, the strategy locates each element corresponding to that PE in the PE min-heaps for all other dimensions as shown in Figure 3.1e, as placing an object on a PE requires updating every dimension of the load of a PE. Finally, the placement is performed: the object is removed from the list, the load vector of the chosen PE is updated to reflect its new load, and the PE min-heaps are updated, resulting in the final state shown in Figure 3.1f. This process repeats until the list of objects is empty, at which point each object has been placed on a PE.

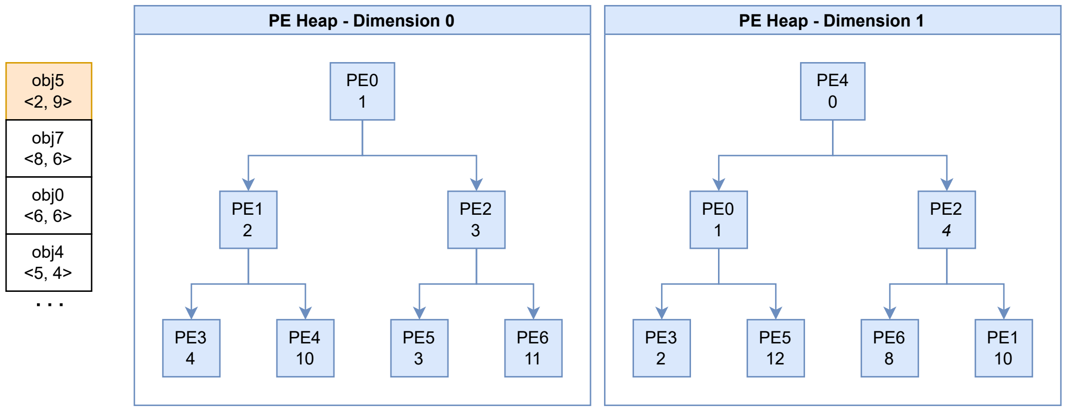
This scheme is not robust since it only considers a single dimension when making placement decisions, but it can still be effective, especially for applications comprised of objects that are each mostly active only in a single phase. For example, in a simple n-body simulation, one set of objects may be responsible for calculating forces, active for only one phase and another set of objects may apply those forces to particles, which only run in a different phase.

3.2.2 Vector Norm Balancing with k-d

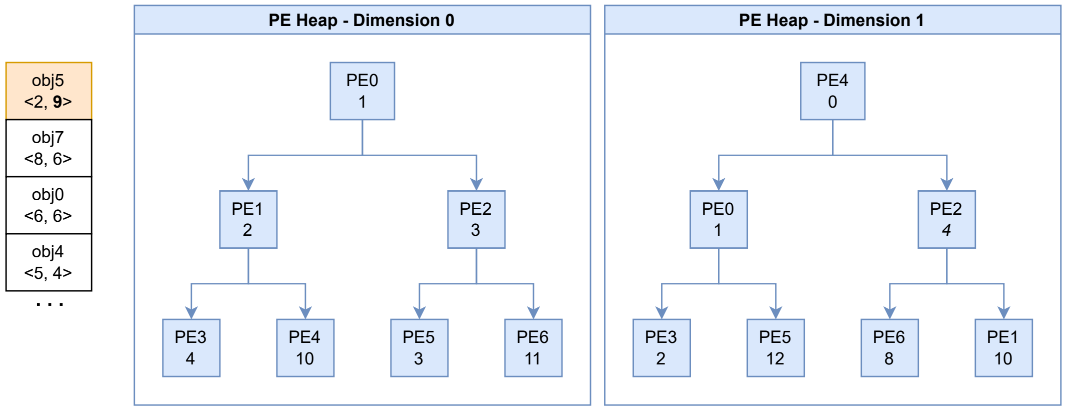
One way of reducing the complexity of the multi-objective optimization problem presented by vector loads is to convert it to a single-objective problem by minimizing the norms of vectors rather than the components of vectors themselves. This approach naturally considers



(a) Existing State

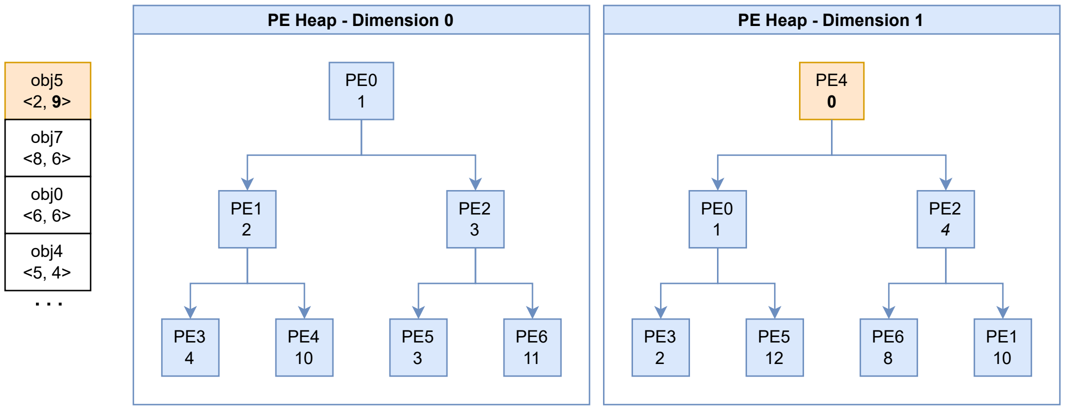


(b) Select Largest Object for Placement, Here `obj5`

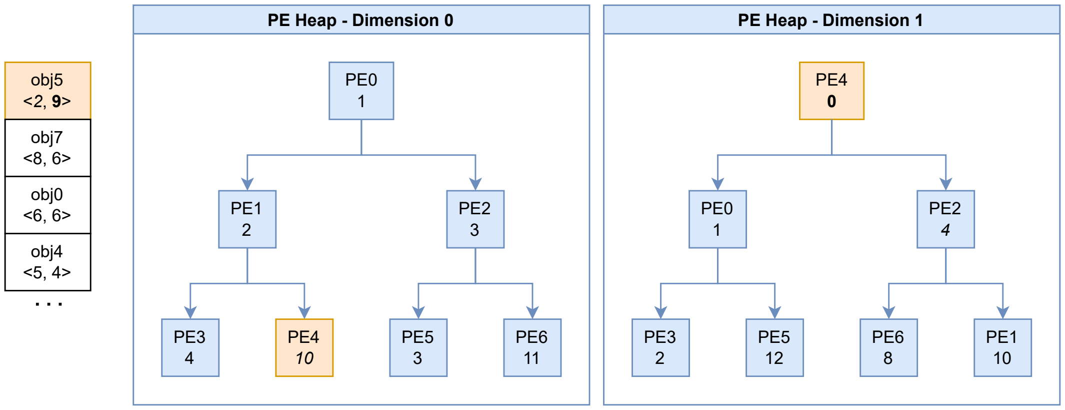


(c) Identify Target Dimension of Object, Here Dimension 1

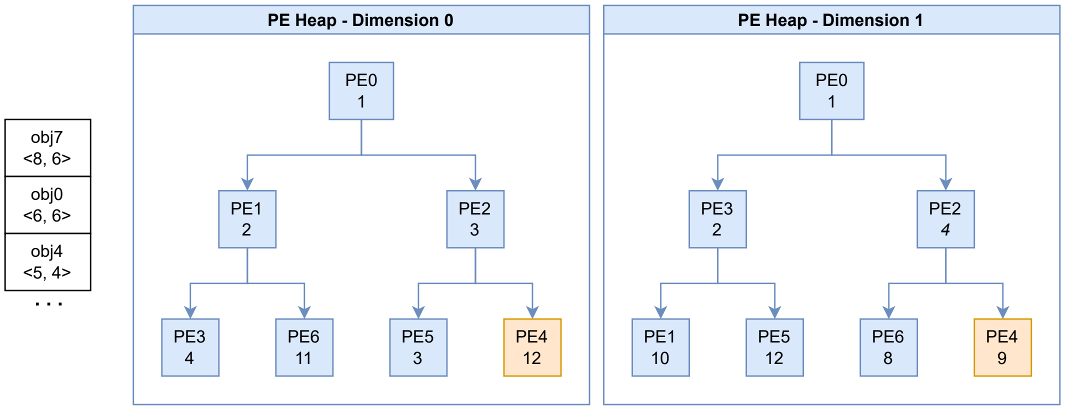
Figure 3.1: Operation of Vector Greedy Strategy



(d) Find Smallest PE in Target Dimension, Here `PE4`



(e) Select Every Dimension of Chosen PE



(f) Place Object on PE and Update State

Figure 3.1: Operation of Vector Greedy Strategy (cont.)

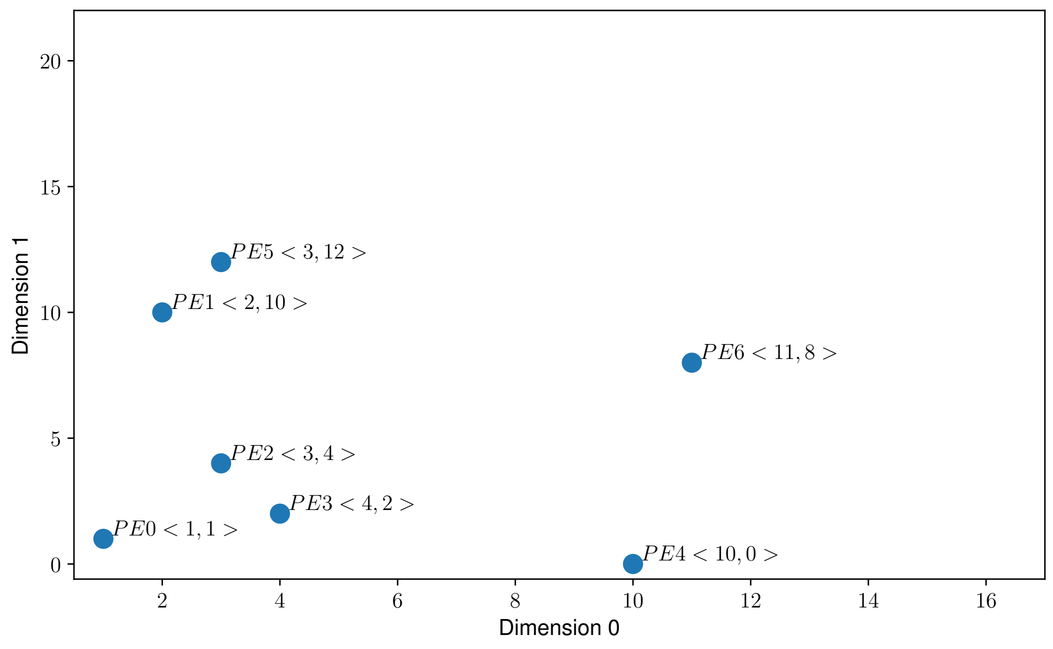
the object and processor loads holistically, using the norm as a proxy to try to ensure that no single component of the processor load grows too large.

In general, this can be done with any norm, but norm selection is very important, as different norms can produce radically different results. Using the 1-norm, which merely sums the components of the vector, is equivalent to the traditional scalar load balancing approach with no awareness of vector load. Using the ∞-norm, which takes the max of the vector, is also a poor fit for the problem since it effectively disregards the contributions of the non-maximal elements. Thus, a p-norm with $1 < p < \infty$ is more suitable for this problem. For the sake of performance, p should be an integer, as that empirically generates much faster, more vectorizable code. Studying the details and impact of norm selection is future work. Unless otherwise stated, assume that all references to norms in the following text are to the standard Euclidean 2-norm.

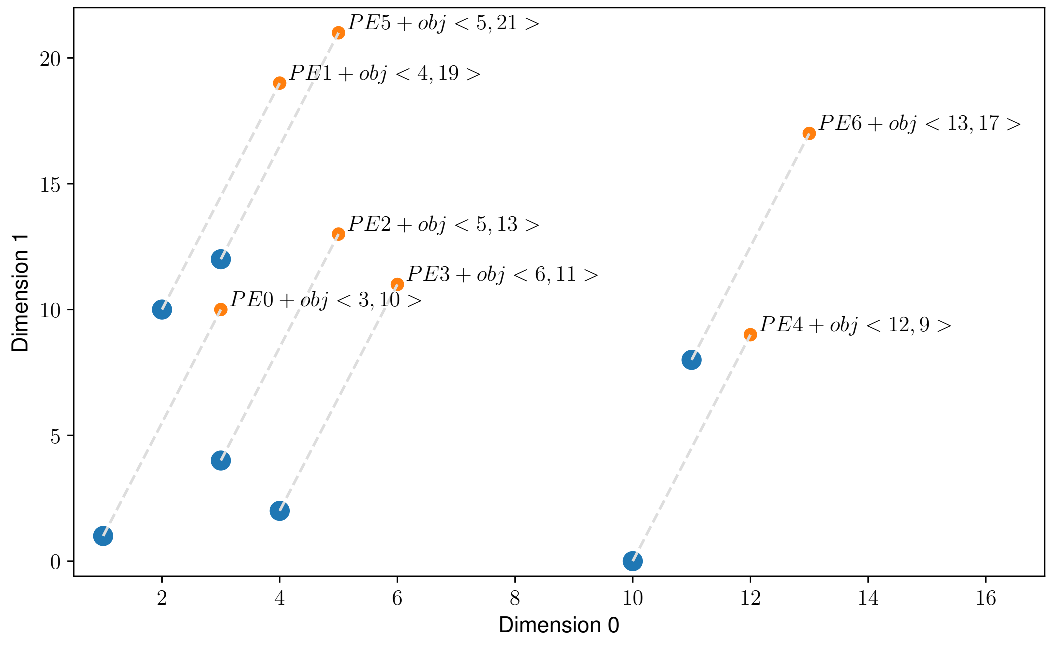
While using a norm does considerably simplify the problem, it is still difficult to assign objects to processors. This algorithm performs a greedy assignment of objects to processors, at each iteration assigning the largest remaining object to the processor with the minimum post-assignment norm. The norm provides a total order on these otherwise only partially ordered sets of vectors, but the processor with the minimum norm at any given step may not be the best choice for assignment because vector addition is not an order-preserving transformation under the norm. As an example, take $\overrightarrow{l_o} = \langle 2, 0 \rangle$ and $\overrightarrow{l_{p_1}} = \langle 3, 0 \rangle, \overrightarrow{l_{p_2}} = \langle 0, 4 \rangle$. In this case, $\|\overrightarrow{l_{p_1}}\|_2 = 3 < \|\overrightarrow{l_{p_2}}\|_2 = 4$, but $\|\overrightarrow{l_{p_1}} + \overrightarrow{l_o}\|_2 = 5 > \|\overrightarrow{l_{p_2}} + \overrightarrow{l_o}\|_2 = 2\sqrt{5} \approx 4.47$. Thus, the set of processors must be reordered relative to each object as it is considered.

Figure 3.2 shows a high level view of how the vector norm strategy performs an object placement, using the same problem data used in Figure 3.1. Figure 3.2a depicts the initial state of the objects, visualizing each PE as a point in 2-space. In Figure 3.2b, the strategy computes the projected load for each PE, shown in orange, if the current object for placement, which has load $\langle 2, 9 \rangle$, were to be placed on that PE. The norms of these projected loads are calculated in Figure 3.2c. Finally, in Figure 3.2d, the algorithm selects the PE on which to place the candidate object by finding the PE with the smallest projected load norm, here `PE0`, shown in green. Note that this result differs from that of the vector greedy strategy depicted in Figure 3.1, which selected `PE4`, because the vector greedy strategy makes placement decisions based on the single largest dimension of the object's load, whereas the use of the norm implicitly considers every dimension.

Since the PE ordering changes depending on the object, finding the minimum processor for an object is non-trivial. In the scalar case, this can easily be done by maintaining a

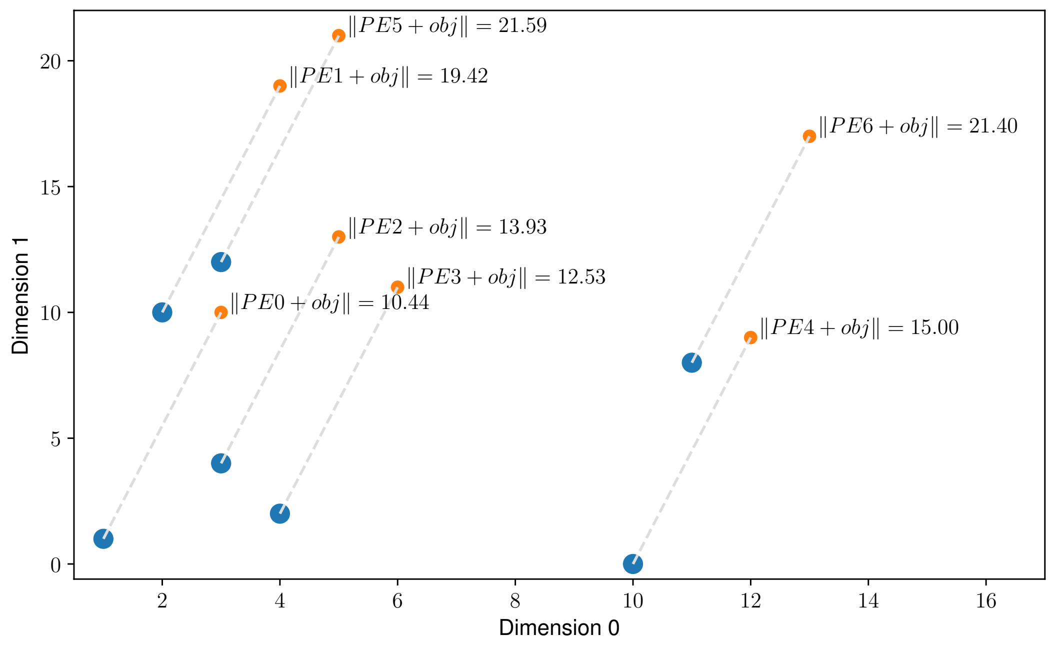


(a) Existing State

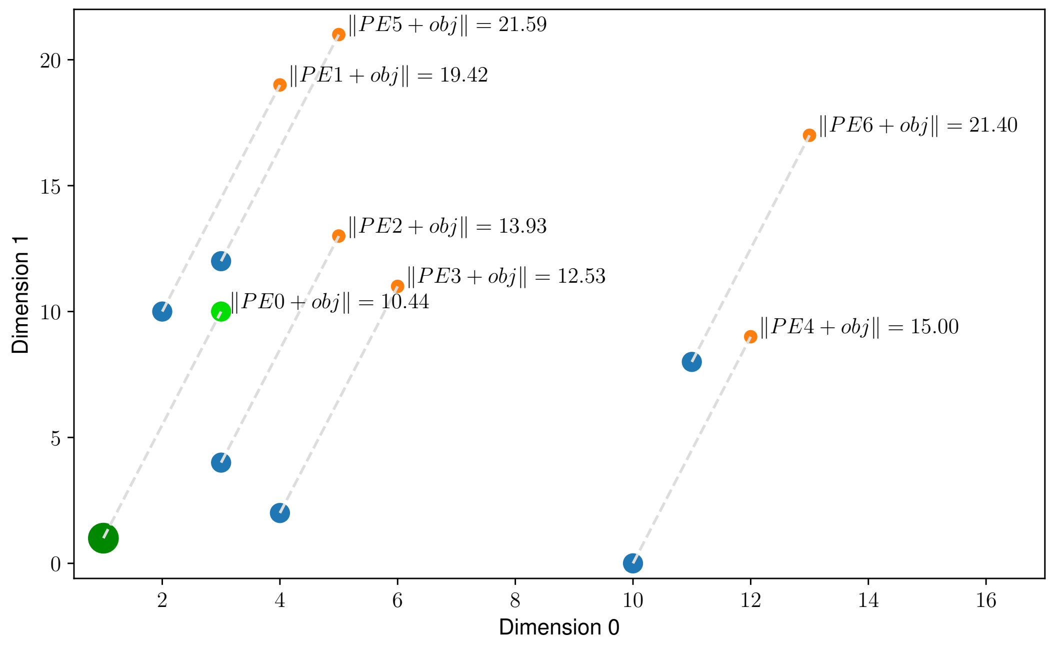


(b) Calculate Projected Loads After Adding Object with Load $\langle 2, 9 \rangle$

Figure 3.2: Operation of Vector Norm Strategy



(c) Calculate Norms of Projected Loads



(d) Select PE with Minimum Norm, Here `PE0`

Figure 3.2: Operation of Vector Norm Strategy (cont.)

min-heap for processors and a max-heap for objects and merely taking the maximum object and assigning it to the minimum processor at every iteration. In the most naïve translation of this to the world of vectors, the processor heap can be remade every iteration, at a total cost of $O(n \cdot p \log p)$ where n is the number of objects and p is the number of processors. However, this is a poor choice, as even an exhaustive linear search over all processors at every iteration is better, with time $O(n \cdot p)$. The earliest implementation of the vector norm balancing algorithm does exactly this, but while it suffices in returning the correct result, its performance is non-scalable and not suitable for anything but small problems.

To achieve more scalable performance, we can consider the d dimensional processor load vectors as points in $\mathbb{R}^d_{\geq 0}$ and use space-partitioning trees to improve query performance. Space-partitioning trees are data structures that hierarchically and recursively subdivide a space into a tree of non-overlapping regions, which can then be exploited to perform fast operations on the underlying data. In particular, we use a k-d tree [33], a space-partitioning tree for point sets of arbitrary dimension; each level of the tree splits the space in a single dimension at one of the points in the set, with the levels cycling through the different dimensions (e.g. for the three dimensional case of (x, y, z), level 0 splits along x, 1 along y, 2 along z, 3 along x, $\ldots$). Each node has a data element, *data*, which stores the split point, and two children, $left$ and $right$, which point to the "smaller" and "larger" partitions, respectively. Once this tree is constructed, finding the minimum object is a fairly straightforward traversal, as described in Algorithm 3.1.

Essentially, this algorithm traverses the space, searching "smaller" candidates before "larger" ones and identifying and pruning partitions that cannot beat the current best. In the worst case, a single iteration of this can still result in an exhaustive search of the processor space, but pruning should significantly reduce the search space for a reasonable distribution of processors, in expectation. However, there are still a few issues with this approach, namely induced imbalance in the tree and the cost of repeated insertions and deletions. At each iteration, after choosing a processor to take the current object, that processor is removed from the tree, updated with the load vector of the object, and then readded to the tree. In other words, we remove a "small" processor from the tree and readd it as a "big" processor. This can cause the tree to become imbalanced, which increases the time for search, addition, and removal. Additionally, while addition is $O(\log n)$ for balanced trees, it can still be costly in practice since additions require traversing the tree until finding a leaf, as insertions at arbitrary positions are not allowed, which also inhibits maintaining a balanced tree.

In order to address the issues with standard k-d trees, we use relaxed k-d trees [34].

Algorithm 3.1: *k*-d Strategy Algorithm

```
$tree \leftarrow MakeTree(P)$;
for $o \in sorted(O)$ do
    $p_{min} \leftarrow FindMinNormPE(tree, o)$;
    $tree \leftarrow Remove(tree, p_{min})$;
    $sol.Assign(o, p_{min})$;
    $tree \leftarrow Add(tree, p_{min})$;
end
Function FindMinNormPE(tree, o, bounds = $\langle 0, \ldots, 0 \rangle$):
    if $tree.left \neq NULL$ then
        $p_{best} \leftarrow FindMinNormPE(tree.left, o, bounds)$;
    end
    if $\|tree.data + o\| < norm_{best}$ then
        $norm_{best} \leftarrow \|tree.data + o\|$;
        $p_{best} \leftarrow tree.data$;
    end
    if $tree.right \neq NULL$ then
        $oldBound \leftarrow bounds[tree.dim]$;
        $bounds[tree.dim] \leftarrow tree.data[tree.dim]$;
        if $\|bounds + o\| < norm_{best}$ then
            $p_{best} \leftarrow FindMinNormPEEarly(tree.right, o, bounds)$;
        end
        $bounds[tree.dim] \leftarrow oldBound$;
    end
    return $p_{best}$
```

Relaxed k-d trees differ from their standard brethren in that consecutive levels no longer regularly cycle through splitting different dimensions, instead, each node stores an arbitrary discriminant $j \in \{1, 2, \ldots, k\}$, which indicates the dimension it splits. In this data structure, children may even split along the same dimension as their parent. This relaxation allows greater flexibility when performing updates to the tree, namely that nodes can be inserted at arbitrary positions at any level, not just as leaf nodes. Coupled with random level and discriminant selection, we can exploit this flexibility to maintain a probabilistically balanced tree while performing updates at every iteration. Empirically, this results in much better algorithmic performance than exhaustive search or the standard k-d tree. We analyze the performance and scalabilty of load balancing strategies in more detail in Chapter 7.

3.2.3 METIS

METIS, a graph partitioner, offers another way of making vector load balancing decisions [30]. In their scheme, a graph is partitioned into some number of partitions such that the load vector of the partition is less than or equal to some tolerance in each dimension while minimizing the sum of the weights of the edges cut. This is accomplished by performing multi-objective bipartitioning, which recursively splits the graph into two subgraphs (METIS also offers a direct k-way partitioning method [35], but both methods provide roughly the same results in terms of performance and quality, so we only describe one method for the sake of simplicity). Partitions are kept balanced by searching through d priority queues, each corresponding to one dimension of the load vector. Each vertex is only in one of these priority queues, the one corresponding to the dimension of its largest load. After partitioning, the partitions are refined to minimize the edge cut using vector extensions of standard algorithms such as Kernighan-Lin [36] or Fiduccia-Mattheyses [37]. Additionally, for the sake of performance, the graph is coarsening by collapsing pairs of connected vertices. This reduces the problem size and, by selecting heavy edges to collapse, also guarantees that these edges stay in one partition. As the algorithm proceeds, the graph is hierarchically uncoarsened, being fully uncoarsened at the leaves.

While METIS is usually used for communication-aware load balancing, it can also be used for general load balancing by trivially viewing the objects as a fully disconnected graph. This does not take communication into account at all, but still provides balanced partitions of objects with vector loads.

3.3 VECTOR LOAD BALANCING IN CHARM++

As a prerequisite to developing, testing, and evaluating vector load balancing strategies, first, support for vector load balancing had to be added to the load balancing framework inside Charm++. This required adding the ability to measure vector loads, store and pass vector loads inside the RTS, and to accept load balancing strategies that can ingest and utilize vector loads.

C++ template specialization is used to support distinguishing vector versions of a strategy from the scalar version of a strategy. Scalar and vector versions of the same strategy may differ in terms of the applicability of certain data structures or particular optimizations (the same can apply to vector variants of different dimensions, which this scheme also supports).

A complete example of the implementation of scalar and vector versions of the greedy load balancer in Charm++ is shown in Listing 3.1.

```cpp
// Vector version
template <typename O, typename P, typename S>
class Greedy : public Strategy<O, P, S> {
public:
  void solve(std::vector<O>& objs, std::vector<P>& procs, S& solution, bool objsSorted)
    {
    // Sorts in descending order of the maximum value in an object's load vector
    if (!objsSorted) std::sort(objs.begin(), objs.end(), CmpLoadGreater<O>());

    // Create one heap per dimension (this special heap type supports arbitrary removal)
    std::vector<ProcHeap<P>> heaps;
    for (int i = 0; i < O::dimension; i++) {
      heaps.push_back(ProcHeap<P>(procs, i));
    }

    // Find the average load vector of an object
    std::array<float, O::dimension> averageLoad;

    for (const auto& o : objs) {
      for (int i = 0; i < O::dimension; i++) {
        averageLoad[i] += o.load[i];
      }
    }
    for (auto& load : averageLoad) {
      load /= objs.size();
    }

    // Going through the objects in descending order of load...
    for (const auto& o : objs) {
      int maxdimension = 0;
      float maxfactor = 0;
      // Find largest dimension relative to the average load
      for (int i = 0; i < O::dimension; i++) {
        if (o.load[i] / averageLoad[i] > maxfactor) {
          maxfactor = o.load[i] / averageLoad[i];
          maxdimension = i;
        }
      }
      // Assign object to the PE with smallest load in that dimension
      P p = heaps[maxdimension].top();
      solution.assign(o, p);
      for (auto& heap : heaps) {
        heap.remove(p);
        heap.push(p);
      }
    }
  }
};

// Scalar version
template <typename P, typename S>
class Greedy<Obj<1>, P, S> : public Strategy<Obj<1>, P, S> {
public:
  void solve(std::vector<Obj<1>>& objs, std::vector<P>& procs, S& solution,
             bool objsSorted) {
    if (!objsSorted) std::sort(objs.begin(), objs.end(), CmpLoadGreater<Obj<1>>());

    // Only one heap needed for one dimension
    std::priority_queue<P, std::vector<P>, CmpLoadGreater<P>> procHeap(
        CmpLoadGreater<P>(), procs);

    // Going through the objects in descending order of load...
    for (const auto& o : objs) {
      // Assign to the PE with smallest load
      P p = procHeap.top();
      procHeap.pop();
      solution.assign(o, p);
      procHeap.push(p);
    }
  }
};
```

Listing 3.1: Vector and Scalar Greedy Load Balancer

CHAPTER 4: LOAD BALANCING FOR PHASE-BASED APPLICATIONS

In this chapter, we describe the motivations for and properties of phase-based applications, the fundamental deficiencies of current load balancing techniques for these applications, and the improvements provided by applying the new vector based load balancing techniques developed in this thesis.

4.1 PHASE-BASED APPLICATIONS

4.1.1 Overview

As high performance computing has grown in adoption throughout the scientific and engineering community, so has the diversity of applications. Further, as the capabilities of machines, libraries, and other software building blocks have improved, so has the intricacy and sophistication of programs, enabling ever faster, larger, and more detailed simulations.

Many factors have contributed to these improvements in diversity and utility. Applications may be composed of several modular pieces, combining the results of each in order to solve a more complex problem than any individual piece is able to on its own. Similarly, scientists may take an existing application or framework and build a novel extension atop it in order to use it for their own field. Additionally, researchers have developed optimized adaptive algorithms that focus their time on the "interesting" parts of the problem space, providing great speedups and bringing previously infeasible problem sizes into the realm of possibility.

One consequence of these advances in applicability and performance is increased complexity in the execution structure of programs, namely the rise of phase-based applications (PBAs, often also called multi-phase applications).

In a phase-based algorithm, each iteration is broken down into several different collaborative components, each specializing in performing a particular part of the larger computation. These individual phases can have arbitrary data domains and dependencies, but a common configuration is one wherein the results of certain phases are used as the input for subsequent phases. The communication domain for phases may also vary; some phases may perform no communication, others may do so only within some local neighborhood, and others may do global reductions, all-to-alls, or other communication across the entire parallel job.

Because of these dependencies, the execution structure of the phases is essentially sequential, with the next phase unable to start until the previous phase has completed. For example, in a molecular dynamics application, the computation of all the various forces, such as bonded forces, electrostatics, and van der Waals forces, must be completed first before applying them to the atoms, yielding two phases, force calculation and force integration.

Note that we define phases of a computation to be completely *orthogonal*, meaning there is no overlap between the execution of one phase and any other phase across the entire system. Many applications exhibit this structure for a variety of reasons, as we discuss in Section 4.1.2. However, some applications may have "blurry" phase boundaries, such as cases where an object only depends on a subset of the other objects and can proceed to the next phase as soon as it has received updates from those dependencies. The techniques discussed in this chapter should generally still work for such cases, as we are solving a stricter version of the problem than required by those cases.

4.1.2 Causes

Phase-based applications are widespread in practice. Partially, this is because of innate qualities of the problem domain or target datasets:

Multiple Variables: Physical problems of interest to researchers often involve studying the combined effects and evolution of several different simultaneous physical processes, e.g. analyzing the change in and interplay between temperature, pressure, wind speed, humidity, and precipitation in weather prediction software. Applications for such work often perform a separate set of calculations for each property of the problem, and are hence called *multiphysics* simulations. [38] Multiphysics simulations have an inherent phase structure, as the calculations for simulating each separate physical process are generally performed in separate phases, and, on top of that, additional phases are needed to integrate all of the disparate results and update the state of the simulation.

Multiscale Data: The data used in scientific problems may span across several orders of magnitude due to underlying physical properties, such as the difference in pressure near a blast wave versus in the remainder of the fluid for shock hydrodynamics applications. In these situations, it may not be necessary to update every component in the simulation domain with the same frequency or precision; for example, regions of the domain near the fast-moving wave need to be updated at high detail and high frequency to capture the nuances of propagation and evolution of the wavefront, whereas regions far from the wavefront can be updated relatively infrequently without sacrificing precision beyond the desired tolerance, as they undergo only very minute changes until the wave reaches them. This strategy of skipping uninteresting or relatively slowly evolving parts of the domain during some timesteps is part of the field of *multiscale* simulation. This optimization can make previously intractable simulations tractable and is the core idea behind techniques such as adaptive mesh refinement (AMR) and multi-timestepping. In these applications, an iteration is subdivided into phases corresponding to multiple smaller "timesteps," each only updating the pertinent subset of the domain, e.g. a structure such as AAB, with two "A" timesteps that update only fine parts of the dataset, and then a "B" timestep that updates both fine and coarse parts of the dataset.

In other cases, a phase structure arises due to design or implementation choices:

Specialization: In particle-based simulations, such as astrophysical or molecular dynamics, forces exerted between particles are one of the main quantities to be calculated at every timestep. Naïvely calculating these pairwise forces takes $O(n^2)$ time. However, by instead classifying these pairwise interactions into separate sets of short-range and long-range pairs based on their distance and processing them separately, techniques such as Barnes-Hut simulation [39] or Ewald summation [40] [41] reduce this to $O(n \log n)$ time. This greatly improves performance and can also introduce a phase structure to the code depending on the implementation.

Note that this is different from the multiphysics case because we are only concerned with the forces of a single physical process and it is also different from the multiscale case because we still update the state of the entire simulation at every timestep.

Libraries: Decomposing a complex algorithm into individual phases consisting of its simpler constituent components or reframing a computation into a different paradigm to enable the use of high-performance external libraries may improve application performance may yield a phase-based structure. For example, a direct implementation of a graph algorithm may be outperformed by a three phase implementation, a phase to convert graph data into a matrix, a phase performing some matrix computation using an optimized BLAS library, and a final phase to convert data back into the original graph format.

Locality: Completing all of one type of computation before moving onto the next can improve spatial locality, leading to increased cache performance and vectorizability with single-instruction multiple-data (SIMD) instructions. However, this can lead to tradeoffs with regards to temporal locality, as the same data may be repeatedly accessed from within different phases rather than reading it once and doing the whole end-to-end computation with it while it is in a register or cache.

Simplicity and Programmability: Some applications might not strictly require barriers to synchronize between separate components of an application for the sake of correctness, but developers may intentionally decide to insert global synchronization regardless. This is usually done to prevent race conditions in cases where checking for the actual necessary dependencies is too difficult to program or computationally expensive. Such situations arise in irregular applications, wherein objects may not be aware how many messages they need to receive before advancing to the next part of the computation, or in applications with dynamic object creation or deletion, in which dependencies may change from iteration to iteration. Further, developers may use synchronization to provide more predictable performance or to avoid overloading memory or the network.

Furthermore, these qualities may co-occur in an application, multiphysics simulations are often used with datasets amenable to multiscale execution, and calling into specialized libraries for parts of their computation is common.

4.2 LOAD BALANCING PHASE-BASED APPLICATIONS

While phase-based applications can provide many benefits as explored above, they also increase the difficulty and complexity of achieving performance and scalability.

In particular, PBAs present a challenge for load balancing because the distribution of load becomes *spatiotemporal* within an iteration, not merely spatial as it is for non-phase-based applications. Specifically, describing load by *where* it occurs is insufficient to fully characterize the execution pattern; it now needs to be described by both where *and when* it occurs.

4.2.1 Limitations of Scalar Load Balancing

Figures 4.1 and 4.2 illustrate this challenge by comparing the behavior and observed load of a non-phase-based application and a phase-based application when using scalar load balancing. Note that spacing has been added for clarity in both figures.

Figure 4.1 traces a hypothetical run of a non-phase-based application on four PEs, arranged vertically, and shows the progression of time horizontally. Each colored box represents the execution of a task on the PE corresponding to the line it sits atop. Each task belongs to an object as indicated by the color of the task. The measured load for each PE is shown to the right of the last task; in this case, each PE has a measured load of 16, indicating a perfectly balanced workload with a $Max : Avg$ ratio of 1. Recall that to calculate the load

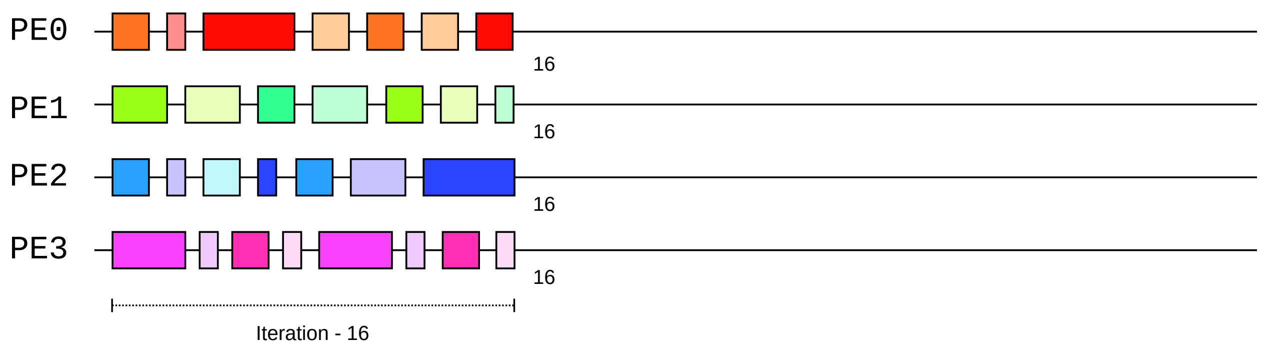


Figure 4.1: Non-Phase-Based Application Timeline

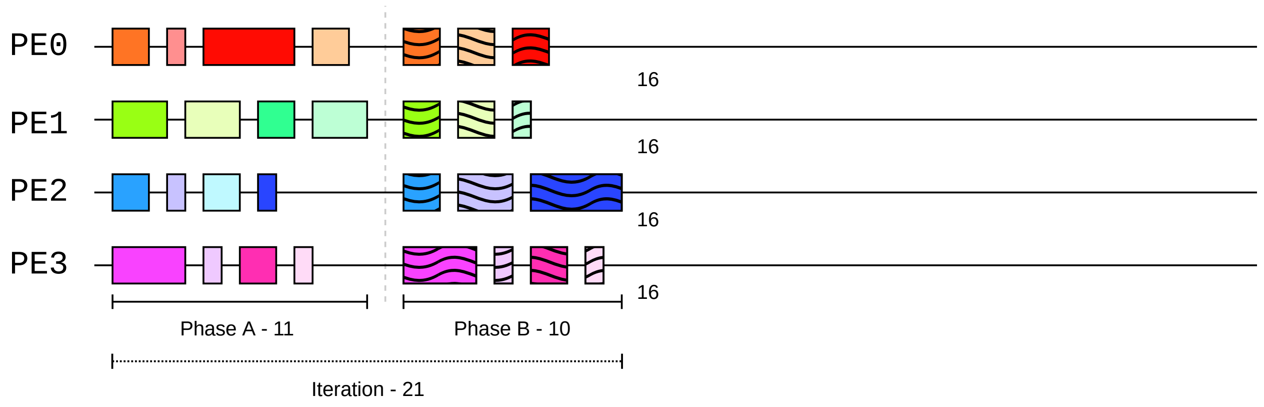


Figure 4.2: Phase-Based Application Timeline

of a PE, we merely sum the loads of its resident objects, and the load of an object is the sum of the execution time of each of its tasks. Finally, the total length of the iteration is 16, calculated by determining how long it takes for all tasks to be completed. Importantly, this corresponds to the maximum observed PE load (in practice, this would not be the case unless all communication were overlapped with computation on the heaviest loaded PE, but in general this is true to some epsilon).

Figure 4.2 shows an execution timeline of a phase-based application consisting of iterations split into two phases, A and B. Tasks executing in phase A have a solid fill and tasks executing in phase B have a wavy fill. Again, the color of an task indicates its object, but note that some objects may only have tasks that execute in one phase of the iteration. A gray vertical dashed line indicates the globally synchronized phase boundary; all phase A tasks must be complete on all PEs before any PE begins any phase B task.

Just as described for Figure 4.1, the observed load for each PE is shown to the right of the last task, and again, each PE is measured to have a load of 16.

This is an accurate measurement, as each PE does indeed spend 16 units of time executing tasks for its objects. However, as the phase and iteration time measurements at the bottom of the figure indicate, the total length of an iteration is 21, made by summing the lengths of the two constituent non-overlapping phases A and B, which have lengths of 11 and 10, respectively. The length of each phase is calculated in an analogous way to how the length of the iteration was calculated in the non-phase-based case, determined by how long it takes for all tasks belonging to *that phase* to be completed.

Here we see the fundamental problem with using scalar load measurement with phase-based applications. Phases impose restrictions on when certain tasks can be executed, but scalar load balancing, by combining load measurements from every phase into a single scalar value, cannot capture these restrictions. Phase-based load is not fungible, yet scalar LB treats it as though it is.

This issue of storing multi-dimensional data as a single dimension is the culprit behind the dissolution of the relationship between scalar load and iteration time for phase-based applications. From an information theory perspective, scalar load does not encode enough information to accurately represent the structure of a PBA.

4.2.2 Benefits of Vector Load Balancing

A solution to the issue of losing the phase provenance of measured load is to use vector load measurement and balancing rather than scalar measurement and balancing. Each phase is assigned a separate dedicated dimension of the load vector, which is used to store the measured loads for that phase. Then, when performing load balancing, vector-aware load balancing strategies can utilize this per-phase data to more effectively balance the load by virtue of having awareness of how potential object migrations affect each phase of execution.

This benefit is illustrated by considering how these different load balancing paradigms would balance the application from Figure 4.2.

Figure 4.3 shows an execution timeline of a phase-based application with scalar load measurement and balancing, while Figure 4.4 shows an execution timeline of the same PBA with vector load measurement and balancing instead.

In Figure 4.3, the scalar load data indicates that every PE has the same amount of load, 16 units in this example. Thus, from the perspective of the LB strategy, the existing mapping is already perfectly balanced, with a $Max : Avg$ ratio of 1, and so should not be altered by migrating objects. Correspondingly, we see in the figure that the locations of the objects are unchanged after load balancing executes. However, as explained in Section 4.2.1, the

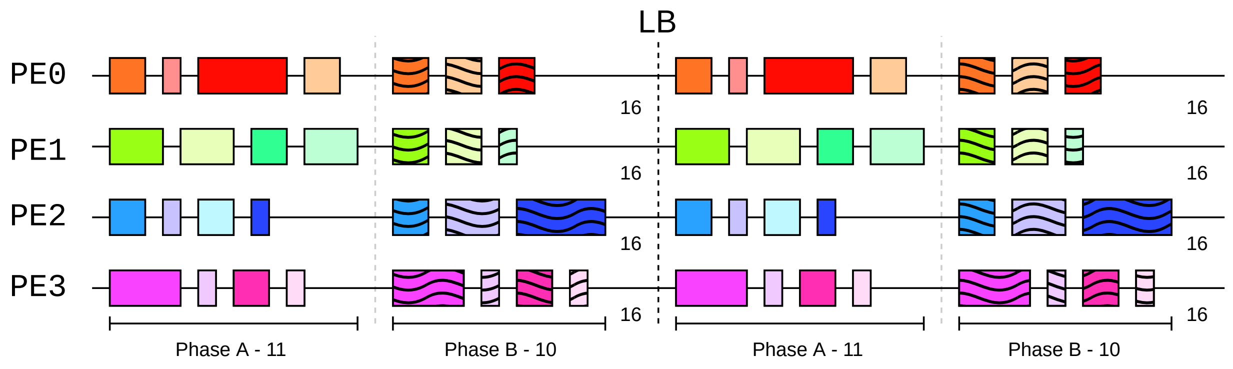


Figure 4.3: Phase-Based Application Timeline With Scalar LB

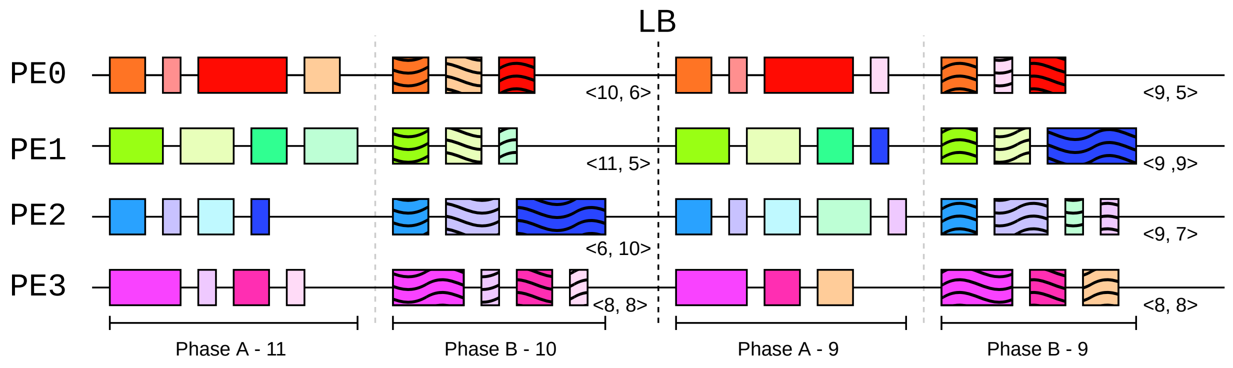


Figure 4.4: Phase-Based Application Timeline With Phase Aware Vector LB

scalar load given to the balancer does not accurately reflect the true load distribution of the application. It is impossible for the balancer to identify that the application is imbalanced at the phase level, and even if it were explicitly told that the data it is given are from an imbalanced execution, it does not have enough information to know how to address that imbalance.

By contrast, in Figure 4.4, the use of vector load measurement captures the differences in per-phase load between the different PEs. Rather than the single scalar load value shown for each PE in Figure 4.3, each PE in Figure 4.4 has a load vector; the pre-LB load for PE 0 is 16 in the scalar case, and $\langle 10, 6 \rangle$ in the vector case, for instance. The sum of the pre-LB load vector for each PE is 16, the same as the observed load in the scalar case, as the amount of time spent executing tasks does not change. However, by categorizing that load into its constituent phases, vector load measurement gives the load balancer awareness of the phase structure of the application, which the scalar case does not provide. We see that the application has imbalance in both phases A and B; Phase A loads range from a low of 6 on PE 2 to a high of 11 on PE 1, and Phase B from 5 on PE 1 to 10 on PE 2. Note that the maximally and minimally loaded PEs in the two phases are the same two PEs in opposite order, a particularly egregious case of misrepresentation in the scalar case that is corrected by using vector loads.

By virtue of using this more detailed and faithful representation of load, vector load balancers are able to identify and ameliorate phase-level imbalance, as shown in the post-LB portions of Figures 4.3 and 4.4. As explained earlier, scalar LB does not identify any load imbalance and accordingly does not migrate any objects, leaving the load distribution unchanged. On the other hand, vector load balancing is able to recognize the imbalance and improve the load distribution relative to the original mapping, reducing the per-phase times for A and B from 11 and 10 to 9 and 9, respectively, and the total execution time from 21 to 18, a 1.17x speedup. Notably, measured in terms of scalar load, the new mapping produced by vector LB is worse than the original mapping, raising the maximum load on a PE from 16 to 18 (PE 1's load of $\langle 9, 9 \rangle$), again demonstrating the incongruity of using scalar load for PBAs.

4.3 RELATED WORK

Zheng proposes a rudimentary phase-based load balancer where each phase is balanced separately [12]. This works well for objects with tasks that are active either in only one or in all of the phases (i.e. they have no restrictions on their execution), but it cannot cope well with more complex execution patterns. The proposed balancer does not use vector balancing, merely multiple passes of scalar load balancing, one for each phase and one for non-phase linked load.

Much of the work discussed in Section 3.1.1 is intended for PBAs, as well.

4.4 SYNTHETIC EVALUATION

We use our load balancing simulator to evaluate the quality of the mappings produced by different load balancing strategies for a variety of load vector configurations representing phase-based applications. We provide a load configuration file to the program which specifies the number of phases for each object and a statistical distribution for each phase, which each object independently samples from in order to generate the load of each of its phases.

The available statistical distributions for generating the load for each phase are:

1. **Constant**: Assigns a given constant value to the phase.

2. **Linear**: Given a $base$, $increment$, and $shift$, assigns to the phase a linearly increasing load $l = base + index_{eff} * increment$, where $index_{eff} = (index_{obj} - shift) \bmod num_{objs}$.

3. **Normal**: Assigns a sample taken from a normal distribution with given $mean$ and $stddev$ to each object.

4. **Exponential**: Assigns a sample taken from an exponential distribution with given $lambda$.

5. **Nested Block**: Given a $ratio$ array and an array of $distributions$, partitions the objects into contiguous blocks sized according to $ratio$ and assigns to each block the load from the corresponding entry in $distributions$. For example, if $ratio = [2, 1]$ and $distributions = [(constant : 10), (constant : 20)]$, the first two-thirds of the objects will be assigned a load of 10 and the final third will be assigned a load of 20 for the phase.

6. **Nested Probability**: Given a $ratio$ array and an array of $distributions$, for each object, selects a distribution with probability corresponding to $ratio/sum(ratio)$ and assigns the load from that distribution. For example, if $ratio = [2, 1]$ and $distributions = [(constant : 10), (constant : 20)]$, in expectation two-thirds of the objects will be assigned a load of 10 and one-third will be assigned a load of 20 for the phase.

4.4.1 Configuration

The quality of selected load balancing strategies for simulated phase-based applications with normally distributed load vectors of dimension two, four, and six is shown in Figure 4.5. Each dimension of the load vector for every object is sampled from a normal distribution with $\mu = 10, \sigma = 3$.

This experiment is repeated for simulated phase-based applications with alternating exponentially and normally distributed load vectors of dimension two, four, and six, with results shown in Figure 4.6. Each exponential dimension of the load vector (i.e. first dimension in the 2D case, first and third in the 4D case, and first, third, and fifth in the 6D case) is sampled from an exponential distribution with $\lambda = 0.15$ and each normal dimension of the load vector is sampled from a normal distribution with $\mu = 10, \sigma = 3$.

Finally, the simulations are repeated again for a more complex three dimensional load problem, with results in Figure 4.7. In this case, the first dimension of each object has an 80% chance of being sampled from a normal distribution with $\mu = 1, \sigma = 0.1$ and a 20%

chance of being sampled from a normal distribution with $\mu = 5, \sigma = 0.1$. The second and third dimensions are sampled from an exponential distribution with $\lambda = 0.1$.

Since these simulations represent phase-based applications, mappings are evaluated using the sum evaluation function given in Equations (3.1) and (3.3). Results shown are the minimum, maximum, and median across one hundred runs with different RNG seeds.

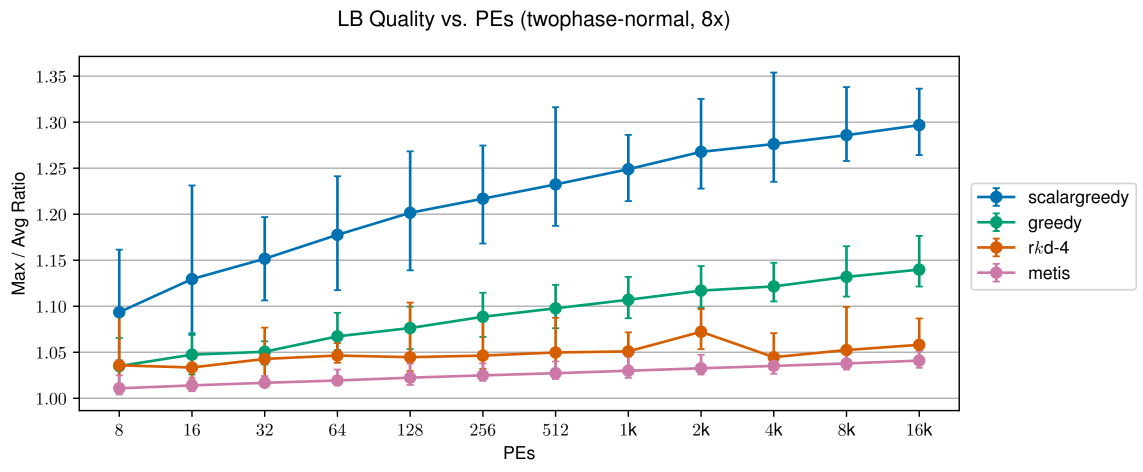
4.4.2 Analysis

With limited exception, the vector aware load balancing strategies provided better mappings than the scalar greedy strategy for most of the load configurations. Further, the more holistic rk-d and METIS strategies generally resulted in better balance quality than the greedy strategy. The rk-d strategy performs the best overall, giving either the best or very competitive mappings for all tested load configurations across the full range of PE counts. METIS essentially ties rk-d for the normally distributed and complex three dimensional cases, but is uncompetitive with rk-d for the (exponentially, normally) distributed case.

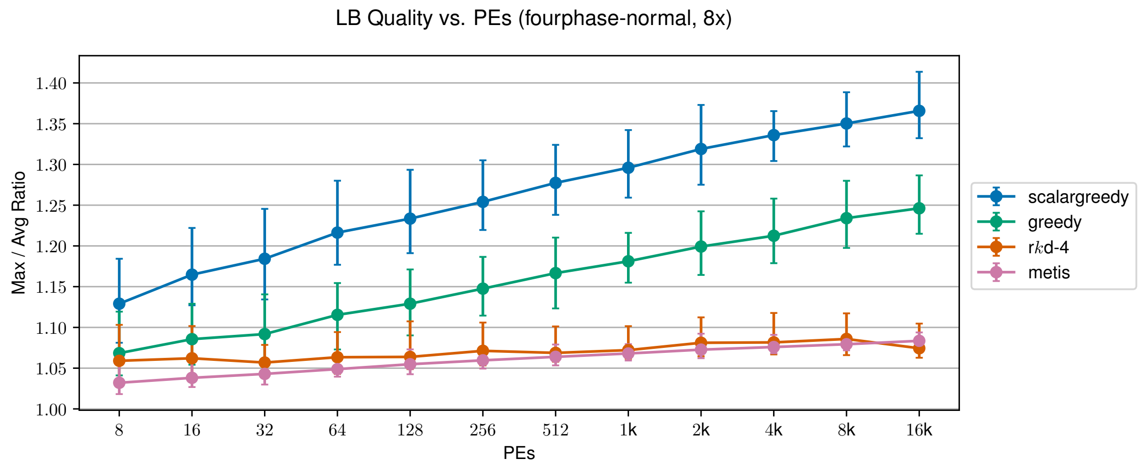
The results feature two notable anomalies, however:

1. METIS provides relatively poor quality mappings for the two phase (exponentially, normally) distributed case at 4k, 8k, and 16k PEs, losing out to even the scalar greedy strategy. It is not entirely clear why METIS delivers bad results here, but it is an isolated enough occurrence that it is likely not a concern in practice.

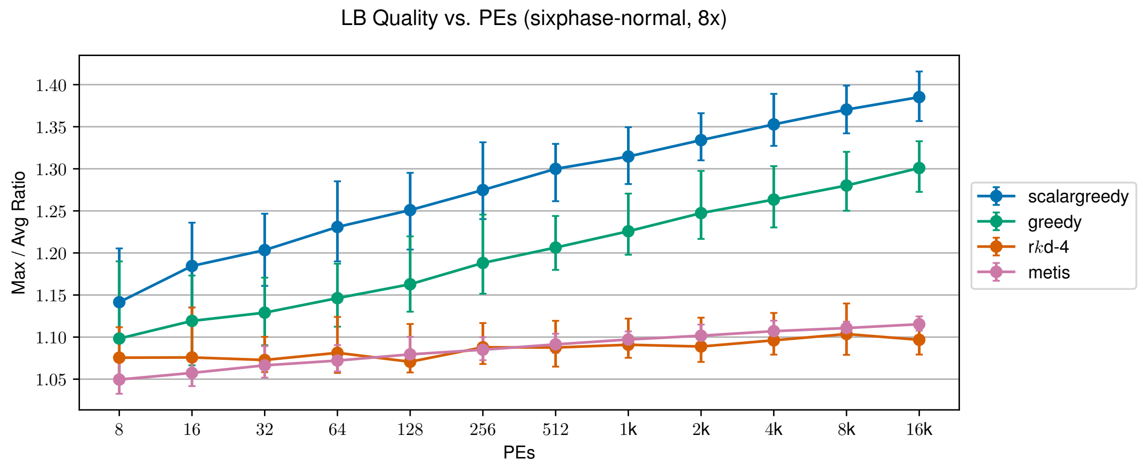
2. The greedy strategy performs on par with or worse than scalar greedy across the board for the three dimensional case. This likely occurs because the greedy strategy searches for a placement according only to a single dimension, the dimension of the object's load vector that is largest relative to the average load in that dimension over all objects. For this configuration, in the 20% of cases where the first dimension of an object's load vector happens to be sampled from the larger normal distribution with $\mu = 5$, then it is expected to be 2.78x larger than the average load (expected average load for the first dimension is $1 \cdot 0.8 + 5 \cdot 0.2 = 1.8$). In this case, for one of the exponential dimensions to be used by the greedy strategy instead, it needs to be $>$ 2.78x the mean of $1/\lambda = 1/0.1 = 10$ in expectation. Using the CDF of the exponential distribution, this only occurs with probability $P(X > 10 \cdot 2.78 = 27.8) = e^{0.1 \cdot 27.8} = 0.06$; and so the probability that one of the two exponential dimensions will be used by the greedy strategy instead of the first dimension in this case is only $1 - P(X <= 27.8)^2 = 0.12$. Thus, the 20% of objects with a large first dimension are likely to be placed without regard to their potentially



(a) Two Dimensional

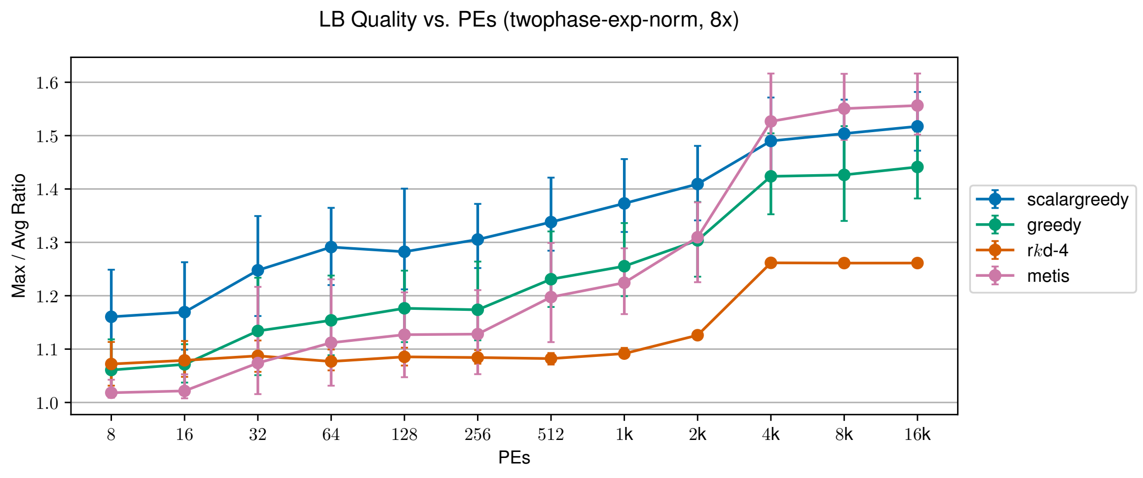


(b) Four Dimensional

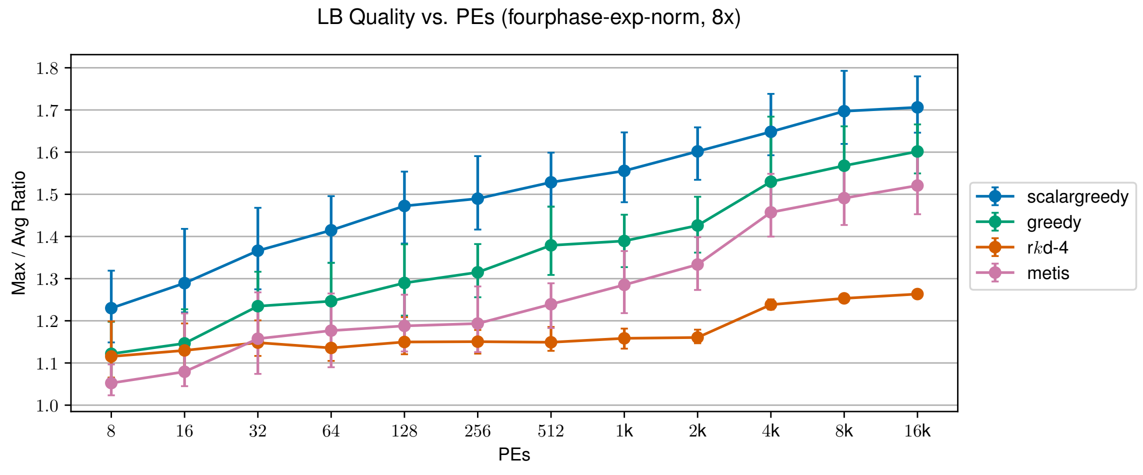


(c) Six Dimensional

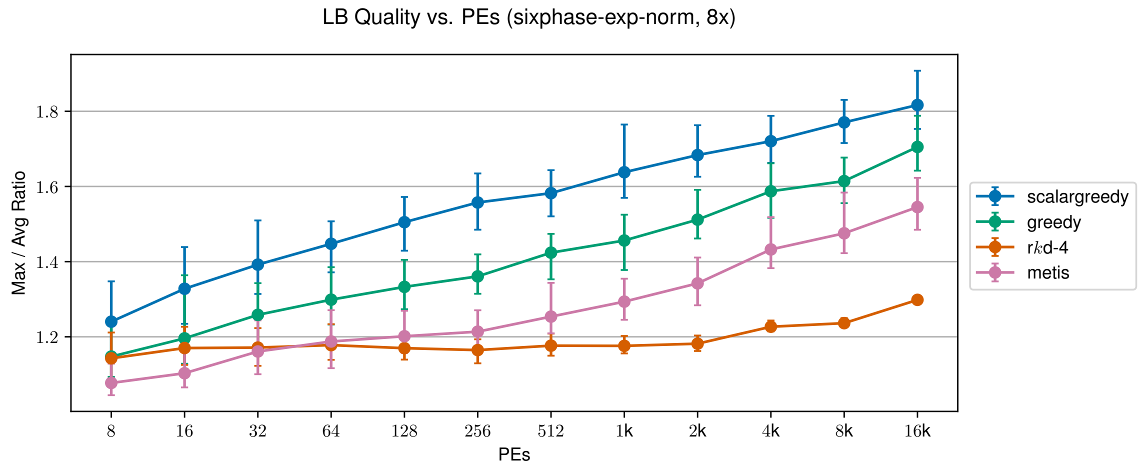
Figure 4.5: Quality Comparison with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional



(b) Four Dimensional



(c) Six Dimensional

Figure 4.6: LB Quality Comparison with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

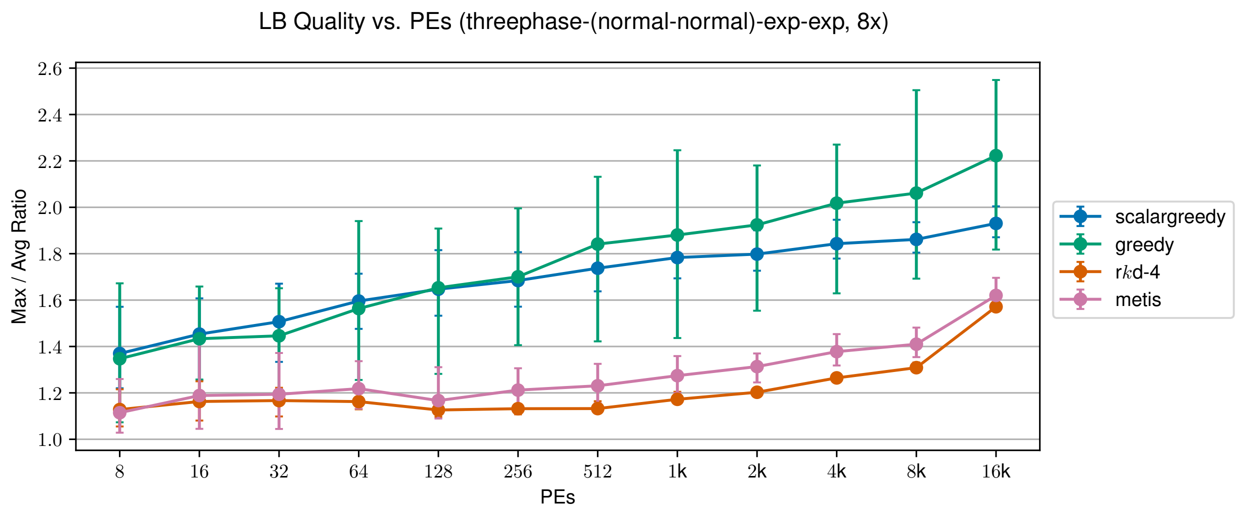


Figure 4.7: LB Quality Comparison with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

large exponentially distributed dimensions. Another consequence of this behavior is the large observed range in quality for the mappings of the greedy strategy.

The scalar greedy strategy avoids this worst case behavior since it uses a sum of all of the dimensions of the load vector for balancing, meaning it will likely avoid colocating objects that are have large loads in multiple dimensions as their sum will be large.

4.5 PRACTICAL EVALUATION

4.5.1 Stencil

We evaluate the efficacy of our vector load balancing algorithms in practice on the KNL partition of Stampede2[1] with the `mpi-linux-x86_64` build of Charm++ via a phase-based 3D 7-point stencil mini-app. The computation pattern of the mini-app consists of a series of user customizable phases, selected from the set of distributions described in the previous section. Every object samples from the specified distribution assigned to it for that particular phase and performs work proportional to the resulting value. At the end of each phase, the application sends message to each of its six neighbors if configured to do so.

For the first experiment, two phases of load are configured, the first phase an exponential distribution with $\lambda = 0.15$ and the second a normal distribution with $\mu = 10.0, \sigma = 3.0$, the

[1]https://www.tacc.utexas.edu/systems/stampede2

same configuration as the simulations shown in Figure 4.6a. Two strong scaling studies are given below, a 10x10x10 grid with 1,000 objects from 64 to 512 PEs is shown in Figure 4.8, and a 20x20x20 grid with 8,000 objects from 64 to 4096 PEs is shown in Figure 4.9. Timings are the average of five runs. For both studies, the maximum tested PE count is approximately the end of scaling, as there are only about two objects per PE on average at that scale.

All load balancing strategies outperform the case without load balancing, meaning the cost of statistics collection, running the load balancing strategy, and migrating objects is less than the benefit provided by improved load in practice (note that the simulations in Section 4.4 did not take these costs into account, only measuring the resulting quality of the mappings produced by LB). The two vector aware strategies outperform the scalar strategy, and rk-d outperforms greedy at scale. Minimum, average, and maximum speedups with the use of load balancing over the no LB case are shown in Table 4.1.

Size	**Strategy**	**Speedup**		
		Minimum	*Average*	*Maximum*
10x10x10	*Scalar Greedy*	1.03	1.14	1.28
	Greedy	1.09	1.23	1.29
	rk-d	1.12	1.28	1.42
20x20x20	*Scalar Greedy*	0.96	1.02	1.22
	Greedy	1.01	1.14	1.28
	rk-d	1.05	1.25	1.46

Table 4.1: Speedup Over Baseline with Load Balancing for Two Phase (Exponentially, Normally) Distributed Problem

The second experiment uses a three phase problem, configured in the same way as the simulations shown in Figure 4.7, the first dimension coming from a normal distribution, with an 80% chance of using $\mu = 1, \sigma = 0.1$ and a 20% chance of $\mu = 5, \sigma = 0.1$, and the second and third dimensions sampled from an exponential distribution with $\lambda = 0.1$. As in the first experiment, 10x10x10 runs up to 512 PEs and 20x20x20 runs up to 4,096 PEs are shown in Figures 4.10 and 4.11, respectively.

Validating the results of the simulations in Figure 4.7, the greedy strategy performs relatively poor here as compared to the quality of the mappings it provided for the two dimensional case, in which it always outperformed the scalar greedy strategy. The reasons behind this degradation in quality are discussed in Section 4.4.2.

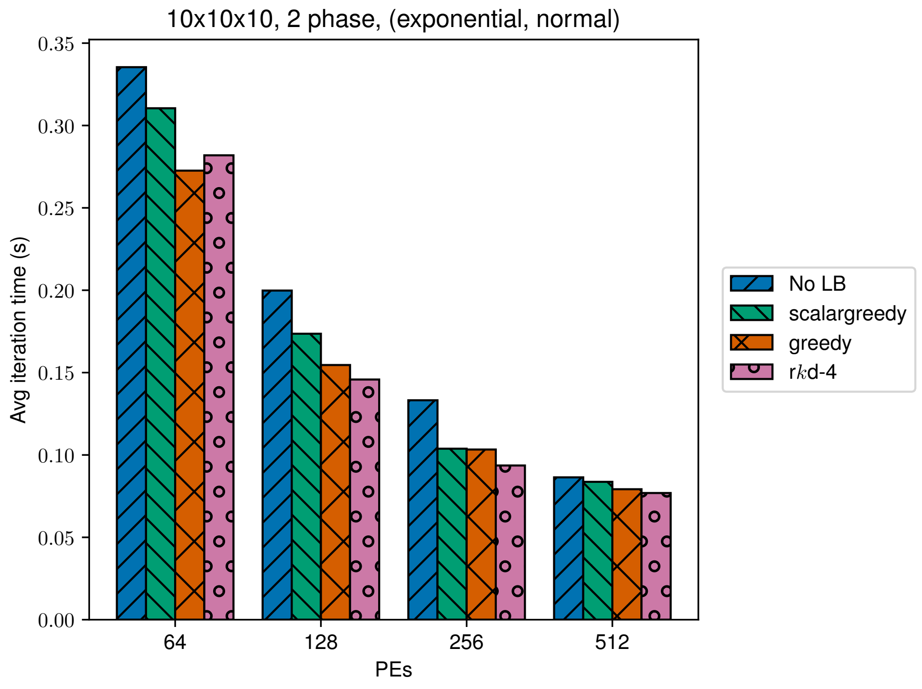
While greedy is outperformed by scalar greedy in some cases for this three phase problem, rk-d outperforms the other load balancers in every case, illustrating the benefit of using a norm-based decision strategy as the load vector gains additional dimensions and a more complex distribution. The 20x20x20 grid at 64 PEs has negative or only marginal benefits with load balancing; there are 125 objects per PE in this case, and because the load of each object comes from the same underlying distribution, there are enough objects per PE that the per-PE loads are likely close enough to the mean that the load balancing will not help much. Note that this may not be the case if there were a structured pattern to the load distribution, e.g. for some dimension, objects with low indices being more likely to have small loads than objects with high indices. Minimum, average, and maximum speedups with the use of load balancing over the no LB case are shown in Table 4.2.

Size	**Strategy**	**Speedup**		
		Minimum	*Average*	*Maximum*
10x10x10	*Scalar Greedy*	1.23	1.29	1.37
	Greedy	1.08	1.20	1.50
	rk-d	1.33	1.56	1.76
20x20x20	*Scalar Greedy*	0.95	1.26	1.72
	Greedy	0.95	1.27	1.43
	rk-d	1.01	1.51	2.12

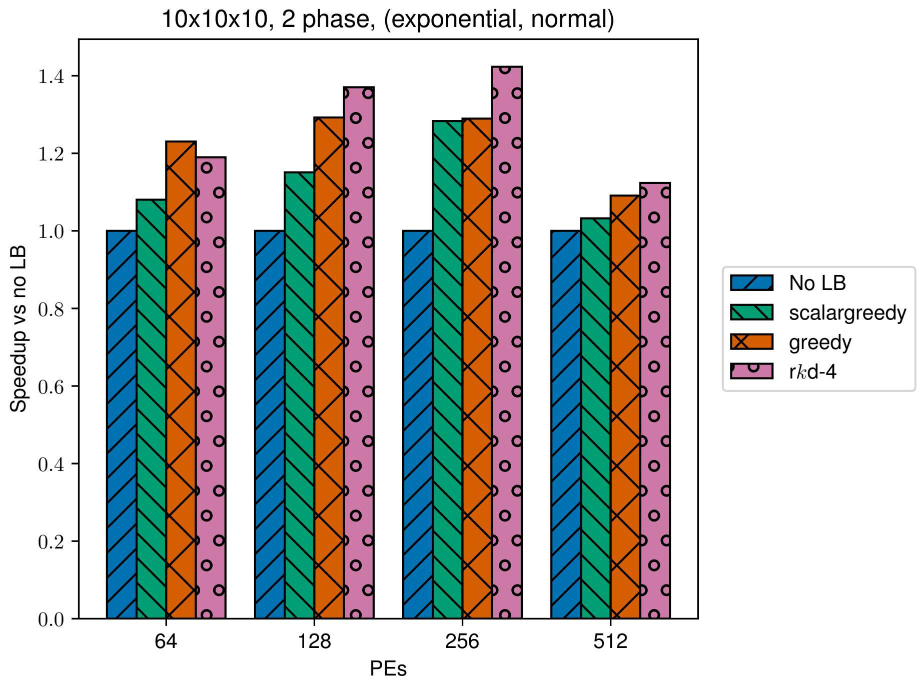
Table 4.2: Speedup with Load Balancing for Three Phase (Normally/Normally, Exponentially, Exponentially) Distributed Problem

4.5.2 Adaptive MPI

Adaptive MPI (AMPI) [42], [43] is an implementation of the Message Passing Interface (MPI) [44] built atop Charm++. The MPI standard defines a parallel programming model wherein an application is executed by several concurrent workers called *ranks*, which communicate data to each other via passing *messages*. MPI is the most widely used parallel programming paradigm in the HPC community. Most MPI implementations implement ranks via creating an operating system level process for each rank, but AMPI distinguishes itself by instead using user-level threads (ULTs) for ranks. This brings several benefits, such as improved performance when the number of ranks is greater than the number of cores, communication-computation overlap and latency hiding for blocking calls via fast context switches, and, most importantly for our purposes, migratability, enabling the use of load balancing. Typically, load balancing is difficult to apply to MPI programs, as the standard does not offer any facilities for data repartitioning, location management, or migrating ranks, so developers must manually add these features to their application. By contrast, AMPI uses the load measurement and load balancing features included in Charm++ along with a special

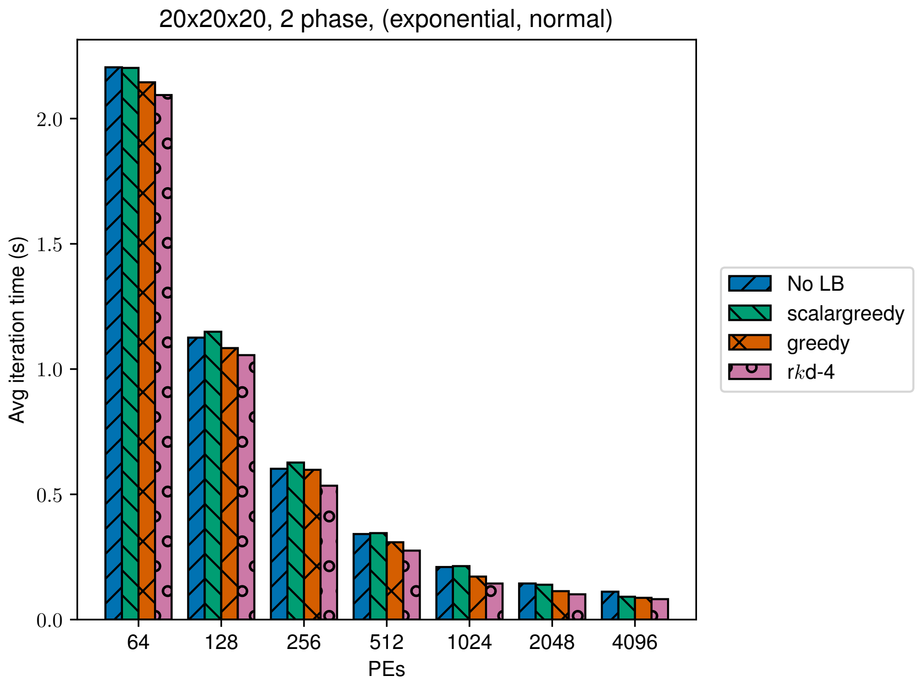


(a) Average Iteration Time

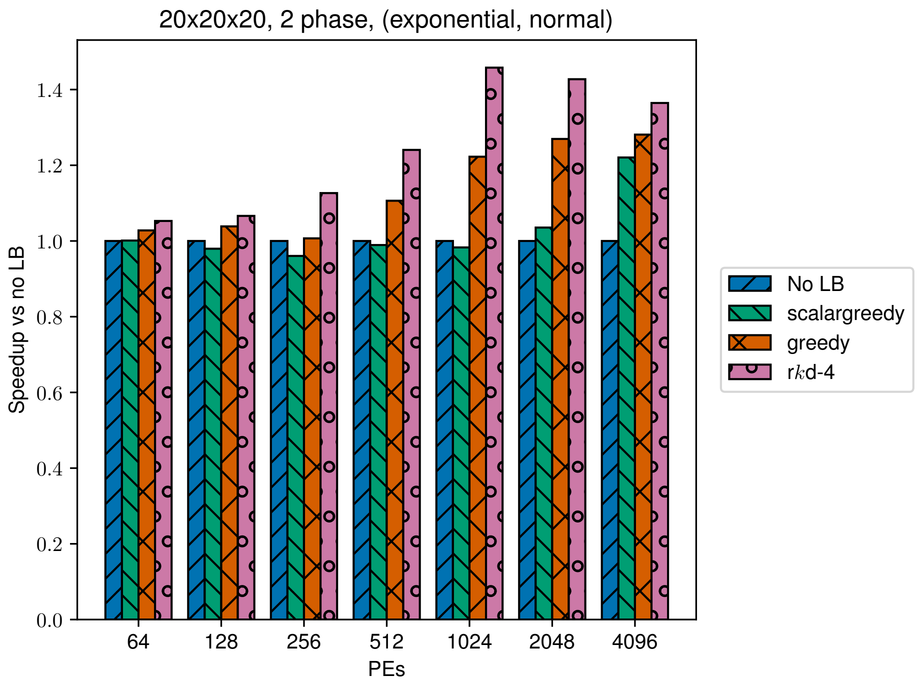


(b) Speedup vs. No Load Balancing

Figure 4.8: 10x10x10 (Exponentially, Normally) Distributed Performance

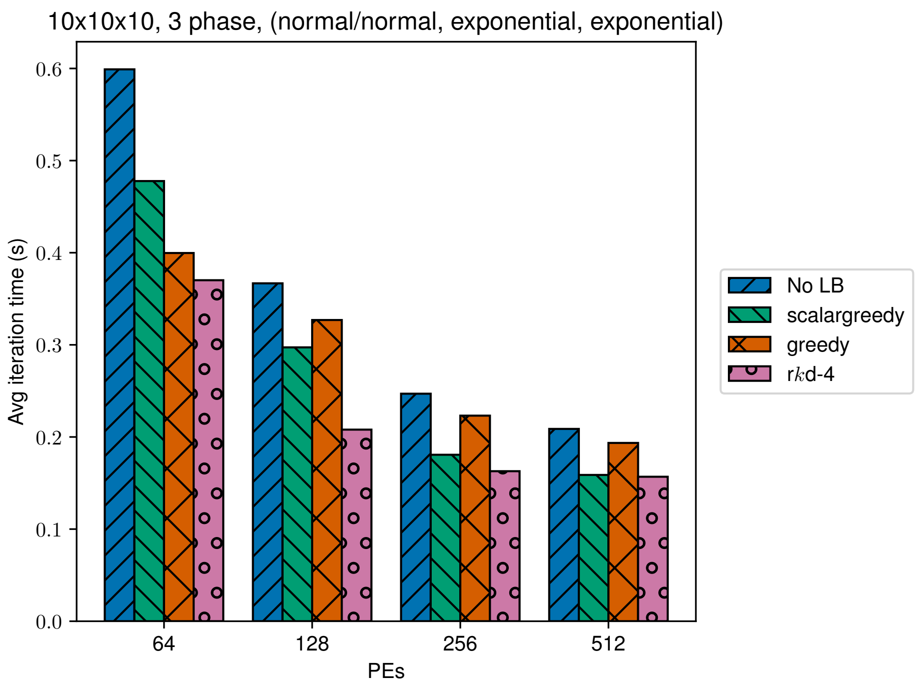


(a) Average Iteration Time

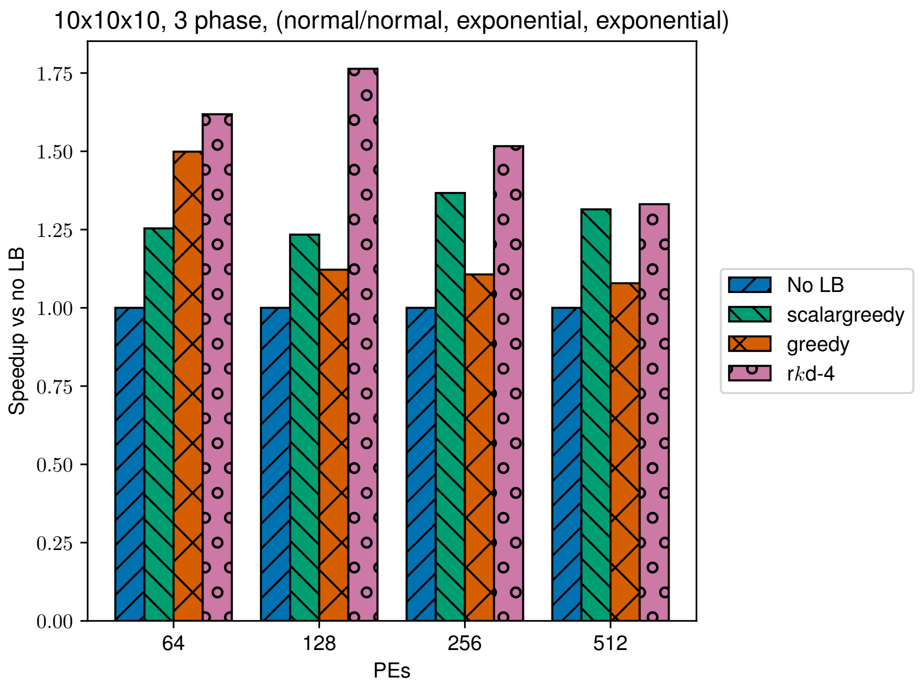


(b) Speedup vs. No Load Balancing

Figure 4.9: 20x20x20 (Exponentially, Normally) Distributed Performance

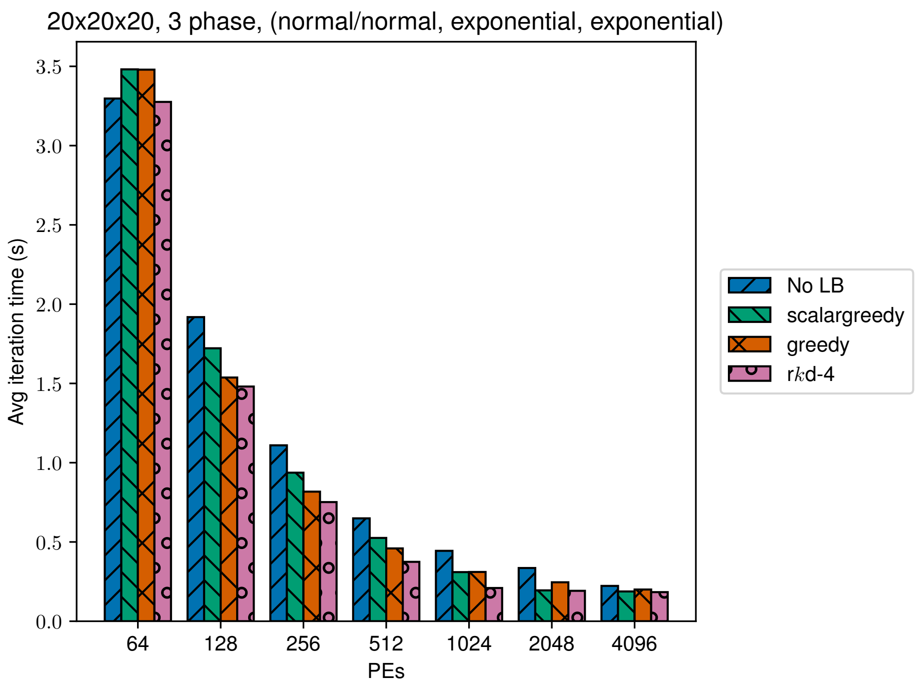


(a) Average Iteration Time

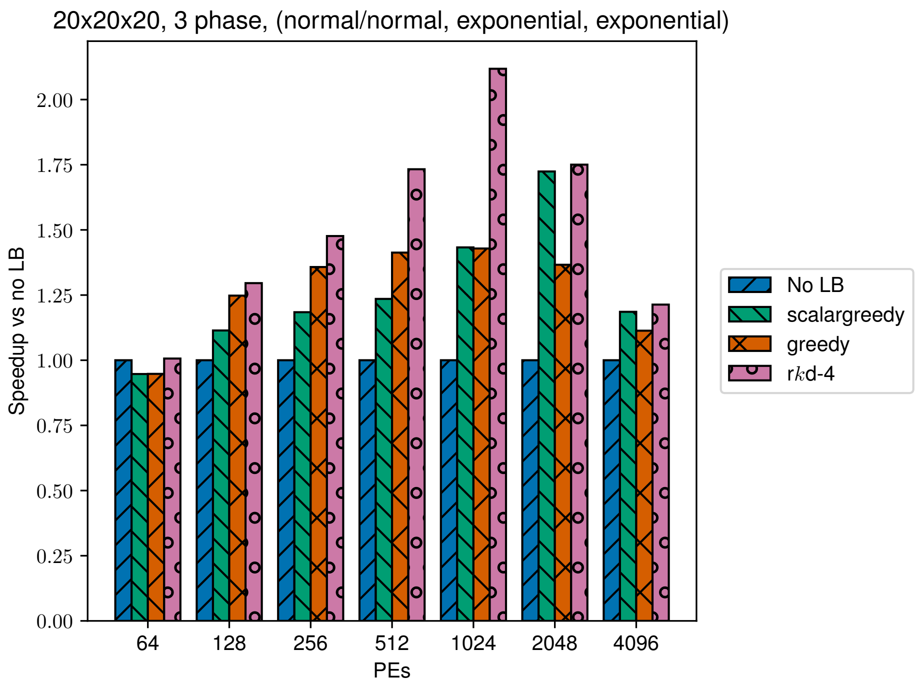


(b) Speedup vs. No Load Balancing

Figure 4.10: 10x10x10 (Normally/Normally, Exponentially, Exponentially) Distributed Performance



(a) Average Iteration Time



(b) Speedup vs. No Load Balancing

Figure 4.11: 20x20x20 (Normally/Normally, Exponentially, Exponentially) Distributed Performance

memory allocator called Isomalloc that automates serialization to allow load balancing to be used with MPI applications with minimal changes.

Because AMPI can transparently make use of Charm++ features, our vector load balancing strategies also work with AMPI. We add a function `void AMPI_Load_set_phase(int phase)` to the AMPI API which allows MPI applications to inform the RTS when a new phase begins so that measured load is attributed to the correct dimension of the load vector, but this minor addition is all that is needed for MPI applications to use vector LB.

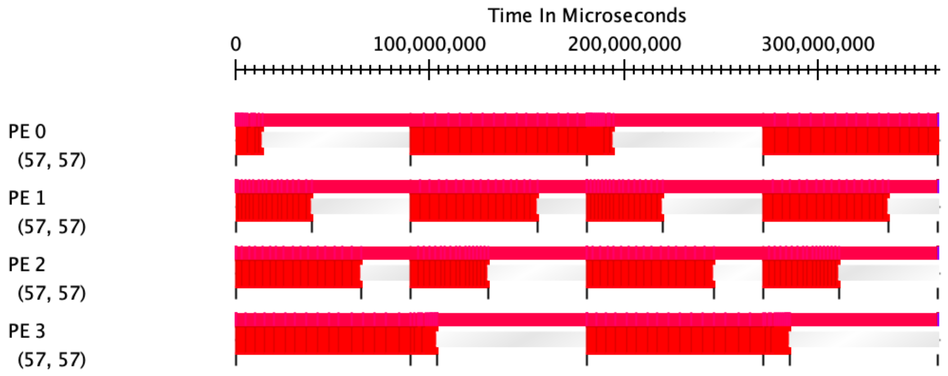
We use a simple phase-based mini-app to demonstrate the benefits of vector load balancing in AMPI. Each iteration in the mini-app consists of two phases. In the first phase each rank does work proportional to its index, and in the second phase each rank does work proportional to the total number of ranks minus its index. Thus, the total amount of work done in total by each rank across the two phases is the same for every rank, but the division of this labor between the two phases is different for each; ranks with small indices do little work in the first phase and a lot of work in the second phase, ranks in the middle of the index space do roughly the same amount of work in each phase, and ranks with large do a lot in the first and little in the second.

Figure 4.12 shows timelines for three different executions of this mini-app with 64 ranks on 4 PEs provided by the Projections performance analysis tool [45]. Each execution consists of two iterations, and, when used, load balancing is invoked between these two iterations. Figure 4.12a does not use load balancing and completes in 361.73 s, Figure 4.12b uses the scalar greedy strategy and completes in 305.34 s and Figure 4.12c uses the rk-d strategy, completing in 288.74 s. Overall, scalar load balancing gives a speedup of 1.18x and vector load balancing a speedup of 1.25x over the no LB case. Visually, the impact of load balancing is evident, with scalar LB dramatically shrinking the idle time at the ends of phases as compared to not using load balancing, but still leaving significant gaps that the vector load balancing ameliorates almost completely.

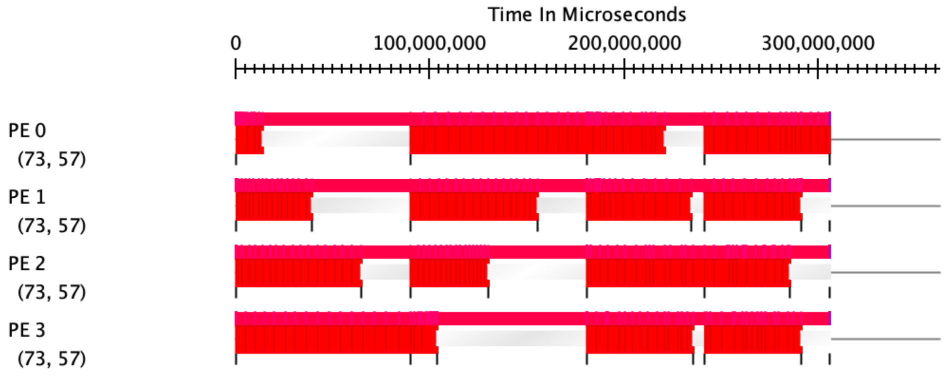
This rudimentary example is admittedly simple, but it showcases the ability of vector load balancing to improve upon what scalar load balancing can offer in an applied scenario and proves the viability of using our strategies for non-Charm++ applications.

4.5.3 Virtual Transport

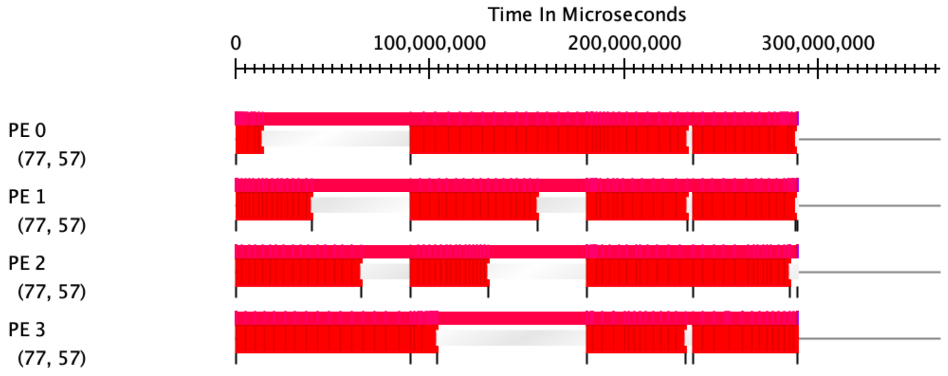
Virtual Transport (VT) [46] is a task-based parallel runtime system developed by the DARMA group from Sandia National Laboratories with many similarities to Charm++.



(a) No Load Balancing



(b) Scalar Load Balancing



(c) Vector Load Balancing

Figure 4.12: Phase-Based AMPI Mini-App Timelines

Notably, it has an active RTS that can measure load, support for migratability, and encourages overdecomposition, all of the prerequisites to load balancing as discussed in Section 2.4.

VT differentiates itself from other task-based runtimes in that it is intended to provide a harmonious and seamless interface between MPI applications and task-based execution, allowing users to enjoy the benefits a task-oriented decomposition can bring, such as load balancing, without requiring a wholesale rewrite of existing code or forcing abandonment of the familiar MPI ecosystem, environment, and libraries. While other task-based systems such as Charm++ support MPI interoperation, VT does this in a more integrated way, using MPI and its associated tools to handle parallel execution, network communication, compilation, and launching.

EMPIRE

One of the motivating applications for the development of VT is *ElectroMagnetic Plasma In Realistic Environments* (EMPIRE) [47], an unstructured finite element method particle-in-cell (PIC) code also developed at Sandia National Laboratories. This application is used to simulate the interaction of electric fields, magnetic fields, and high-energy plasmas in complex geometries, as may be found in fusion energy research, for example. EMPIRE is a purely MPI code, using Trilinos [48] and Kokkos [49] to provide performance portable single node parallel solvers.

There are two separate implementations of the PIC functionality in EMPIRE. The original implementation is a pure MPI design, in which cells are distributed across ranks according to the same decomposition as used in other parts of the code. Each rank is responsible for computations on particles found within the cells it owns.

Some input datasets result in large, dynamic load imbalance in EMPIRE, particularly in the particle operations. To address this, a second implementation was developed using VT, which we call EMPIRE-VT. This implementation obtains *overdecomposition* by further subdividing the sets of cells assigned to each rank. The data and computations on particles residing in these subsets of cells are managed by migratable objects, giving the load balancer scope to address imbalances. In EMPIRE-VT, these further subdivided pieces are called *colors*. To interact with the rest of the code, these objects each communicate with a “home” object on the original rank, to which its cells are assigned for the rest of the code.

EMPIRE and EMPIRE-VT are security controlled applications, so further details on its structure, composition, and applicability are not available for distribution.

Load Balancing in VT

VT supports measuring and recording vector loads in phase-based applications, with one load measurement per phase (note that in the parlance used by VT, what we call a *phase* VT calls a *subphase*, and what we call an *iteration* VT calls a *phase*; here we will use *phase* and *iteration*, respectively). However, at the time of writing, it does not provide any load balancing strategies that can exploit this level of per-phase load granularity.

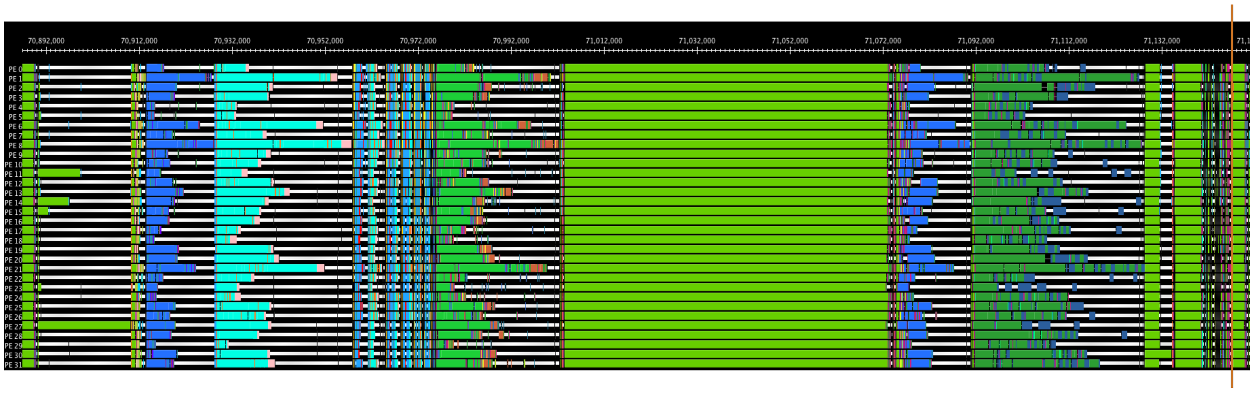
In order to support VT applications, we wrote an adapter that allows VT to use the vector load balancers that we developed for use in Charm++. Additionally, support for ingesting VT load logs was added to our LB simulation tool, enabling offline analysis and testing of various LB strategies and options with real data.

In particular, we focus on a representative dataset for EMPIRE-VT that presents fourteen distinct phases of execution per iteration. This data set is comprised of eight colors, a measure of the degree of overdecomposition for the particle-based work. We compare the results of *k*dLB and TemperedLB [50], a novel load balancing algorithm developed by the DARMA group, on a 32 node run of EMPIRE-VT in Figures 4.13 and 4.14. Experiments were performed on an internal cluster at Sandia equipped with 2.0 GHz ARM Cavium Thunder-X2 processors and connected with EDR Infiniband. TemperedLB, while sophisticated in its analysis of load distribution and migration decisions, operates only on a scalar load value and thus is unaware of the load distribution of individual phases in the computation. *k*dLB, however, utilizes a load vector of the per-phase loads in this case.

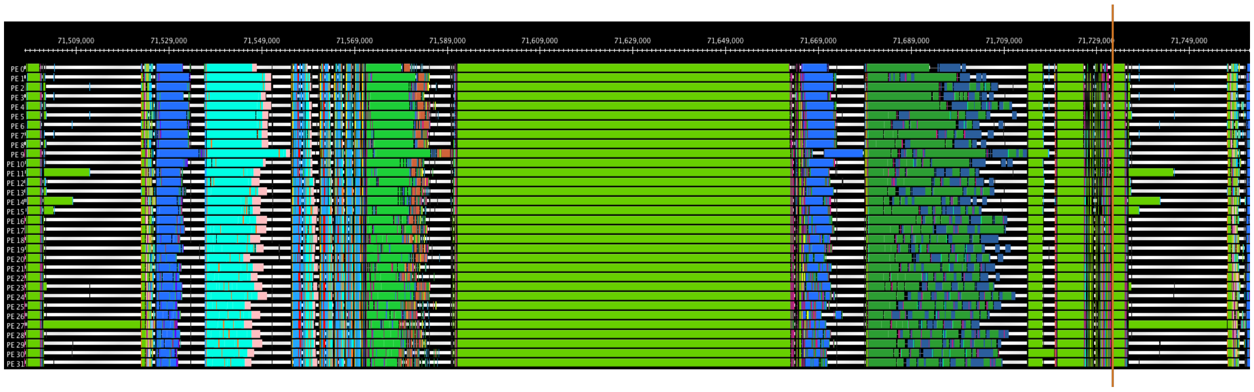
Figure 4.13 shows execution timelines of EMPIRE-VT with the two different load balancers as provided by the tracing framework inside VT and the Projections performance analysis tool. Time is on the x-axis and PEs from 0-31 are on the y-axis. The timestamps do not align since these are taken from two different runs, but the timescale is the same in both timelines and both show the same numbered iteration (304) from their respective runs, aligned for ease of comparison. The length of time shown corresponds to approximately one iteration for the TemperedLB run, about 263ms. For both timelines, the iteration begins at the left edge of the chart and ends at the vertical orange line extending beyond the top and bottom borders of the chart near the right side.

By visual inspection, it is clear that the iteration with *k*dLB is shorter than the iteration with TemperedLB. Comparing the different phases shown in the timeline indicates where this performance delta arises.

Figure 4.14 shows a high-level comparative analysis provided by VT tools. Each of the



(a) With TemperedLB



(b) With kdLB

Figure 4.13: Execution Timelines of EMPIRE-VT

graphs has time on the y-axis and iteration count on the x-axis. The top row of graphs shows the overall performance delta between the kdLB and TemperedLB runs. As expected, the plots of non-VT capable work in the bottom row indicate that there is negligible performance difference in the non-VT capable work between the two load balancers, as this corresponds to the cell-based work that is not load balanced and maintains the same static decomposition as the pure MPI code. Thus, the overall performance delta is attributable solely to the VT capable work shown in the middle row. These plots shows a divergence between the two load balancers, with kdLB generally resulting in shorter iterations, cumulatively resulting in approximately 12% less time spent in VT capable work across 1,000 iterations as compared to TemperedLB.

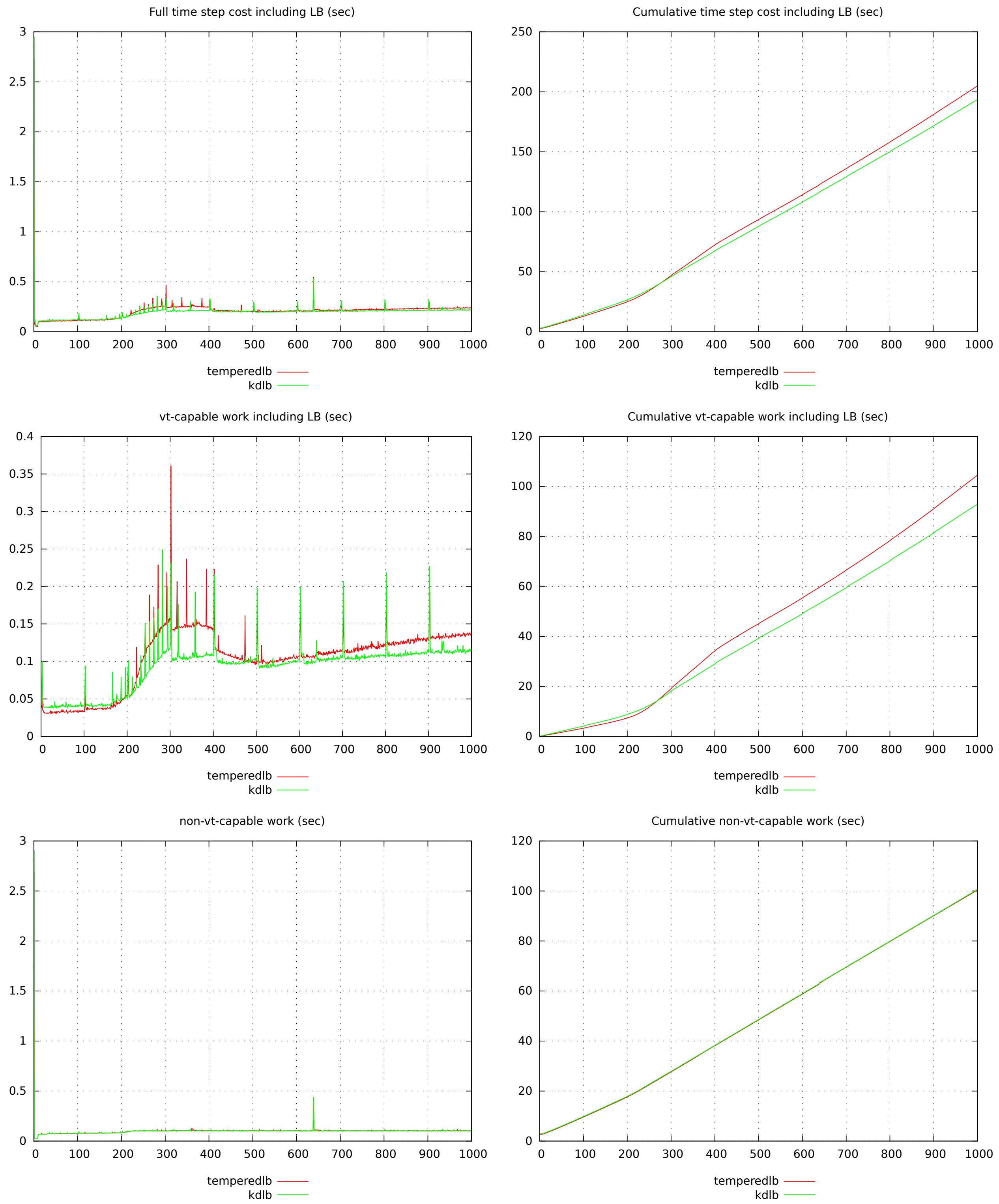


Figure 4.14: Comparison of EMPIRE-VT Performance with *k*dLB and TemperedLB

CHAPTER 5: HETEROGENEOUS LOAD BALANCING

In this chapter we analyze the load imbalance that arises in heterogeneous applications and systems and the ways in which using load vectors allows us to address this imbalance. "Heterogeneous" is a somewhat ambiguous term as used in extant literature, here we specifically use it in the sense of a system composed of a host CPU and an accelerator device, and we call an application heterogeneous when it contains at least one component eligible for execution on a CPU and at least one component eligible for execution on an accelerator device such as a GPU, and when these components are non-trivial (e.g. a CPU component must do some meaningful work beyond merely launching and waiting for the completion of GPU kernels). Further, we concern ourselves only with the case where these heterogeneous components run concurrently, overlapping their execution (Chapter 4 covers the case where they run sequentially as a series of phases).

We classify heterogeneous work into two categories: retargetable and non-retargetable. Retargetable applications have tasks that can run on multiple different targets, i.e. they have semantically equivalent implementations for multiple architectures that can be dynamically selected between at runtime based on some criteria. On the other hand, non-retargetable work can only execute on a single *a priori* designated target. These different types of work may coexist in an application, but here we focus on each in isolation to avoid confounding factors.

Section 5.1 explores load balancing for retargetable work, and Section 5.2 studies load balancing for non-retargetable work. We focus primarily on uses of NVIDIA's Compute Unified Device Architecture (CUDA) due to its popularity, but our methods apply equally to the use of any accelerator.

5.1 VECTOR MALLEABILITY FOR INTRA-NODE HETEROGENEOUS BALANCING[2]

5.1.1 Introduction

Many current supercomputers derive a majority of their compute power from accelerator devices. NVIDIA and AMD GPUs, as well as other accelerators, have already seen widespread adoption in many Top 500 machines. As the exascale era continues, several new machines will

[2]Contains content previously published in [51].

derive a sizable portion of their overall FLOPS from GPUs. These include both the Aurora system at Argonne National Laboratory and the El Capitan system at Lawrence Livermore National Laboratory. However, programming models and systems have been slow to adapt to this changing environment. In this section, we examine an extension to the Charm++ parallel programming framework that enables coordinated execution of heterogeneous tasks. We focus on compute kernels developed for NVIDIA GPUs using CUDA. Our framework automatically generates tasks from user-annotated functions that can be executed on either the host or device. This strategy ensures full utilization of available hardware and reduces computation time. In this section we examine the heterogeneous performance of two mini applications, a two-dimensional stencil application called `stencil2d` and a molecular dynamics application called `md`.

5.1.2 Background

The parallel structure and methods of Charm++ programs are described in a charm interface file, which is parsed by the charm translator `charmxi` to generate code for the runtime. In this work, we modify `charmxi` to generate both host and CUDA versions of the entry methods tagged for execution on different devices. It can be extended to generate code for any hardware platform, but these two targets are sufficient for our tests. We also augment the Charm++ runtime, adding the capability to schedule heterogeneous work across the host and device based on a provided heuristic.

Graphical processing units (GPUs) are becoming prevalent in the HPC community, as is evident from their number over time in the Top 500. Originally intended as special purpose accelerators for graphics applications, they are now user programmable and often referred to as the "device" (as opposed to the CPU or "host" cores) due to their supplementary use in a system. A variety of languages and tools for GPU programming exist (e.g. OpenCL [52], CUDA [53], etc.), but GPUs remain more difficult to program for than traditional host cores. Unlike CPUs, GPUs are made up of hundreds of lightweight cores grouped together into streaming multiprocessors (SMs). These SMs share critical resources, such as registers and shared memory. Collections of threads, called warps, are launched on these SMs and execute in lockstep. This unique design can lead to strong performance for some highly parallel applications, e.g. graphics, but can be hampered by its strict SIMD nature (for instance when encountering branch divergence in code). Data movement is also a concern since the GPU cannot directly access host memory using the usual memory bus. Therefore, data must be copied to the device before being used, which often limits performance due to the latency and bandwidth constraints associated with transferring data across the PCI Express bus.

5.1.3 Related Work

Load balancing for retargetable work is well studied in literature [54], [55], but the vast majority of work is limited to doing so for a particular problem, partitioning data across the hardware resources in an algorithm specific way.

A similar approach to using runtimes in heterogeneous environments can be found in the StarPU programming library [56]. They also schedule tasks, called codelets, which can have multiple implementations targeting different hardware platforms, and they automate data transfer dynamically across the different targets. However, StarPU does not have a mechanism to automatically generate kernels for different platforms as our work does.

The Legion programming model [57] can also execute in heterogeneous environments using similar techniques to those of StarPU, supporting variants of tasks and automating data movement, but relying on explicit kernels or using some third-party generation facility such as Kokkos.

We distinguish ourselves from other task based run times such as OmpSs [58] by offering more generality, not requiring entire programs to be explicitly constructed as a DAG. Similar work has also been carried out in the context of OpenCL [59], [60] with great success, but our work can be extended to work across multiple nodes in the future, due to the distributed nature of Charm++.

5.1.4 Methodology

Our execution model builds upon the earlier work of GPU Manager [61], which handles the delegation and execution of CUDA kernels in the context of the asynchronous message-driven runtime of Charm++. This allows us to focus our work on higher-level concerns, such as code generation and dynamic target selection in our framework.

Charm++ GPU Manager

The GPU Manager operates by registering target GPU kernels with the runtime system, and integrating with the RTS to coordinate data and execution flow between the host and the device. This integration enables the runtime to asynchronously invoke kernels when data is available on the device, which automates the overlap of data movement and execution as seen in Figure 5.1. Due to the inherent asynchrony of Charm++, it is important to ensure that blocking operations, such as `cudaHostMalloc`, are handled by the system and do not block in user code. GPU Manager also automates some tedious CUDA-related tasks, namely copying data to and from the device before and after kernel execution.

When using GPU Manager directly, the user must write an explicit CUDA kernel and denote buffers which need to be moved to and from the device. The programmer must also register a callback with the runtime, which is called when the kernel is finished and data has been copied back to the host. This step is necessary since the call to GPU Manager returns once the runtime has copied the CUDA buffers; it does not block until the kernel has finished. GPU manager coordinates data movement and kernel invocations through a FIFO queue. When a PE goes idle and enough time has passed, the runtime invokes a progress function to issue new requests to the GPU. At this time, GPU Manager attempts to offload data for a new kernel, launch a kernel with complete data on the device, and move data for the completed kernel back to the host. Finally, when the data for the completed kernel is fully copied back, GPU Manager invokes the user supplied callback to continue execution.

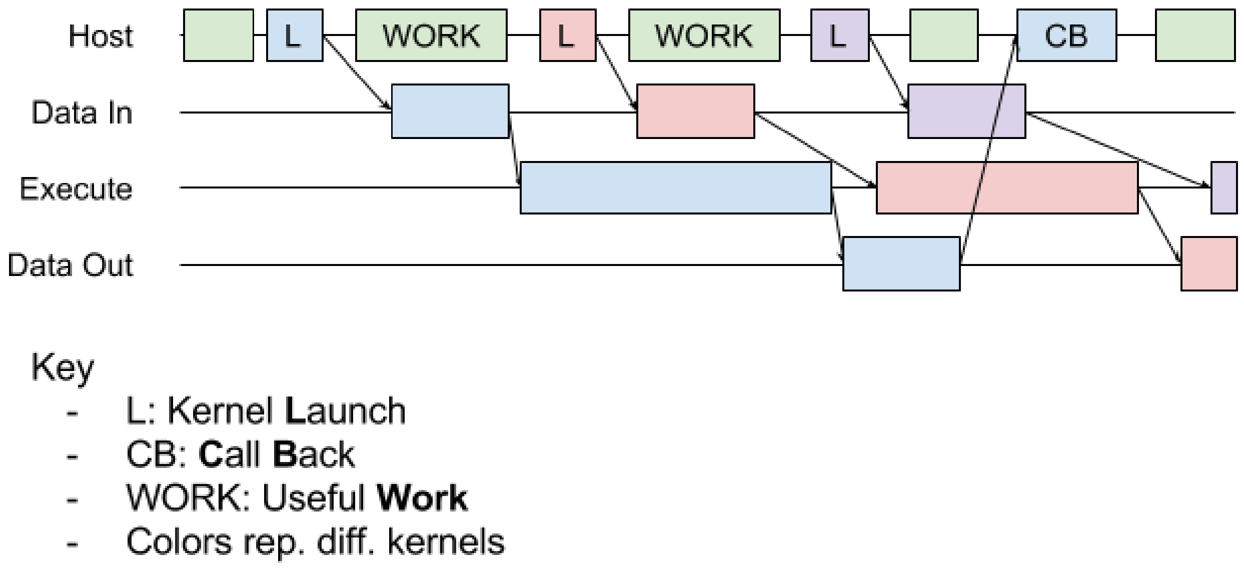


Figure 5.1: GPU Manager

Accel Framework

The Accel Framework [62], or ACCEL, extends GPU Manager by adding the functionality of automatically generating CUDA kernels from specially tagged code.

ACCEL alleviates many of the programmer productivity problems associated with using

GPUs effectively in parallel applications by virtue of its automatic kernel generation. This generation occurs only for entry methods annotated with the `accel` keyword. To improve performance, additional tags can be applied to methods, such as `splittable`, which allows methods to be split into several independent tasks, which can more fully utilize the many processors on a GPU. Inside `splittable` methods, `splitIndex` and `numSplits` variables are defined, analogous to the `threadIdx` and `blockDim` variables in CUDA. This differs from CUDA in that the code can be targeted to a variety of platforms. A full listing of other annotations can be found in [62].

In order to maximize GPU utilization and avoid serialization, ACCEL tries to batch multiple device method calls into a single kernel launch. This batching occurs when a specified count is reached or a certain amount of time elapses. The `triggered` keyword informs the runtime system that the accelerated entry method (AEM) will be invoked on every chare and that all chares will invoke said entry method before any chare invokes it a second time. Programmers can also specify the number of threads to be used per block in a kernel launch instead of having the runtime automatically determine one.

It is beneficial for the RTS to minimize data movement and overlap it with computation when possible. Data movement is automatically overlapped with computation as described in Section 5.1.4. Method parameters are automatically copied to the device, but are not copied back since the Charm++ model dictates that entry method parameters have no lifetime beyond the entry method. However, object data used in an `accel` annotated method must be marked as `readonly`, `writeonly`, or `readwrite` to indicate whether it should be copied to the device before the start of the AEM, from the device after the completion of the AEM, or both. Additional annotations such as `shared` and `persistent` allow the user to control the lifetime of the data on the device. With these annotations, `charmxi` automatically generates code to move data to and from the device. The `implObj` variable seen in the code is required to refer to the fields of the chare itself due to the lack of a proper Charm++ compiler since we require a handle to the chare object and its data; it can be thought of as equivalent to `this` in a regular C++ object.

Listing 5.1 shows how these annotations may be used in the example AEM `doCalculation`. The variables variables `matrix` and `matrixTmp` are used as input and output for `doCalculation` and are linked to corresponding fields of the chare via `implObj`. The last token in Listing 5.1, `doCalculation_post`, specifies a callback invoked when the `accel` entry method is finished executing. It is used in the same way as in GPU Manager but is listed here instead of as an input or member variable due to parsing constraints. The callback is used to send messages to invoke other methods since Charm++ messages cannot be sent from accelerator devices.

```
entry [triggered splittable(NUM_ROWS) accel] void doCalculation() [
 readonly:float matrix[DATA_BUFFER_SIZE]
     <implObj->matrix>,
 writeonly:float matrixTmp[DATA_BUFFER_SIZE]
     <implObj->matrixTmp>
] { ... } doCalculation_post;
```

Listing 5.1: Accelerated Entry Method Annotations

Host-Device Load Balancing with Accel

As described above, ACCEL produces CUDA kernels from specially tagged entry methods, but these entry methods can also be executed on the host CPUs of machines. Thus, at runtime, there are two semantically equivalent compiled versions of these entry methods. By virtue of generating multiple versions of entry methods targeted at different hardware resources, ACCEL has the ability to dynamically spread the execution across the available hardware.

ACCEL has a variety of strategies to determine where to execute particular entry methods. The strategy is passed to as a runtime argument. Example strategies include `+accelHostOnly`, `+accelDeviceOnly`, `+accelPercentDevice`, which specify a static division of work between the computing resources. In our experiments, we manually sweep through different static divisions to observe the performance behavior of the various configurations. However, there are several available automated methods to find the best split, such as greedy strategies and hill climbing. Further description of available strategies is detailed in [62].

5.1.5 Results

We analyze performance for varying distributions of work between the host and device for two different applications, `stencil2d`, which implements a two dimensional stencil, and the more complex `md`, which simulates electrostatic molecular dynamics. In both applications, the main compute methods have been annotated with `accel` and other tuning parameters. Our tests vary the percentage of work allocated to the device from 0% to 100% in increments of 5%. Theoretically, hybrid computation will improve performance, since more hardware can be used, but data transfer and batching costs create performance impediments.

The experimental results were gathered on the Stampede supercomputer [3]. In particular, we used the visualization nodes of the system, which each feature an NVIDIA K20 GPU and two Intel Xeon E5-2680 processors. All runs were performed on a single node of the system with 16 Charm++ processing elements, matching the 16 cores in the node. We measured elapsed time from the start of the calculation to the end of the last error calculation for both applications. This does not include startup or other fixed costs.

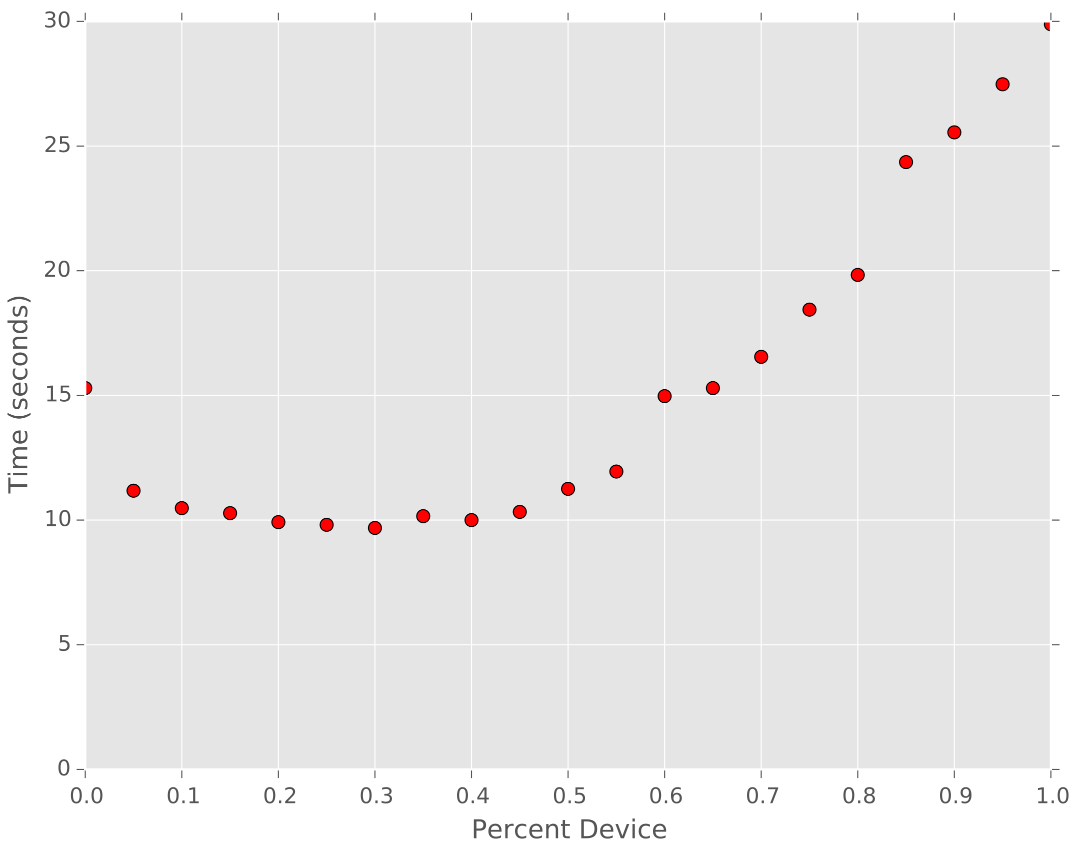


Figure 5.2: Timing for `stencil2d`

5.1.6 Stencil 2D

`stencil2d` performs a single-precision weighted five point stencil. Given results use a 6144x6096 2D array decomposed into 24 tiles per dimension, a 254x254 section per element. For work performed on the GPU, the algorithm performs approximately 1.25 single-precision FLOP per transferred byte (10 FLOP/(1 float in + 1 float out)). As shown in Figure 5.2,

[3]https://www.tacc.utexas.edu/stampede1/

this low FLOP/byte ratio causes the host only case to beat the device only case. Optimal performance occurs in the 30% device case.

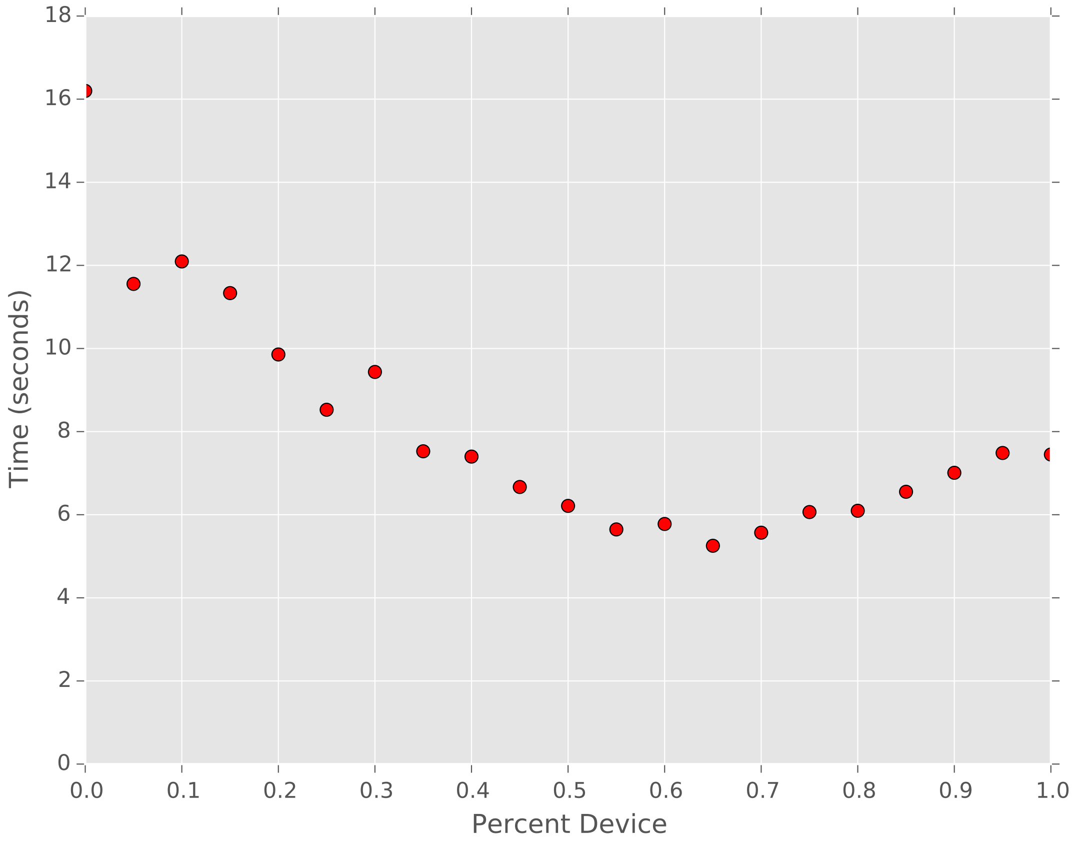


Figure 5.3: Timing for `md`

5.1.7 Molecular Dynamics

`md` executes much faster in the device only case than in the host only case. Given results use a 5x5x5 3D array with 256 molecules per array element, a total of 32k molecules. The FLOP per byte ratio for `md` is higher than that of `stencil2d` since each particle has a relatively complex interaction with every other particle in the simulation, requiring distance, electrostatic force, position, and acceleration calculation and normalization. Optimal performance occurs in the 65% device case.

5.1.8 Analysis

For both selected applications, we observe an increase in performance when using both the host and the device as compared to using only the host or only the device.

There are some clear discontinuities in the performance of the chosen applications; see the jumps at 60% and 85% device in Figure 5.2. These are likely a performance artifact of the batching used in the Accel Framework. Since the GPU is a throughput-oriented device, launching an additional batch takes much longer than adding some work to an existing batch. This behavior is not seen for smaller allocations of work to the device because the host was spending more time on the work than the device, so it was the dominant term.

The timing data follows a "bathtub plot", so termed because it is low in the middle and high on both sides. When performance follows this pattern, the goal is to set the parameters such that execution happens in the "floor" region. As shown in Table 5.1, the best configurations achieve speedups of between 1.46x and 3.09x relative to host only and device only configurations.

	Best Split	Host Only	Device Only
`stencil2d`	30% device	1.58x	3.09x
`md`	65% device	3.02x	1.46x

Table 5.1: Speedup of Best Configuration Relative to Host/Device Only

5.1.9 Caveats

Not all applications benefit from a heterogeneous execution system. Even applications that are amenable to heterogeneous execution may not see benefit in all configurations. The most significant reason for this is data movement. Just as HPC applications can slow down when run on two nodes versus one node due to the effects of adding network communication, using a GPU can degrade performance unless the application amortizes the costs of data movement. Additionally, not all algorithms are well suited to run on the GPU. In particular, programs that make heavy use of branching, that cannot expose enough parallelism to fully utilize the GPU, or that are composed of a variety of disparate tasks do not perform well on GPU hardware.

However, large HPC applications often feature a variety of different kinds of work, so it is likely that some portion will improve when executed heterogeneously.

5.2 VECTOR LOAD BALANCING FOR NON-RETARGETABLE WORK

5.2.1 Introduction

In this section, we study the issue of load imbalance in non-retargetable heterogeneous applications. Existing load measurement and balancing approaches often focus solely on tasks that execute on the CPU, ignoring the existence of accelerator devices altogether. However, load imbalance afflicts work performed on accelerators in the same way as it does to CPU work. This problem is increasing in importance and proliferation as GPUs continue to improve in peak performance at a higher rate than CPUs and applications utilize GPUs for higher and higher proportions of their entire computation.

GPUs can achieve immense performance, but they are not well suited for all workloads. GPUs generally provide much higher memory bandwidth than CPUs and can provide higher throughput for kernels composed of data parallel instructions with regular access patterns. However, tasks with complicated, irregular control flow, access patterns, or data dependencies are often better suited for execution on the CPU instead.

As discussed in Chapter 4, many modern science and engineering applications use techniques such as multiphysics in their implementations. Each of these different physical processes are simulated in their own particular way, some of which may be more appropriate for GPU execution, and others for CPU execution. In addition to their core computation, applications usually have modules for interfacing with external services, such as performing data ingestion, visualization, or *in situ* analytics. All in all, applications are comprised of a diverse set of different computations, and it is likely that a performance maximizing configuration will run some of these on the host CPU and some on accelerators.

When these different modules execute in an overlapped fashion, with the CPU and GPU each executing their assigned work simultaneously, iteration time is determined by which of these resources finished their work last. Thus, rather than sum-based objective and mapping evaluation functions used for load balancing phase-based applications in Chapter 4, the maximum-based objective and mapping evaluation functions given in Equations (3.2) and (3.4) apply in this case.

5.2.2 Motivation

While using retargetable tasks for heterogeneous execution has the benefit of offering scope for dynamic runtimes to adaptively balance the workload across all of the different available resources based on utilization, there are several good reasons why developers may use a non-retargetable approach when adding support for heterogeneity to their applications.

First, writing explicit kernels to target a particular architecture allows developers to unlock higher levels of performance and perform more detailed tuning by using non-portable low level primitives or vendor-specific proprietary features, analogous to the use of inline assembly or microarchitecture-specific extensions in code targeting CPUs. Additionally, supporting only a single target lets developers take advantage of bleeding edge features that may take much longer to be supported by general frameworks and to use specialized vendor provided libraries, such as NVIDIA's collective communication library NCCL [63].

Specifying the location of where to execute a task provides more predictable, consistent performance, avoiding potential thrashing issues and enabling certain optimizations. A key factor in achieving high performance from a heterogeneous program is limiting the amount of data movement between the host and accelerator. This communication occurs across some sort of bus that links the hardware resources together; for modern systems this is usually PCI Express, with an approximate round trip time of 2 µs [64]. Higher performance proprietary connections such as NVIDIA's NVLink [65] exist and are sometimes used in HPC system, such as the Summit machine at Oak Ridge National Laboratory[4], but these mostly improve the bandwidth of transfers, while latency improvements are more modest, bounded by the physical constraints of the distance between the resources and the speed of light. Dynamic retargeting of work requires data to be accessible from the new resource, meaning that it must be transferred over this connection. When a task is frequently retargeted between a CPU and GPU, it induces thrashing, as the associated data is repeatedly moved back and forth, accumulating a large latency penalty. While researchers have developed methods that attempt to decrease the number of necessary transfers [66], the best way of avoiding them is often to restrict tasks to a single type of resource. Separately, when the location of data is known, internode transfers can use optimized techniques such as remote direct memory access (RDMA) to move data directly from the memory space of one GPU into the memory space of a remote GPU without having to copy to and from the host memory first.

It is usually easier to add GPU support to an application in a non-retargetable way. The official documentation for most accelerators will direct users to write code explicitly for that platform. Additionally, there are more available training materials and it is easier to find experienced developers for something like CUDA rather than a retargetable model, and these retargetable models can be unreliable and lacking in support, as they often come from

[4]https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

academia or research laboratories lacking the funding of a large corporation.

Finally, it is often less intrusive to integrate into the structure of existing applications. Support for retargetable work often requires the use of exotic, experimental compilers or a new runtime system, which may be incompatible with parts of an application or require significant porting effort, whereas direct use of an GPU may be as simple as using the vendor's well tested compiler for the kernel and adding a single additional library to the existing build process. It also allows for easier use of tools like first party profilers and debuggers.

5.2.3 Related Work

Developers and users have identified CPU-GPU load imbalance as an issue for scaling the astrophysics application ChaNGa [16] and the molecular dynamics application NAMD [15] on modern heterogeneous machines. Both of these applications feature non-retargetable work, determined by the suitability of different parts of their code to GPU execution.

Load balancing for non-retargetable work is well studied in literature, but almost always only concerning itself only with GPU load balancing, ignoring the CPU. Chen *et al.* propose a solution [67] that divides work into fine grain tasks and enqueues them into a set of GPU task queues, from which GPU threads take work as they become free. This potentially allows for heterogeneous execution and dynamic load balancing. However, they use a work stealing approach with a persistent device kernel, instead of a central manager, and they do not show results for mixed CPU-GPU execution as presented in this section. Fazenda *et al.* apply the dynamic load balancers of Charm++ to a fully GPU-accelerated application running on top of AMPI [68], again only balancing GPU work. Hagan *et al.* present a scheme [69] to load balance a mixed simulation and visualization workload across GPUs. There are several other similar studies [70]–[72].

StarPU [56] and Legion [57], previously discussed in Section 5.1.3, support balancing non-retargetable work in the case where tasks only have one variant, meaning they can only execute on a single type of hardware resource. However, compared to our approach, their balancing schemes are more akin to the work stealing/spreading style, making mapping decisions on the fly as new tasks become available rather than the periodic remapping we do, and they are also more tightly wedded to their runtime systems. Additionally, we inherently support load balancing for distributed systems, while other systems may require additional modules or mappers to do so.

5.2.4 Synthetic Results

Using the same configurations and parameters detailed in Section 4.4.1, with the addition of a new exponentially distributed case where each dimension of the load vector is sampled from an exponential distribution with $\lambda = 0.15$, we evaluate our load balancers for the overlapped execution case. Mappings are evaluated using the maximum evaluation function given in Equations (3.2) and (3.4). The results show the minimum, maximum (indicated by bars), and median across one hundred runs with different RNG seeds. Note that while these tests use up to six dimensions, load balancing across a host and accelerator in practice generally only uses two dimensions, one for each hardware target. It is possible that future machines may necessitate the use of more than two dimensions for load balancing by virtue of having more complex structures than current systems, such as the use of accelerators from different vendors on the same node or the adoption of FPGAs or other specialized hardware in addition to the GPUs commonly used today,

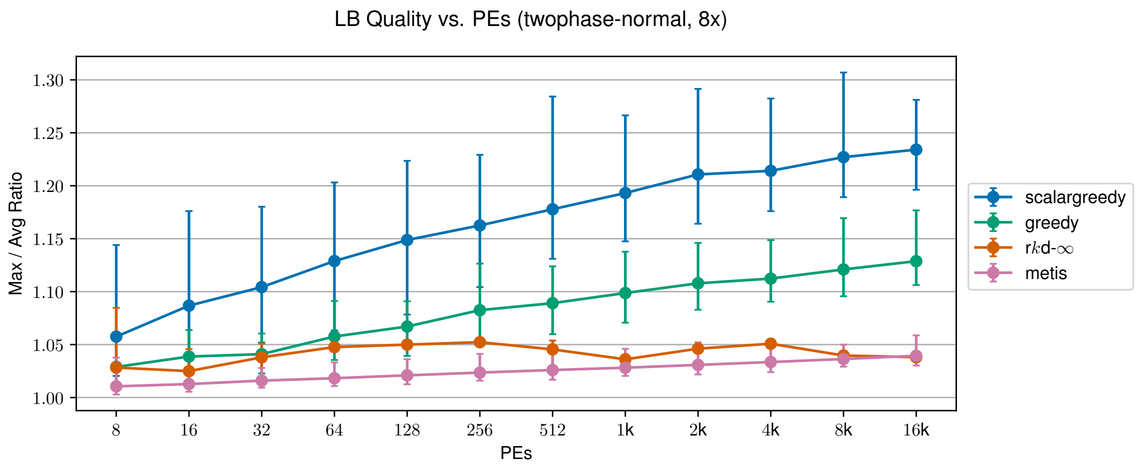
Results are shown in Figures 5.4 to 5.7. In general, the rk-d and METIS strategies perform the best for the normally distributed and complex three dimensional load scenarios at all scales except the largest scale for the latter load scenario. The exponentially distributed case is more interesting, with the rk-d and METIS strategies showing stepwise scaling behavior and losing to scalar greedy in the two dimensional runs at large scale from 4,096 PEs onward. The (exponentially, normally) distributed case is the most interesting of the lot in that METIS performs very poorly, much worse than both of the greedy strategies for the large scale two dimensional runs and comparably for all other cases. Meanwhile, rk-d delivers a maximum $Max : Avg$ ratio of approximately 1.1 across all numbers of dimensions and scales.

5.2.5 Practical Results

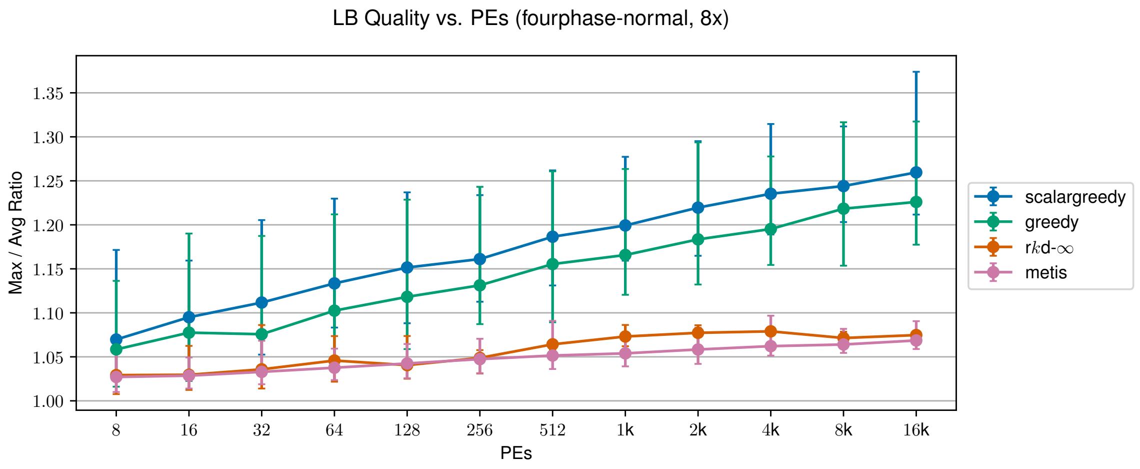
We evaluate the utility of our vector load balancing methods for non-retargetable heterogeneous work on the Delta supercomputer[5] at NCSA with the `mpi-linux-x86_64-cuda` build of Charm++. In particular, we use the nodes containing four Nvidia A40 GPUs and AMD Milan 7763 CPUs, running with four Charm++ PEs per node, one per GPU.

Tests shown here use the same 3D 7-point stencil mini-app used in Section 4.5.1, modified to add support for GPU execution and GPU load measurement. Two paradigms of work distribution are used in the tests, *separate* chares and *unified* chares. In the separate chare paradigm, each chare is assigned only one task per iteration, half of the chares only performing

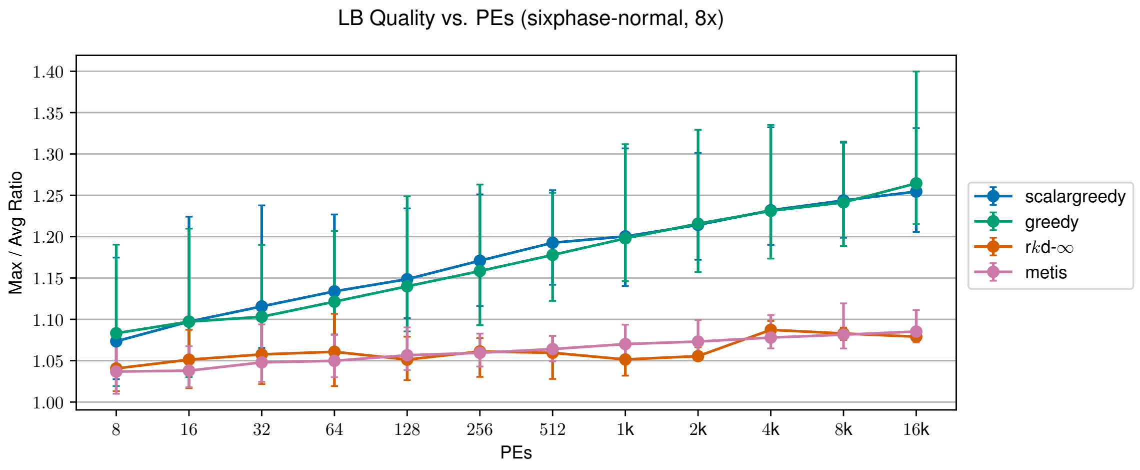
[5]https://www.ncsa.illinois.edu/research/project-highlights/delta/



(a) Two Dimensional

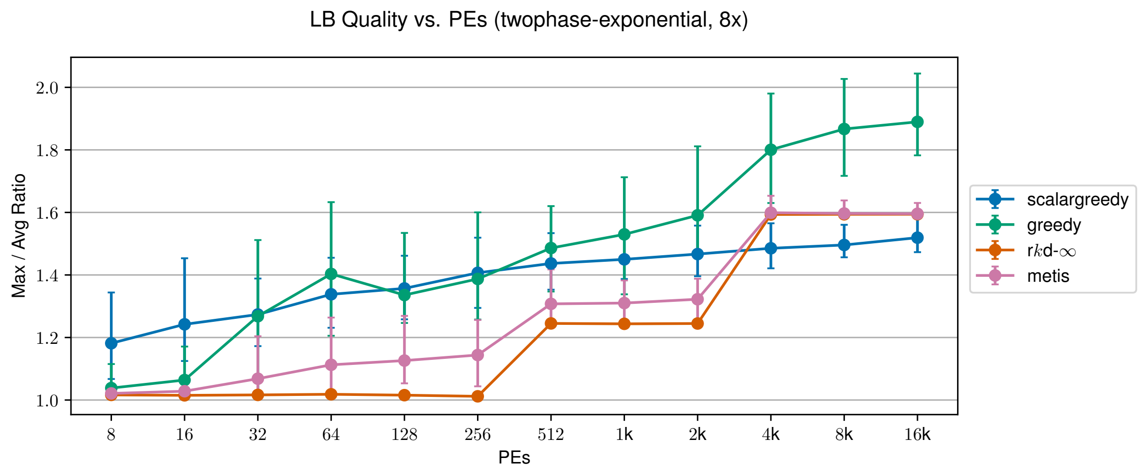


(b) Four Dimensional

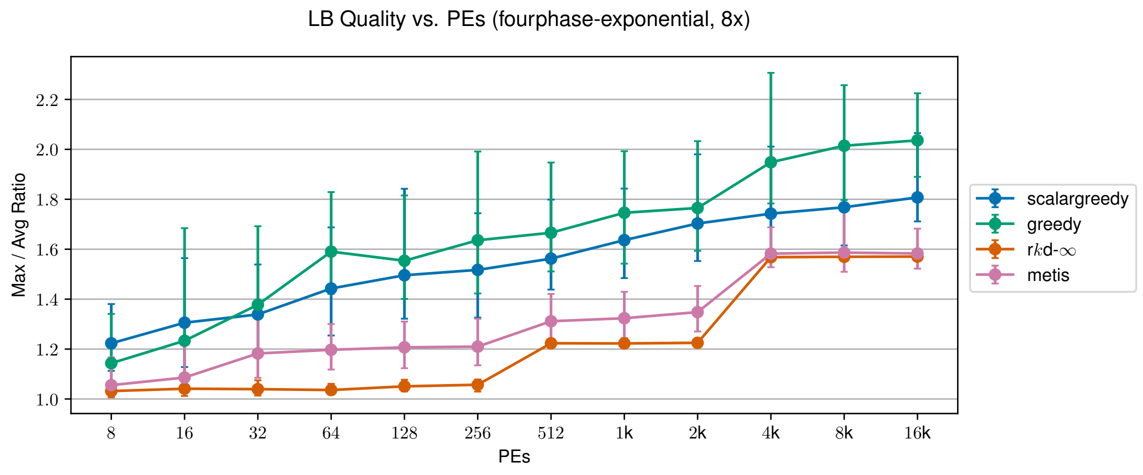


(c) Six Dimensional

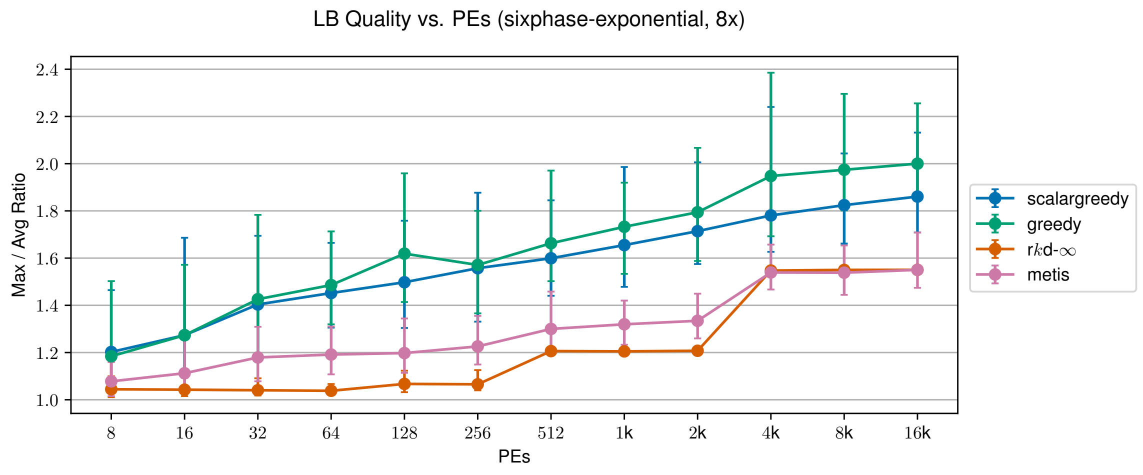
Figure 5.4: Quality Comparison with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional

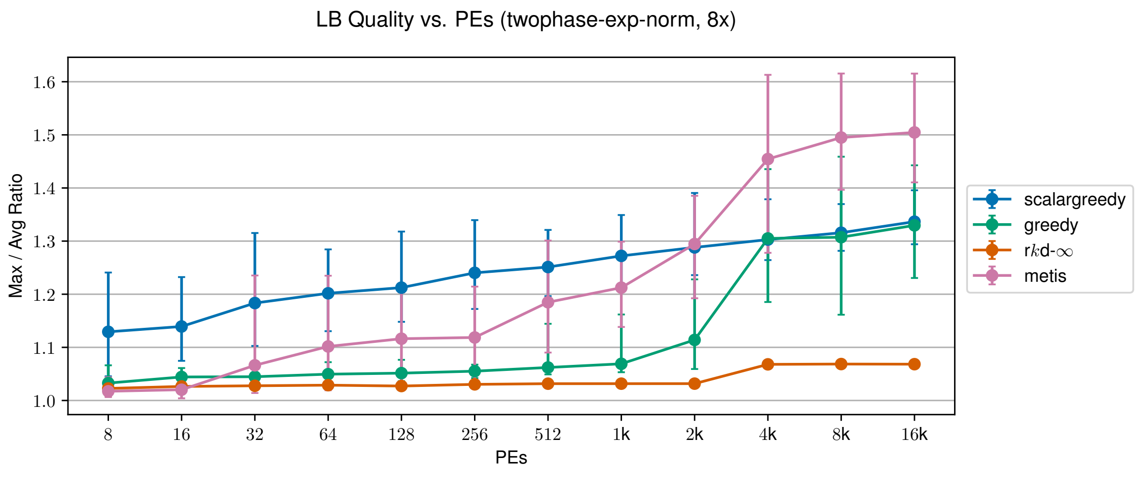


(b) Four Dimensional

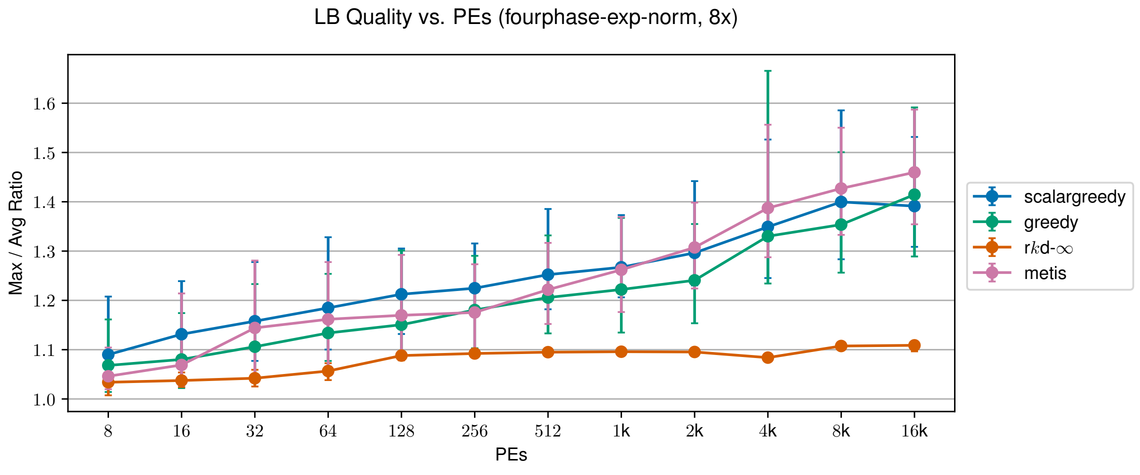


(c) Six Dimensional

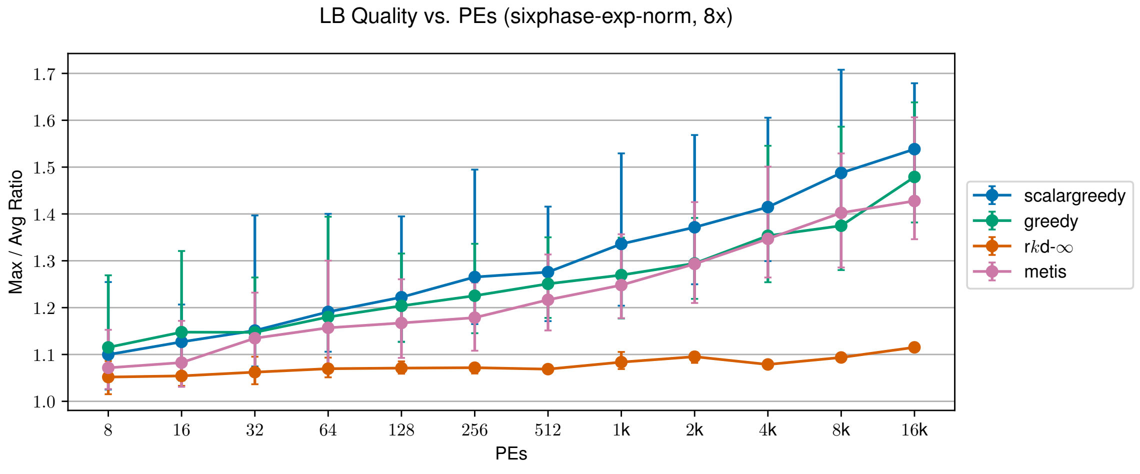
Figure 5.5: Quality Comparison with 2/4/6-Exponentially Distributed Vectors, 8 Objects/PE



(a) Two Dimensional



(b) Four Dimensional



(c) Six Dimensional

Figure 5.6: Quality Comparison with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

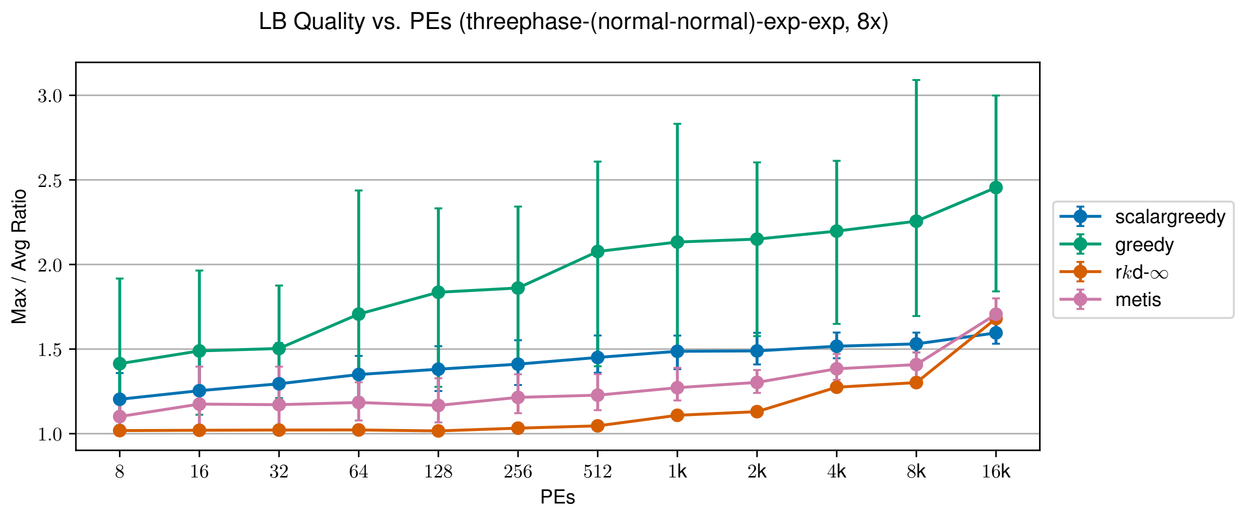


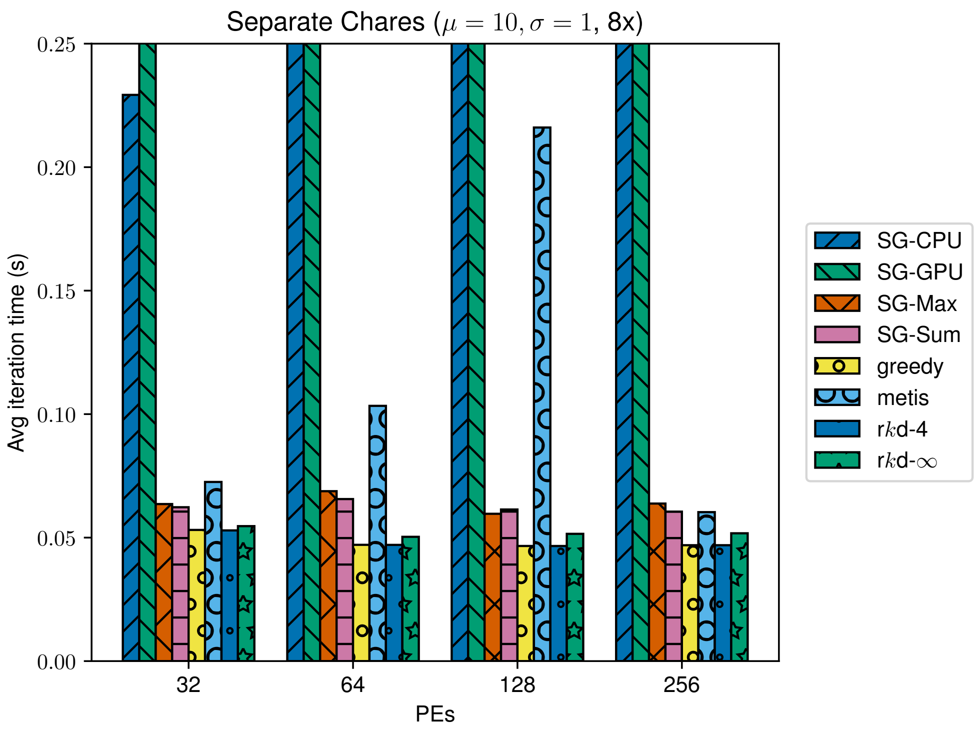
Figure 5.7: Quality Comparison with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

CPU-based work and the other half only performing GPU-based work. By contrast, in the unified chare paradigm, every chare has two tasks, one CPU-based and one GPU-based, which can run concurrently with each other. The workload of each task is sampled from a normal distribution with $\mu = 10$ in all tests and either $\sigma = 1$ or $\sigma = 5$, depending on the test. The tests perform weak scaling with a fixed ratio of eight objects per PE and the provided timings are the mean of five runs with a fixed RNG seed.

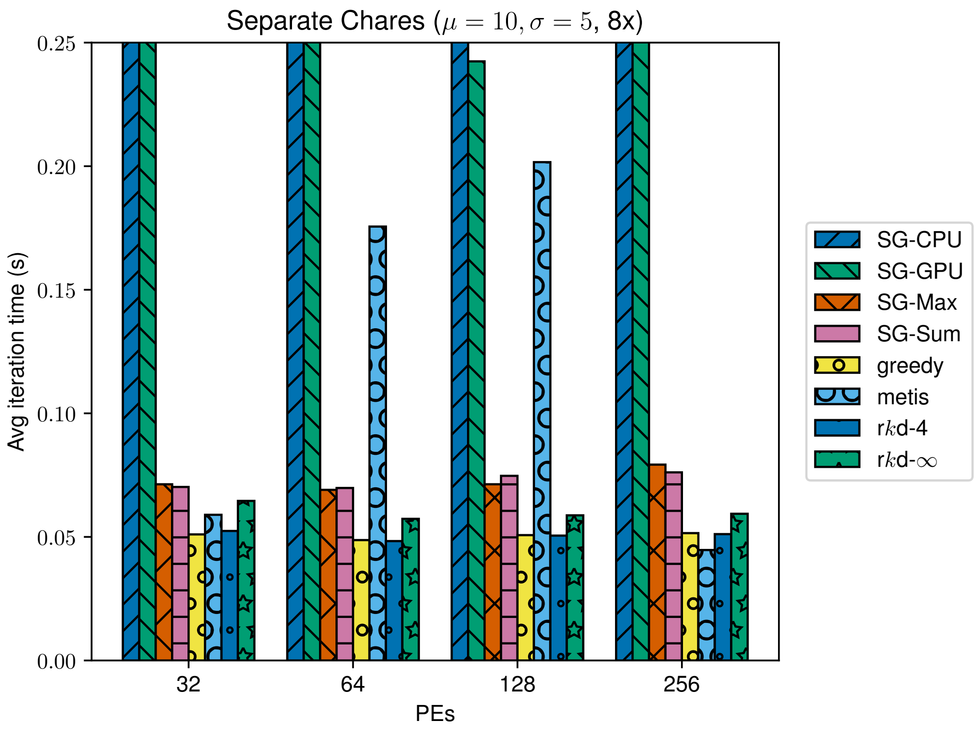
As "load" is now coming from two different hardware resources in these tests, we evaluate several different scalar load metrics with the scalar greedy strategy, only CPU timings (SG-CPU), only GPU timings (SG-GPU), the maximum of the CPU and GPU timings (SG-Max), and the sum of the CPU and GPU timings (SG-Sum).

Results for the tests with separate chares are shown in Figure 5.8, with $\sigma = 1$ in Figure 5.8a and $\sigma = 5$ in Figure 5.8b. SG-CPU and SG-GPU performed very poorly in these tests; for the sake of keeping the visualization intelligible, full results for these LBs are provided in Table 5.2. These results are so poor because they each completely ignore one of the two approximately equally contributing dimensions of work. The times for SG-CPU are generally better than SG-GPU because the GPU-only chares have a small CPU workload due to kernel launching, communication, and synchronization, preventing SG-CPU from placing all of the GPU chares on the same PE, whereas the CPU-only chares have essentially no GPU load, allowing SG-GPU to concentrate them onto the lightest GPU-loaded PE.

SG-Max and SG-Sum provide much better results since the resulting loads have some



(a) $\sigma = 1$



(b) $\sigma = 5$

Figure 5.8: Average Iteration Time with Separate Tasks, 8 Objects/PE

PEs	Avg. Iter.	Time (s)
	SG-CPU	*SG-GPU*
32	0.23	1.31
64	0.27	2.59
128	0.31	5.16
256	0.27	10.34

(a) $\sigma = 1$

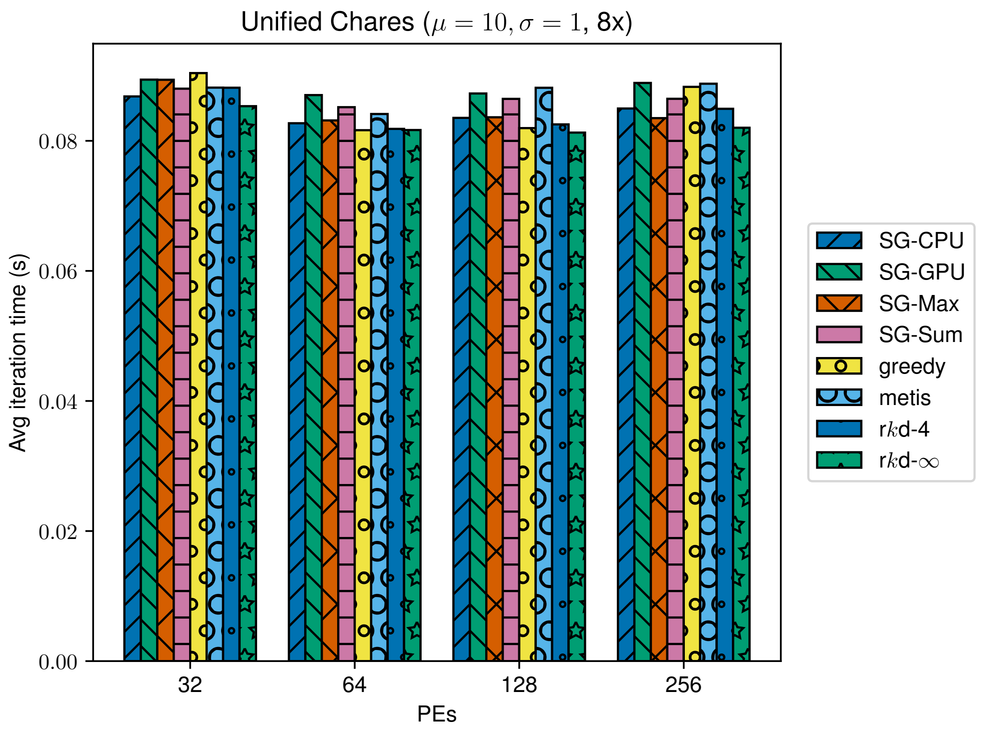
PEs	Avg. Iter.	Time (s)
	SG-CPU	*SG-GPU*
32	0.49	0.48
64	0.50	1.40
128	0.55	0.24
256	0.64	2.71

(b) $\sigma = 5$

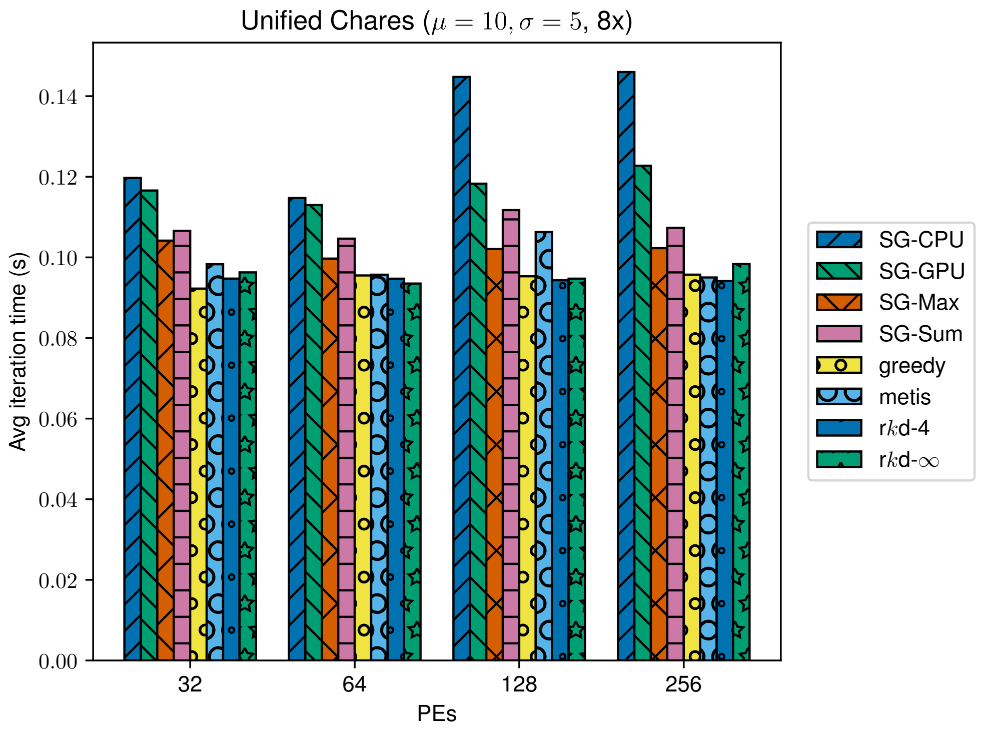
Table 5.2: Average Iteration Time with Scalar Greedy-CPU and Scalar Greedy-GPU

correspondence to the actual workload of the chares, but these are generally worse than the vector-aware strategies because the scalar balancer is not aware of which hardware target the load corresponds to. The METIS strategy is inconsistent in these tests, providing reasonably good results competitive with the other vector LBs at 32 and 256 PEs, but giving rather poor balance quality at 64 and 128 PEs. Vector greedy and the norm-based rk-d strategies provide the best results for these tests overall. The greedy strategy's weakness of making placement decisions based on only one dimension is not an issue in these tests due to nature of the work distribution of the separate chare paradigm. The slightly worse results given by rk-d with the ∞ norm are likely due to noise or variability in the non-significant dimension preventing improved placements (as an exaggerated example, placing an object with load $\langle .05, 5\rangle$ on a PE with load $\langle 10, 4\rangle$ instead of one with load $\langle 10.05, 0\rangle$).

Results for the tests with unified chares are shown in Figure 5.9. The different metric variants of scalar greedy perform much better with the unified chare paradigm than they did with separate chare paradigm, as all of the different metrics now reflect at least some meaningful portion of the total load of each chare. All of the load balancers provide approximately similar results in the $\sigma = 1$ case shown in Figure 5.9a, with all results being 8.5% of the best performing LB at every PE count. This is expected, as the CPU and GPU workload for each chare should be very similar to each other due to the equivalent mean and small standard deviation of the generating distribution. When every dimension of a set of vectors highly correlate with each other, the problem consists of essentially only one dimension (see Section 8.2.3 for discussion on exploiting this property in general). However, for the $\sigma = 5$ case in Figure 5.9b, results are more similar to those seen in the separate chare case, albeit less extreme. The scalar greedy strategy performs poorly, especially the myopic SG-CPU and SG-GPU variants. SG-Max is the best of the scalar strategies, as the total wall clock execution time of a chare in this test is the maximum time taken between its CPU and GPU tasks. The vector strategies beat the scalar strategies in all cases (except for the anomalous METIS LB at 128 PEs).



(a) $\sigma = 1$



(b) $\sigma = 5$

Figure 5.9: Average Iteration Time with Unified Tasks, 8 Objects/PE

CHAPTER 6: LOAD BALANCING WITH CONSTRAINTS

The work of the previous chapters was driven by developing load balancing strategies for applications where runtime performance is primarily characterized by a load vector consisting of time measurements of portions of the program. In this chapter, we study the load balancing problem for cases where application performance or execution feasibility also depends on non-temporal measurements.

6.1 INTRODUCTION

The load vectors to objective functions defined in Chapter 3, more specifically, Equations (3.1) and (3.2) apply when the quantities making up the dimensions of the vector are of the same type. In particular, in a typical use case where Equation (3.1) applies, the dimensions correspond to sequential phases of a parallel computation, each measured in units of time, such as seconds. In that case, it is clearly sensible to add the maximum times for each phase and to use this sum as the overall metric to assess the quality of the resulting object mapping. Similarly, for Equation (3.2), a typical use case is for applications comprised of tasks with overlapping execution, for example a program with some tasks that execute on the CPU while other tasks execute on the GPU or other accelerator. Even though these are executing on different targets, since these loads can both be measured in units of time, if the loads may execute concurrently, the equation makes sense because taking the maximum of the two different dimensions of load accurately describes the makespan of the application and the combined load presented to the hardware resources of the system.

These assumptions break down when the load vector is composed of values measured in different units, such as when the dimensions of the load vector do not all represent a measurement of execution time. In this chapter, we will focus on how vector load balancing can be used to deal with these situations. As we demonstrate, measuring and tracking the load of individual objects across different types of metrics, combined with modifications to vector load balancing algorithms to support new balancing objectives allow us to beneficially apply load balancing to an expanded set of application and resource scenarios.

6.2 RELATED WORK

The Dynamic Resource Utilization Model (DRUM) [73], [74] assesses the demand and availability of multiple system resources, such as CPU time and network usage, via a combination of static benchmarking and dynamic monitoring of applications. The model uses a weighted sum to distill these characteristics into a scalar value they call “power”, which it then uses as input for scalar load balancing. In their tests, DRUM provides improved execution time over traditional methods. Reducing the problem down to scalar load balancing allows DRUM to be used with any of the wide variety of extant scalar LB techniques, but it also fundamentally prevents the load balancer from distinguishing the unique imbalances of different resources.

Merkel *et al.* develop a system to estimate the temperature impact and power consumption of tasks and create a scheduler that avoids overheating CPUs in [75]. By avoiding overheating induced CPU throttling, they are able to improve the performance of their target applications, but this is not an explicit goal and they do not explicitly consider the computational load of tasks when making placement decisions.

Sarood *et al.* [76], [77] combine adjustable dynamic voltage and frequency scaling (DVFS) with a custom temperature-aware load balancing strategy to constrain processor temperatures and reduce cooling energy while minimizing performance loss. Similar ideas of load balancing under frequency, power, and/or thermal constraints are explored in [78]–[80].

Bremer *et al.* apply constraint-based load balancing to a storm surge simulation in [81], developing a semi-static scheme to balance cells in their application in terms of computational load while remaining below a memory threshold, borrowing the multi-constraint refinement algorithm from [82].

The multi-criteria graph partitioning schemes in [30] are constraint-based, not minimization-based, allowing the caller to provide an array specifying the desired maximum threshold for each dimension. However, in practice, these thresholds may be relaxed during the refinement phase [31], resulting in mappings which violate the specified constraints.

6.3 MOTIVATION

When balancing with dimensions of disparate measurements, there are several problems with the utilizing vector load balancing as we have in previous chapters:

Comparability The most glaring of these issues is that values of different types are not necessarily comparable. Consider a set of object load vectors consisting of one dimension measuring CPU time utilized by the object and another measuring the number of network messages sent by the object. When balancing with these vectors, how should the strategy respond when faced with a situation where decreasing the maximum load in one dimension causes an increase in the other? This question is easy to answer for vectors where each dimension measures time spent in phases: choose the option that maximizes the difference between the decrease and increase, as that will provide the largest net benefit to the makespan. However, when the dimensions are of different types, the answer to this question is much less clear. Increasing the number of messages sent from a node may have no significant impact on application performance because the latency will be hidden, or perhaps it may cause a large and sudden reduction in performance because the extra messages may now cause a link to saturate or fill up a router buffer.

In a vector with mixed types, the impact of changes in a dimension varies depending on the dimension and overall composition of the vector. Some dimensions may tolerate changes within some margin without affecting performance at all, whereas changing others may cause a linear or non-linear impact on performance. Fundamentally, it is the load balancing equivalent of trying to compare apples and oranges; there is no meaningful way to compare such vectors without providing a customized objective function, which we do not support as it would greatly complicate the design and performance of load balancing strategies.

Scale While every value in a load vector must be a number by definition, the values in each dimension may be of vastly different scale, depending on the units of the measurement provided by the user or runtime system. Suppose the user passes in a dimension representing the time an object has spent in a critical section in terms of clock ticks and that is placed in the vector alongside runtime measured phase time in seconds. For the same interval of time, the two measurements will vary by a factor of a billion! Such a large disparity will cause a norm-based balancer, for example, to likely ignore the dimension denominated in seconds, as its impact on the norm of the load vector will be negligible. Thus, even when two dimensions are semantically comparable, different scales can prevent meaningful comparison.

This can be avoided by normalization, rescaling values so that the sum of each dimension across all vectors in the set equals one or some other constant. In order to effectively solve this issue, normalization must be done to every dimension in the vector, but it is unacceptable for cases with multiple dimensions of the same type where the difference in scale is intentional and significant, such as the CPU times of a very small and a very large phase.

Performance Impact The formulations of the objective functions defined in Equations (3.1) and (3.2) assume that minimizing the maximum value in a dimension (any dimension in Equation (3.1), the largest dimension in Equation (3.2)) will improve the performance of the application. However, this is not the case in general; take a load vector in which one dimension represents the memory footprint of an object. Reducing the maximum value of this dimension will reduce the amount of memory occupied by that PE, but will likely not improve application performance (ignoring cache effects). For such dimensions, it suffices to ensure that the maximum value merely does not exceed the available capacity.

6.4 CONSTRAINT-BASED LOAD BALANCING

We introduce *constraint-based load balancing* as a solution to these problems. In constraint-based load balancing, rather than minimizing holistically across the whole load vector, we alter the optimization problem, instead only aiming to find a mapping that minimizes across *a subset* of the dimensions such that the other dimensions remain below some specified thresholds.

We illustrate the utility of constraint-based load balancing by examining the case of balancing with a memory utilization constraint: A good assignment of objects to PEs in terms of the balance of computational load may be infeasible if the objects assigned to one of the nodes cumulatively require more memory than is available on the node, as it would lead to a capacity induced memory allocation failure.

Although one can easily keep track of the computational load and the memory footprint of each object, strategies based on Equation (3.1) do not make sense here since adding up time units and memory units is clearly futile. A similar argument disqualifies the utility of Equation (3.2) for this situation as well. One way out of this conundrum is to recognize the total memory capacity of each load as a constraint, and then to use a constraint-aware load balancing strategy to maintain this constraint while balancing.

We develop a new variant of our rk-d load balancing algorithm, modified to add support for constrained load balancing, called rk-d-constraint. The design of this load balancer is provided in Algorithm 6.1, with changes from the original version highlighted in yellow.

This new load balancing strategy accepts an additional parameter, a list of constraint

Algorithm 6.1: rk-d Constraint Algorithm

```
Input: Set of objects $O$, set of PEs $P$, list of constraints $C$
Output: New mapping in $sol$
1  $tree \leftarrow MakeTree(P)$;
2  forall $o \in sorted(O)$ do
3      $p_{min}, \_ \leftarrow FindMinNormPEConstrain(tree, o, C)$;
4      $tree \leftarrow Remove(tree, p_{min})$;
5      $sol.Assign(o, p_{min})$;
6      $tree \leftarrow Add(tree, p_{min})$;
7  end
8  Function FindMinNormPEConstrain($tree$, $o$, $C$, $bounds = \langle 0, \ldots, 0 \rangle$):
9      $data_{min} \leftarrow tree.data[: |C|]$;              /* Part of the vector to minimize */
10     $data_{con} \leftarrow tree.data[|C| :]$;             /* Part of the vector to constrain */
11     if $tree.left \neq NULL$ then
12         $p_{best} \leftarrow FindMinNormPEConstrain(tree.left, o, C, bounds)$;
13     end
14     if $data_{con}[i] \leq C[i]\ \forall i \in \{1, \ldots, |C|\} \wedge \|data_{min} + o[: |C|]\| < norm_{best}$ then
15         $norm_{best} \leftarrow \|tree.data + o\|$;
16         $p_{best} \leftarrow tree.data$;
17     end
18     if $tree.right \neq NULL$ then
19         $oldBound \leftarrow bounds[tree.dim]$;
20         $bounds[tree.dim] \leftarrow tree.data[tree.dim]$;
21         $bounds_{min} \leftarrow bounds[: |C|]$;
22         $bounds_{con} \leftarrow bounds[|C| :]$;
23         if $bounds_{con}[i] \leq C[i]\ \forall i \in \{1, \ldots, |C|\} \wedge \|bounds_{min} + o[: |C|]\| < norm_{best}$
            then
24             $p_{best} \leftarrow FindMinNormPEConstrain(tree.right, o, C, bounds)$;
25         end
26         $bounds[tree.dim] \leftarrow oldBound$;
27     end
28     return $p_{best}$
```

values C. In this implementation, without loss of generality, we assume that the dimensions to be constrained are at the end of the load vector, ergo in the last $|C|$ positions. We split the load vector into two parts, the first containing dimensions to be minimized and the second containing dimensions to be constrained. Only the part used for minimization is used for calculating norms. When performing search, a PE is only considered a viable candidate if the post-placement norm is less than the current best norm and the constrained portion post-placement load vector does not violate any of the constraints in C. The same applies when deciding whether or not to prune or search the right child of a node in the tree; we use the space partitioning properties of the tree to determine a minimum bound vector *bounds* for all of the PEs contained in the right subtree and decide to pursue it only if the hypothetical placement of the object onto *bounds* results in a better norm than the current best and satisfies all of the constraints in C.

This scheme guarantees that the resulting mapping satisfies all provided constraints. This may not be possible if C overconstrains the problem or if the search algorithm happens to get into a state where no satisfactory PEs remain, even if the problem itself is feasible. We intend to study how to make the search process more robust against this in the future.

6.5 RESULTS

6.5.1 Practical Evaluation

We evaluate the utility of the constraint-aware strategy in practice via testing a memory constrained load balancing scenario with 3D stencil mini-app on the KNL partition of Stampede2[6] with the `mpi-linux-x86_64` build of Charm++.

The program consists of two type of objects: compute-heavy/memory-light and compute-light/memory-heavy. The results in the figure are from a 4 node run with 28 objects, 14 compute-heavy/memory-light, with a computational load sampled from $\mu = 12, \sigma = 4$ and a memory footprint sampled from $\mu = 1\,\text{GB}, \sigma = 100\,\text{B}$ and 12 compute-light/memory-heavy, with a computational load sampled from $\mu = 1, \sigma = 0.1$ and a memory footprint sampled from $\mu = 12\,\text{GB}, \sigma = 100\,\text{B}$. Each node of Stampede2 has 96 GB of memory, but due to the fact that the asynchrony of migration means that incoming objects may arrive before outgoing objects are removed after load balancing occurs, the memory constraint provided to rk-d-constraint was 45 GB, meaning that the physical constraint will not be violated even in the worst case (the remaining 6 GB is reserved for the operating system, daemons, and other

[6]https://www.tacc.utexas.edu/systems/stampede2

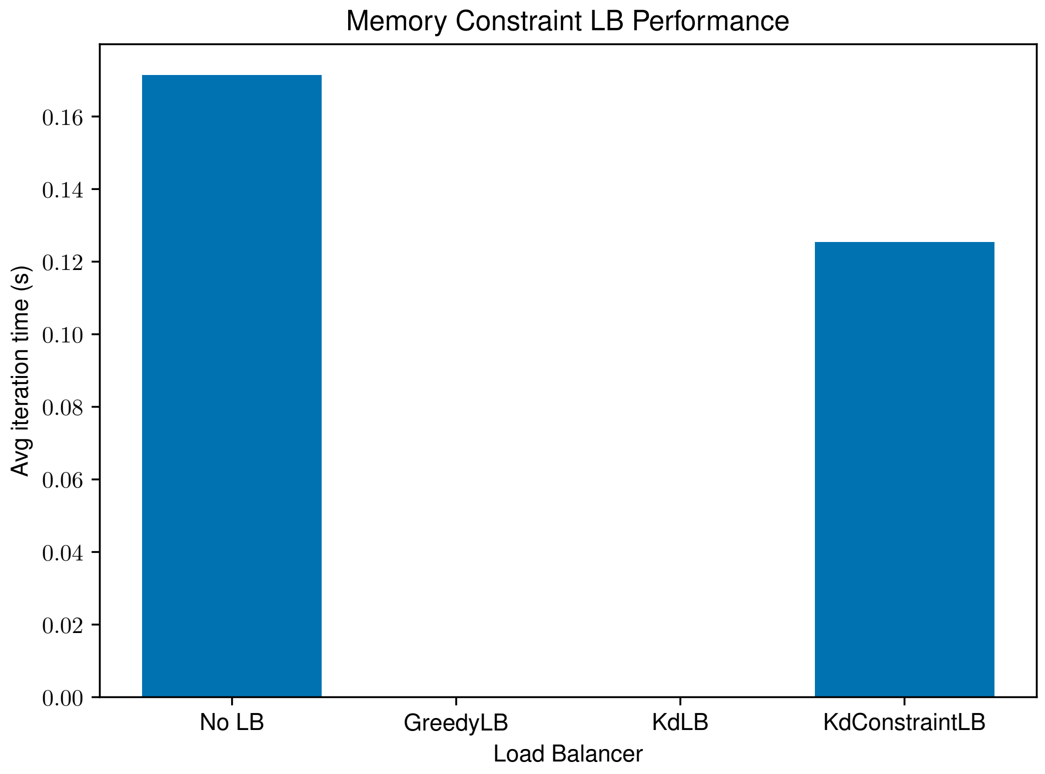


Figure 6.1: Performance Results for Memory Constrained Scenario

system services). An example configuration file showing how to express a constraint to the load balancer is provided in Listing 6.1. Note that the constraint has no expressed units or semantics in the configuration file, it is currently assumed that the user passes in the values to be constrained as the last entries in the load vector with the same units specified in the configuration file. We plan to create a more structured and expressive syntax for specifying constraints in the future.

Figure 6.1 shows the performance results of this experiment in terms of average iteration time. Only the run with no load balancing and the run using the rk-d-constraint strategy

```
{
  "tree": "PE_Root",
  "root":
  {
    "pe": 0,
    "strategies": ["rKdConstraint"],
    "rKdConstraint": {"constraints": [45] }
  }
}
```

Listing 6.1: Example Constraint Configuration File

completed successfully. The rk-d-constraint strategy provides a 1.37x speedup over the no LB case. The other runs, using the greedy and rk-d strategies, resulted in mappings that attempted to allocate memory in excess of the available capacity and crashed. These crashes are somewhat non-deterministic, as noise or other minor variances in the actual measured load may by chance lead to a mapping that does not violate the memory constraint.

CHAPTER 7: SCALING VECTOR LOAD BALANCING

In previous chapters, we have focused on the improvements in load balancing quality that vector load balancing can provide. However, another important consideration is the runtime of the load balancing strategy itself, particularly in scenarios where load balancing is performed several times during the execution of an application, as is the norm with dynamic load balancing in HPC.

In this chapter, we discuss the performance characteristics of load balancing algorithms, the performance differences between scalar and vector load balancing, our approach to and results from creating a load balancing strategies that optimize for performance, and the tradeoffs between performance and balance quality.

7.1 OVERVIEW

Vector load balancing is innately a more complex problem than scalar load balancing. Scalar load balancing itself is already an NP-complete problem [83] when solving for the optimal solution, and vector load balancing is NP-hard [20]; thus, for practical applications, we focus on using heuristics or approximation algorithms to perform load balancing.

As compared to scalar LB, the vector case has the additional difficulties of having load data be larger in size (as each object now has d, the number of dimensions, load entries per object instead of 1) and, more significantly, having to organize, compare, and search through a multidimensional space and optimize for a multidimensional objective function when performing load balancing.

In the scalar case, strategies can exploit the fact that scalar values can be totally ordered to use optimized data structures to quickly get the minimum or maximum object or PE, a common operation in load balancing strategies. In the vector case, load vectors can only be partially ordered in general, and thus there is usually no globally minimum or maximum object or PE, but rather a set of Pareto optimal candidate objects or PEs. Finding this Pareto frontier in the first place can be costly, and even once it is found the strategy must still search through the frontier to find the best candidate. Our implementations use what our testing identified to be empirically the best performing specialized data structures for each strategy, but even so, the innate difficulty of the problem means that runtime is still relatively long.

Further, supercomputers continue to grow in size as measured by core count as we grow into the nascent exascale era and inexorably march toward larger and faster machines. For example, over the past decade, the highest core count for a machine in the top ten of the TOP500[7] list has gone from 1,572,864 for Sequoia in November 2012 to 10,649,600 for Sunway TaihuLight in November 2022, an increase of approximately an order of magnitude. As the number of PEs used in a job increases, load balancing correspondingly takes more time. We can break this increase in time down into two components: managing the data structure used to store PEs and searching the PE search space for deciding an object placement. In the scalar case, when using the common paradigm of using a heap to store PE data, at each step the management cost increases by a logarithmic factor of the increase in the number of PEs, while the search cost remains constant since accessing the minimum element of a heap is $O(1)$. However, the cost increase for vector load balancing can be particularly acute due to the lack of total ordering described above. Assuming the use of a balanced k-d tree to store PE data, at each step the management cost increases by a logarithmic factor of the increase in PE count, but the search cost may increase by a linear factor in the worst case where every PE is in the Pareto frontier.

Exacerbating this issue of performance is the phenomenon called the *curse of dimensionality*. As the number of dimensions of the problem increases, data tend toward increasing sparsity because the hypervolume of the space grows very quickly. Additionally, due to this sparsity, the runtime of performing spatial point queries such as searching for the nearest neighbor of a point grows with the dimension; the distances between points are likely to be similar and more a larger portion of the candidate space must be searched before identifying a candidate that satisfies the search criteria.

Due to the combined impact of these factors, the execution times of the vector load balancing strategies explored in this thesis can sometimes be multiple orders of magnitude longer than those of the extant scalar load balancing strategies in Charm++. Even with this additional strategy cost, vector load balancing can still show benefit with a variety of applications in practice, as demonstrated in previous chapters. However, this runtime cost can limit scalability or useful load balancing frequency, limiting the utility of vector LB for large runs or very dynamic applications. The performance benefit provided by load balancing should not be eclipsed by the cost of running load balancing itself, otherwise it is preferable to not use load balancing at all.

An additional factor with regards to scalability is the use of parallel execution. In previous

[7]https://www.top500.org/

chapters we have discussed the design and applications of load balancing strategies and the quality of their mapping decisions, but not how and where load balancers themselves are actually executed. The simplest way to execute a load balancer is using a *centralized* scheme, in which all of the PE and object data is gathered onto a single PE, where the load balancing strategy is then executed serially. Due to Amdahl's Law, for an otherwise scalable application, this serial execution becomes an increasingly large portion of runtime as core count increases. This is worsened by the fact that the runtime of the strategy grows with core count, meaning this issue afflicts both strong scaling and weak scaling. In order to be suitable for large scale parallel applications, load balancing strategies should also run in parallel as much as possible.

Taken together, all of this means that the performance and parallelism of vector load balancing is a critical factor in its utility for applications. In the remainder of this chapter, we describe the baseline performance of our vector load balancing strategies and the modifications and optimizations we have developed to improve strategy performance.

In practice, load balancing time consists of three components:

1. Statistics collection, gathering load information and preparing it for balancing
2. Strategy execution, the runtime of the actual balancing algorithm
3. Object migration, the movement of objects to apply the new mapping

In this chapter, we focus on the analysis and optimization of the time spent in strategy execution, as this component is where the largest disparity between similarly structured scalar and vector load balancers arises. Collection time is invariant for load balancers that run using the same execution scheme (e.g. centralized) and require the same input data (collection can take longer if an LB needs the communication graph, for example). Migration time can vary depending on the design of the strategy, some are specifically designed to minimize migrations for scenarios where moving objects is costly. However, none of the load balancers examined in this chapter take migrations into account, so we ignore migration time in our analysis. We leave the consideration of statistics collection and migration cost as future work, pending the inclusion of distributed or refinement-based vector load balancers.

7.2 EXPERIMENTAL SETUP

All experiments in this chapter, unless otherwise noted, were performed on a dedicated ARM Ampere A1 CPU at a frequency of 3 GHz. Code was compiled with `gcc 11.3.0` using `-O3` and the `-ffast-math` flag, as we do not require strict conformance with IEEE 754 floating-point specifications within the load balancing strategies.

7.3 BASELINE PERFORMANCE

We examine the baseline performance of the scalar greedy strategy, vector greedy strategy, vector rk-d norm-based strategy, and the vector METIS strategy. Full details of the different strategies are provided in Section 3.2. All strategies are executed in a centralized manner.

The experiments presented here range from 8 to 16,384 PEs and use a fixed overdecomposition ratio of eight (meaning there are eight objects per PE), mimicking a weak scaling test. Both axes are given in log scale. Plotted values are the median runtimes of one hundred runs with different RNG seeds for each configuration. Two plots are shown for each load pattern, the left one shows runtime and the right one shows runtime normalized to that of the scalar greedy strategy.

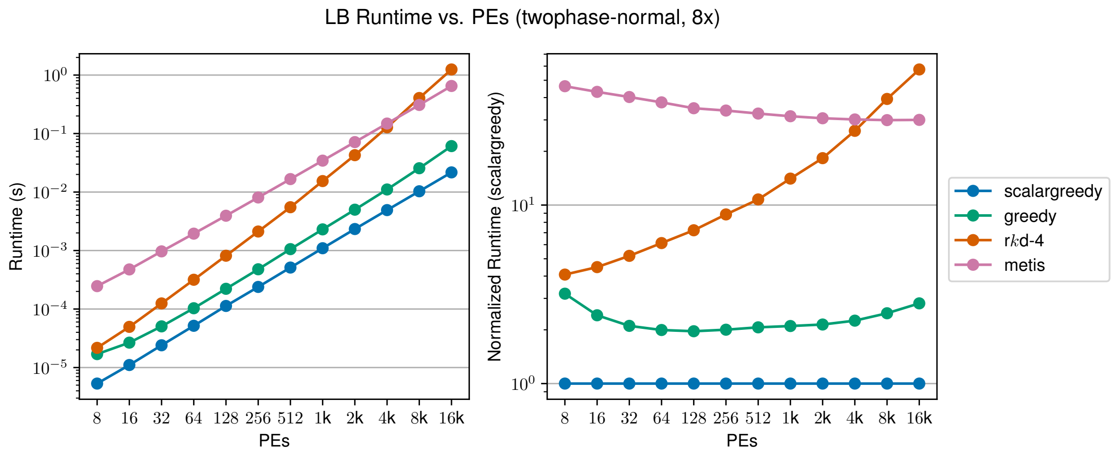
The baseline performance of the load balancing strategies with normally distributed load vectors of dimension two, four, and six is shown in Figure 7.1. Each dimension of the load vector for every object is sampled from a normal distribution with $\mu = 10, \sigma = 3$.

The baseline performance of the load balancing strategies with alternating exponentially and normally distributed load vectors of dimension two, four, and six is shown in Figure 7.2. Each exponential dimension of the load vector (i.e. first dimension in the 2D case, first and third in the 4D case, and first, third, and fifth in the 6D case) is sampled from an exponential distribution with $\lambda = 0.15$ and each normal dimension of the load vector is sampled from a normal distribution with $\mu = 10, \sigma = 3$.

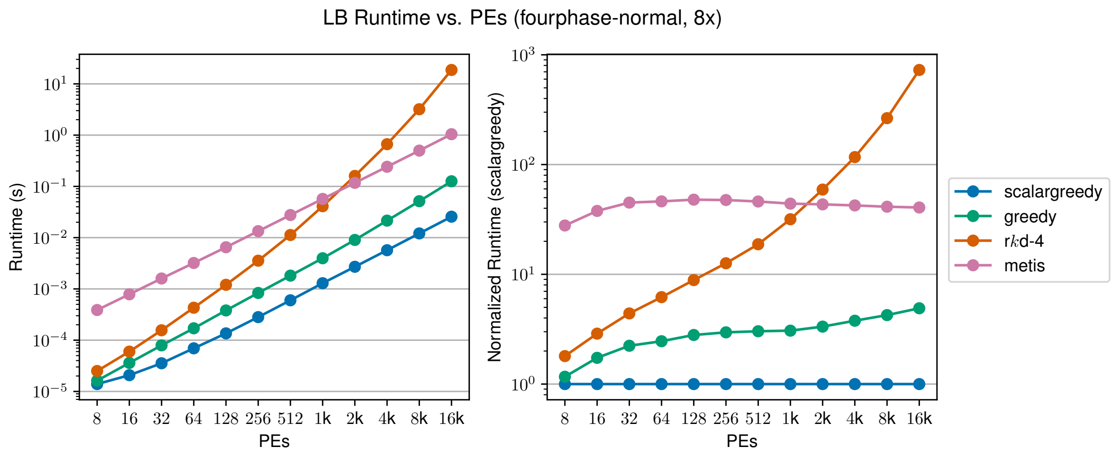
Finally, baseline performance of the strategies with a three dimensional load problem is shown in Figure 7.3. In this case, the first dimension of each object has an 80% chance of being sampled from a normal distribution with $\mu = 1, \sigma = 0.1$ and a 20% chance of being sampled from a normal distribution with $\mu = 5, \sigma = 0.1$. The second and third dimensions are sampled from an exponential distribution with $\lambda = 0.1$.

7.3.1 Analysis

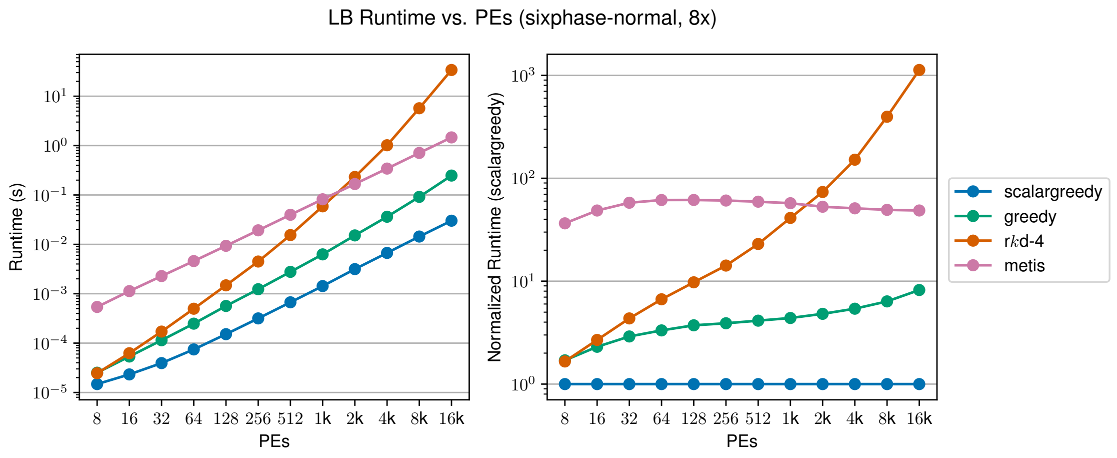
Across all of the different configurations, the scalar greedy strategy is the fastest in general, as expected, since it is solving an inherently simpler problem than the other strategies. As discussed in previous chapters, the resulting mappings of scalar load balancing for vector problems are often poor, so we use the scalar greedy baseline not as a viable alternative



(a) Two Dimensional Baseline

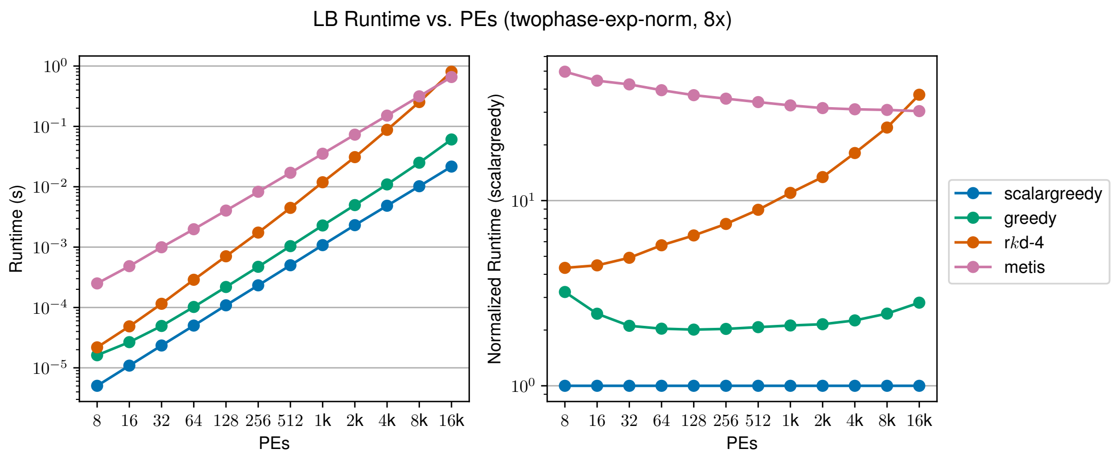


(b) Four Dimensional Baseline

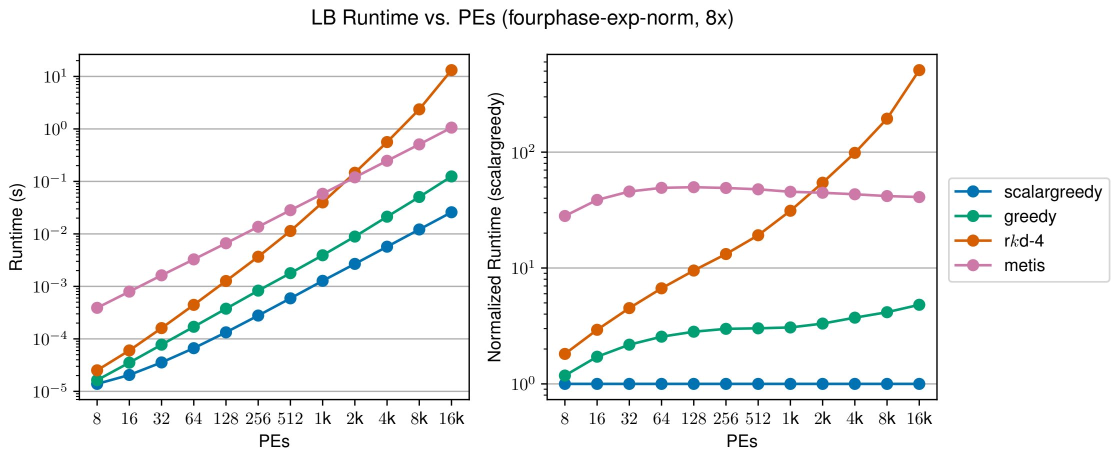


(c) Six Dimensional Baseline

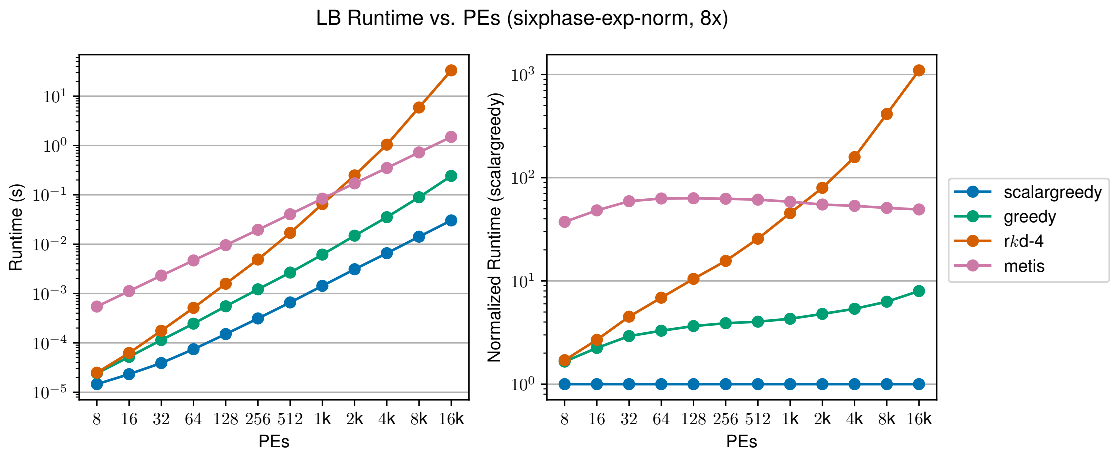
Figure 7.1: Baseline Performance with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional Baseline



(b) Four Dimensional Baseline



(c) Six Dimensional Baseline

Figure 7.2: Baseline Performance with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

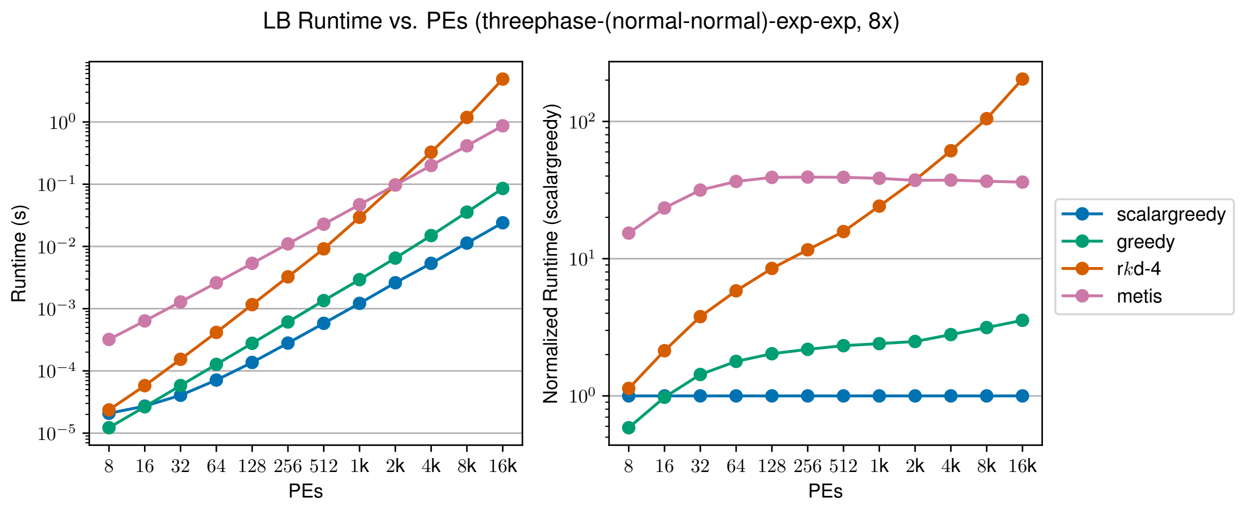


Figure 7.3: Baseline Performance with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

for vector LB, but as a comparative upper bound on the maximum performance one can reasonably achieve for LB in general. Scalar greedy runs in $O(n \log p)$ time in the worst case, where n is the number of objects and p is the number of PEs. Regardless of the dimension of the problem, scalar greedy treats everything as one dimensional, so increasing dimension does not affect its performance, modulo the overheads of bookkeeping and data movement of the larger load vectors.

The vector greedy strategy is the next fastest, slower than scalar greedy but faster than every other vector strategy for every configuration. This is expected, since it is essentially the same as the scalar greedy strategy, except selecting from and updating multiple PE heaps, one per dimension, while computing object placements. Vector greedy runs in $O(dn \log p)$ time, where d is the dimensionality of the load balancing problem. The performance gulf to the scalar greedy strategy grows as d increases, as each additional dimension brings with it an additional heap to update. While this provides scalable performance approximately within a factor of d of the scalar greedy strategy, the quality of the resulting mapping is often subpar as compared to the more holistic norm and partitioning based vector LB strategies.

The METIS strategy is the slowest strategy at small scales for all configurations. However, as we scale the number of PEs, it matches and then outperforms the rk-d strategies. The crossover point where METIS begins to beat the best of the rk-d strategies happens at smaller scales as the number of dimensions in the problem increases; in Figure 7.1, we see that the crossover point for the 2D case is 16k PEs and that it shrinks to 2k PEs for the 4D case. As indicated by the slightly decreasing values of the METIS entries in the normalized runtime plots for all configurations ≥ 128 PEs, the runtime for the METIS strategy scales at approximately the same rate as scalar greedy, but from a larger starting point, staying within a factor of 100x. We use the multi-constraint multilevel recursive bisection scheme of METIS in our LB strategy; internally, the coarsening, bipartitioning, and refinement performed by METIS [30] run with a complexity of $O(dn \log p)$. In practice, the impact of changing dimension is rather small: similar to vector greedy, METIS uses a heap per dimension, but unlike vector greedy, each element is only in a single heap, that of the largest dimension in its load vector. This provides METIS with scalable performance, albeit one much slower than the greedy strategies, with runtime on the order of about one second for the 16,384 PE case across all tested load distributions. However, the quality of the mapping produced by METIS can be somewhat inconsistent, in our testing, it generally provides good results in cases where each dimension is similarly distributed, but it can atrophy in cases where dimensions have different distributions, particularly at large scale.

While the norm-based strategies using rk-d have good performance for small PE counts, their scalability is quite poor. For every tested load configuration, the rk-d strategies are either the slowest or tied for the slowest at the largest tested scale of 16,384 PEs. Also notable is their degree of performance degradation as dimensionality increases; as shown in Table 7.1 at 16,384 PEs, the four dimensional case is over an order of magnitude slower than the two dimensional case for both the normal and Pareto variants, and the six dimensional case over 40x and 300x slower than the two dimensional case for the normal and Pareto variant, respectively. No other tested strategy has close to the same dropoff in performance with increasing dimensionality as these do.

A final takeaway from this set of results is that different load distributions have less impact on performance than the dimensionality of the problem. Using the timings shown in Figures 7.1 and 7.2, we see the same trends in scaling performance for each LB and almost identical orderings of strategies when comparing between the data for the same dimensionality at every PE count (the sole exception being in two dimensions with METIS outperforming rk-d Pareto at 16k PEs for the normally distributed case but underperforming it for the (exponentially, normally) distributed case).

Since the rk-d strategies result in the most resilient solutions in terms of quality, providing reasonable mappings across all tested load distributions, we would like to improve the performance of these strategies to make them tenable for use at large scale and with high dimensional load vectors. In the remainder of this chapter, we discuss our efforts to solve this problem and the results of these optimization efforts on the rk-d strategy.

7.4 PARETO SEARCH

7.4.1 Motivation

In order to place each object, the norm-based strategy searches through the PEs to find a PE with the minimum post-object placement load vector norm. While the norm-minimizing PE changes depending on the load vector of the object, it is always a member of the Pareto frontier of the set of PEs, as shown in Lemma 7.1.

Lemma 7.1. A norm-minimizing PE pe_{min} for an object o is a member of the Pareto frontier F of the set of PEs for any p-norm $\|\vec{x}\|_p$ with $1 \leq p < \infty$.

Proof. Recall that $\vec{l}_o, \vec{l}_{pe_{min}}, \vec{l}_{pe_{dom}} \in \mathbb{R}^d_{\geq 0}$. Suppose $pe_{min} \notin F$. Then pe_{min} is dominated by some other PE, pe_{dom}, meaning:

$$0 \leq \left(\vec{l}_{pe_{dom}}\right)_i \leq \left(\vec{l}_{pe_{min}}\right)_i \qquad \forall i \in \{1, \ldots, d\} \tag{7.1}$$

$$0 \leq \left(\vec{l}_{pe_{dom}}\right)_i < \left(\vec{l}_{pe_{min}}\right)_i \qquad \exists i \in \{1, \ldots, d\} \tag{7.2}$$

Then, calculating the post-placement load vector for o:

$$0 \leq \left(\vec{l}_{pe_{dom}} + \vec{l}_o\right)_i \leq \left(\vec{l}_{pe_{min}} + \vec{l}_o\right)_i \qquad \forall i \in \{1, \ldots, d\} \tag{7.3}$$

$$0 \leq \left(\vec{l}_{pe_{dom}} + \vec{l}_o\right)_i < \left(\vec{l}_{pe_{min}} + \vec{l}_o\right)_i \qquad \exists i \in \{1, \ldots, d\} \tag{7.4}$$

But:

$$\sum_{i \in \{1,\ldots,d\}} (\vec{l}_{pe_{dom}} + \vec{l}_o)_i < \sum_{i \in \{1,\ldots,d\}} (\vec{l}_{pe_{min}} + \vec{l}_o)_i \tag{7.5}$$

$$\left(\sum_{i \in \{1,\ldots,d\}} (\vec{l}_{pe_{dom}} + \vec{l}_o)_i^p\right)^{1/p} < \left(\sum_{i \in \{1,\ldots,d\}} (\vec{l}_{pe_{min}} + \vec{l}_o)_i^p\right)^{1/p} \tag{7.6}$$

$$\left\|\vec{l}_{pe_{dom}} + \vec{l}_o\right\|_p < \left\|\vec{l}_{pe_{min}} + \vec{l}_o\right\|_p \tag{7.7}$$

Meaning pe_{min} is not a norm-minimizing PE for o. Thus, our supposition is false and $pe_{min} \in F$. QED.

Remark 7.1. For the ∞-norm case, since $\|\vec{x}\|_\infty = \max(\vec{x}_1, \ldots, \vec{x}_d)$, a PE $pe_{out} \notin F$ may have the same post-placement norm as the PE $pe_{in} \in F$ that dominates them. However,

pe_{out} cannot have a post-placement norm less than pe_{in} since:

$$0 \leq \left(\vec{l}_{pe_{in}} + \vec{l_o}\right)_i \leq \left(\vec{l}_{pe_{out}} + \vec{l_o}\right)_i \qquad \forall i \in \{1, \ldots, d\} \tag{7.8}$$

$$\left\|\vec{l}_{pe_{in}} + \vec{l_o}\right\|_\infty \leq \left\|\vec{l}_{pe_{out}} + \vec{l_o}\right\|_\infty \tag{7.9}$$

Thus, searching only through F is sufficient to find a norm-minimizing PE. QED.

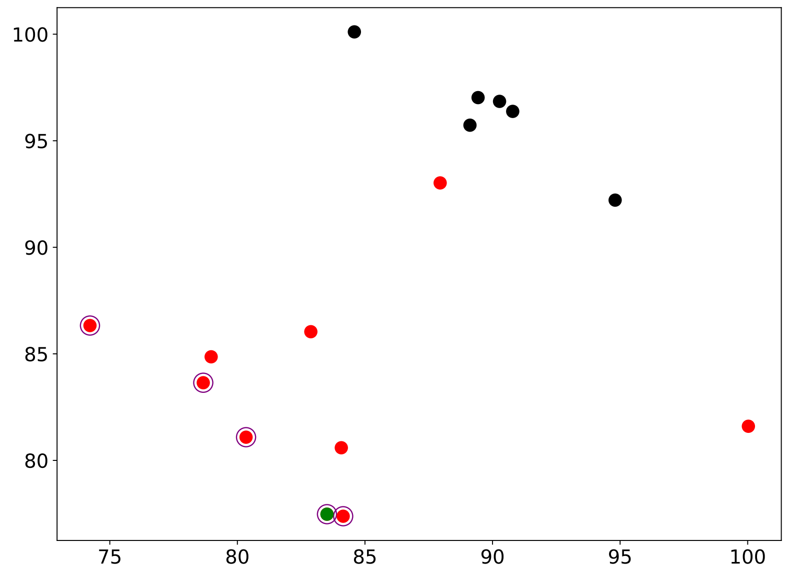


Figure 7.4: Iteration of rk-d Strategy Search Process

However, the rk-d strategy may search through elements outside of the Pareto frontier, as it has no knowledge of which PEs comprise the frontier. Figure 7.4 shows a representative search iteration of the rk-d strategy for a two dimensional problem with 16 PEs. The x-axis represents the first dimension of the load vector, the y-axis the second dimension, and each point a PE, plotted based on the state of its current load vector. In this particular search iteration, the algorithm is looking for a PE to place an object with load vector $\langle 10.09, 15.01\rangle$. Black points represent PEs that were skipped during the search, ruled out by being in a region of the tree eliminated by some discovered candidate. Red and green points represent PEs that were searched, red points being eventually discarded and the green point being the final returned solution. Points circled in purple are the members of the Pareto frontier for the given state of the PEs. There are only five PEs in the Pareto frontier, yet the algorithm searches through a total of ten PEs.

7.4.2 Implementation

To exploit this observation that merely searching through the Pareto frontier is sufficient to find a norm-minimizing PE, we created a new variant load balancing strategy built atop the r*k*-d strategy called *rk-d Pareto*. The idea behind the Pareto variant is to reduce the space traversed by the search algorithm when placing each object by only searching through the Pareto frontier, with the supposition that the cost of maintaining the Pareto frontier would be less than the benefit from the smaller search space. Mapping quality does not change, as the resulting mapping of this variant is the same as that of the regular r*k*-d strategy (excepting rare situations where norms are tied, unlikely to arise in practice); the search logic and objective are identical to the original version.

Algorithm 7.1: r*k*-d Pareto Algorithm

```
Input: Set of objects $O$, set of PEs $P$
Output: New mapping in $sol$
1  $T \leftarrow MakeTree(P)$;
2  $F \leftarrow MakeFrontier(T)$;
3  forall $o \in sorted(O)$ do
4      $p_{min} \leftarrow FindMinNormPE(F, o)$;    /* Search only through frontier F */
5      $T \leftarrow Remove(T, p_{min})$;
6      $F \leftarrow Remove(F, p_{min})$;
7      $nn \leftarrow NearestNeighbor(F, p_{min})$;
8      $sol.Assign(o, p_{min})$;
9      $T \leftarrow Add(T, p_{min})$;
10     $F \leftarrow UpdateFrontier(T, nn)$;
11 end
```

Algorithm 7.1 describes the execution of the r*k*-d Pareto strategy; highlighted lines indicate new additions to the baseline implementation. The frontier F is a second r*k*-d tree containing only the PEs on the Pareto frontier, such PEs are contained in both the frontier F and the overall tree T. For every object, the algorithm searches through the frontier for the PE p_{min} with minimum post-placement norm, then removes it from both the frontier and the overall tree. Then, after placing the object on p_{min}, pe_{min} gets added back to the overall tree, but not to the frontier, as it is now carrying the additional load from the object's load vector and is thus no longer likely to be a member of the frontier. Finally, the frontier is updated to include any new members, such as those that were dominated by the previous load of p_{min} and no other PEs in T (note that the updated p_{min} may be added back to the frontier in this step, particularly when objects with small loads are being placed).

To improve the performance of the update procedure, we find the nearest neighbor in the frontier $nn \in F$ to the pre-placement version of p_{min}. When updating the frontier, we first check if candidate PEs are dominated by nn before checking if they are dominated by any member of the frontier, as elements that were dominated by the now removed pre-placement p_{min} are more likely to be dominated by nn than any other member of F.

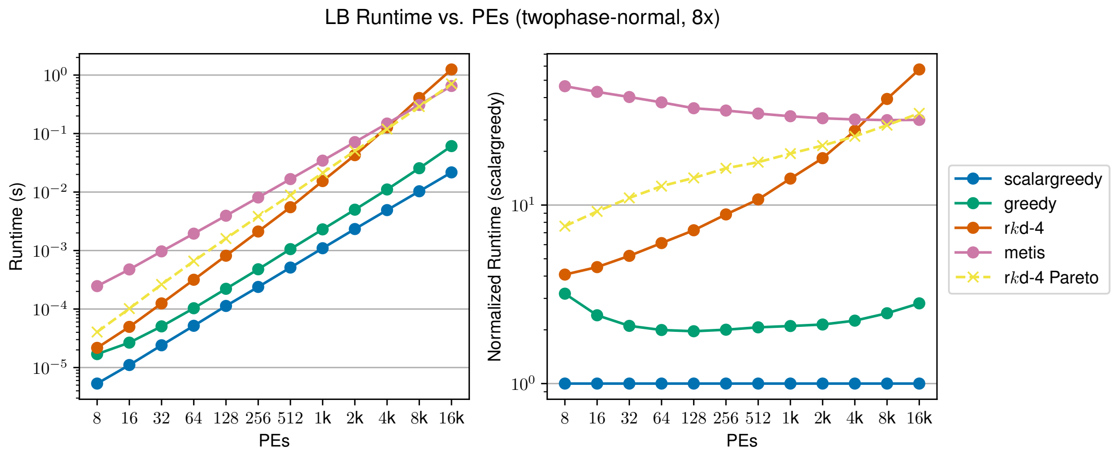
7.4.3 Performance Results

The performance results in Figures 7.5 to 7.7 show that the Pareto variant does not provide an improvement over the regular version in general. The Pareto variant only outperforms regular rk-d at large scale (4,096 PEs at minimum) and, further, only for small dimensions (two and three). For all other tested configurations, the Pareto variant is slower than its progenitor, at times by up to an order of magnitude. Based on the shape of its scaling curve at the high end, the Pareto variant will likely outperform regular rk-d for dimensionality greater than three at larger PE counts than were tested, but both strategies perform so poorly in the large scale regime that they are likely disqualifying for production use.

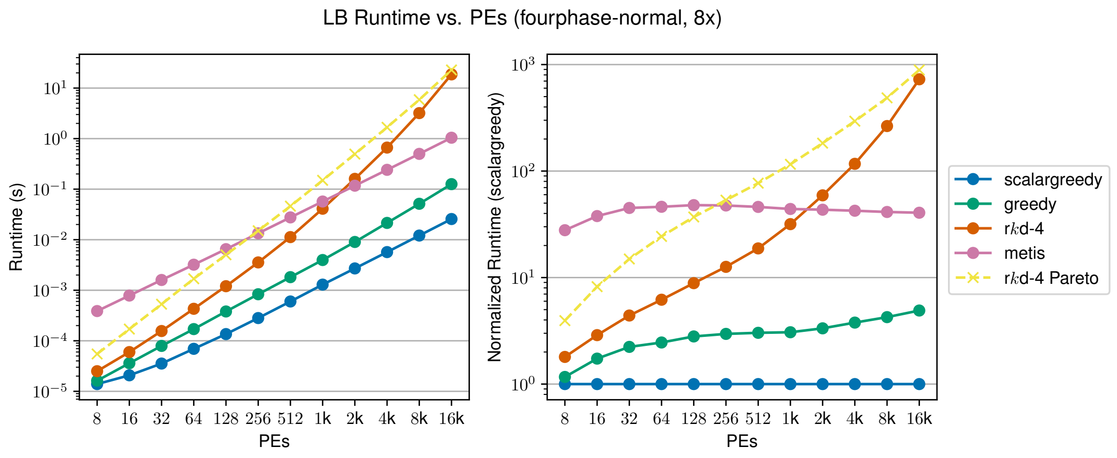
Table 7.1 compares the performance of the regular and Pareto rk-d strategies for load vectors of the (Exponentially, Normally) distributed case at 16,384 PEs with increasing dimension. As dimensionality increases, the Pareto variant goes from beating the regular version at two dimensions to being 5.12x slower at six dimensions. In absolute terms, spending over 100 seconds for strategy execution is untenable for many applications, particularly those with quickly changing load patterns requiring frequent load balancing. As a point of stark comparison, the scalar greedy strategy takes 0.03 s and vector greedy 0.24 s for the same configuration.

Strategy	**Runtime (s)**		
	2D	*4D*	*6D*
rk-d	0.81	13.19	33.22
rk-d Pareto	0.55	16.60	170.14

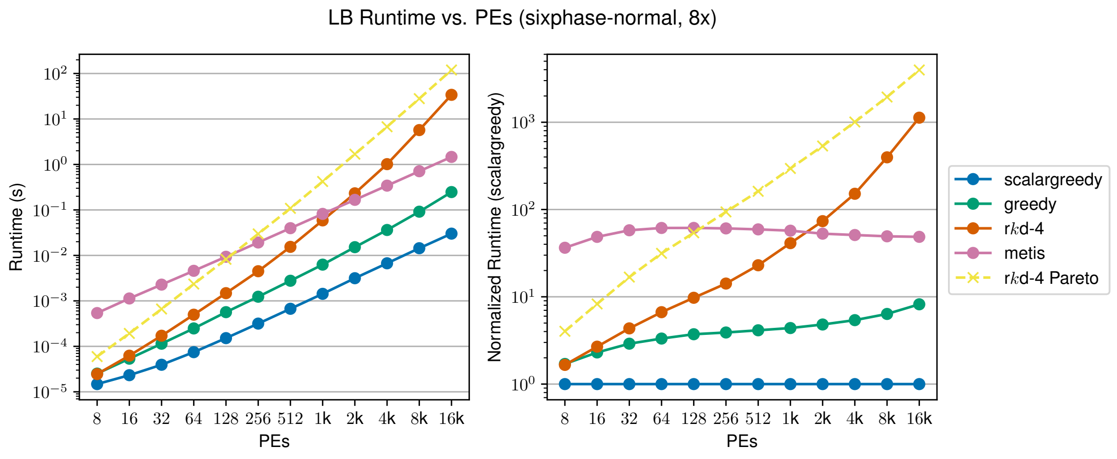
Table 7.1: rk-d Family Runtime at 16k PEs for (Exponentially, Normally) Distributed Load Vectors



(a) Two Dimensional Pareto

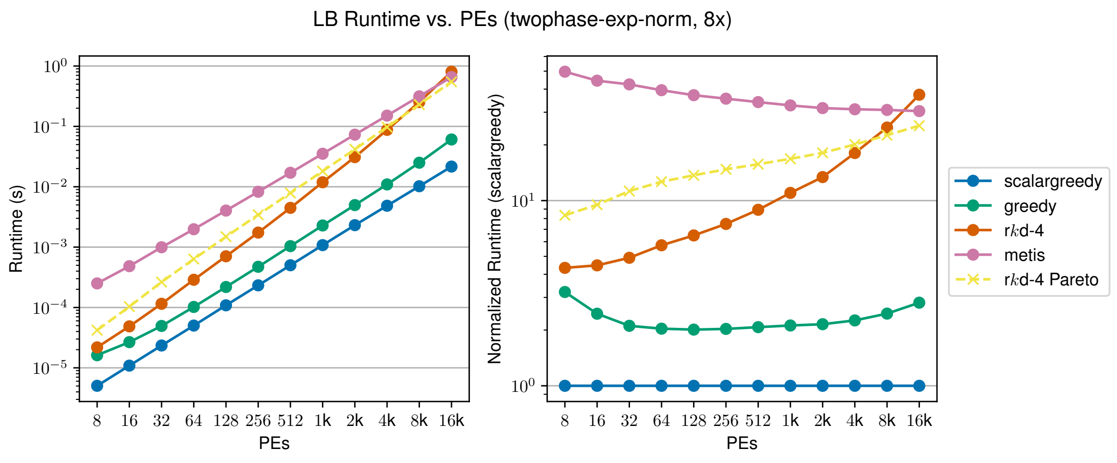


(b) Four Dimensional Pareto

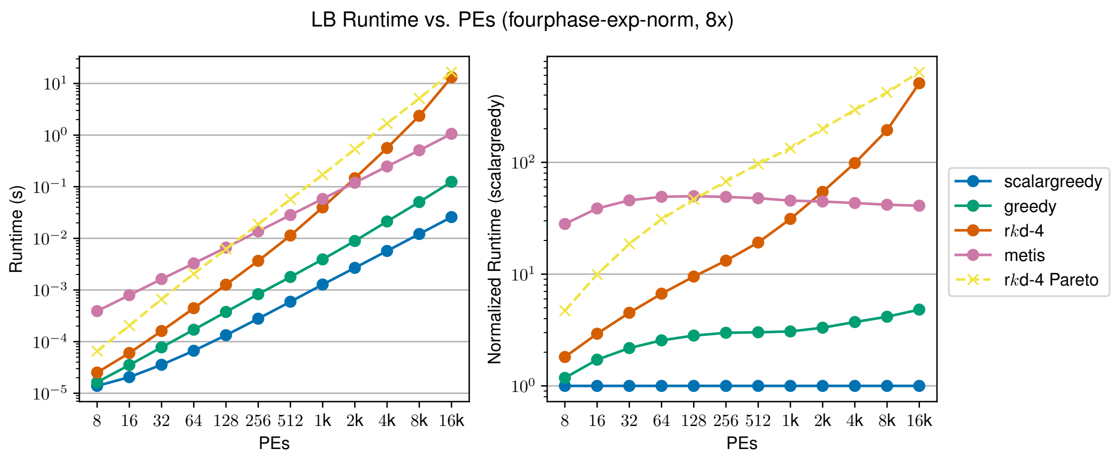


(c) Six Dimensional Pareto

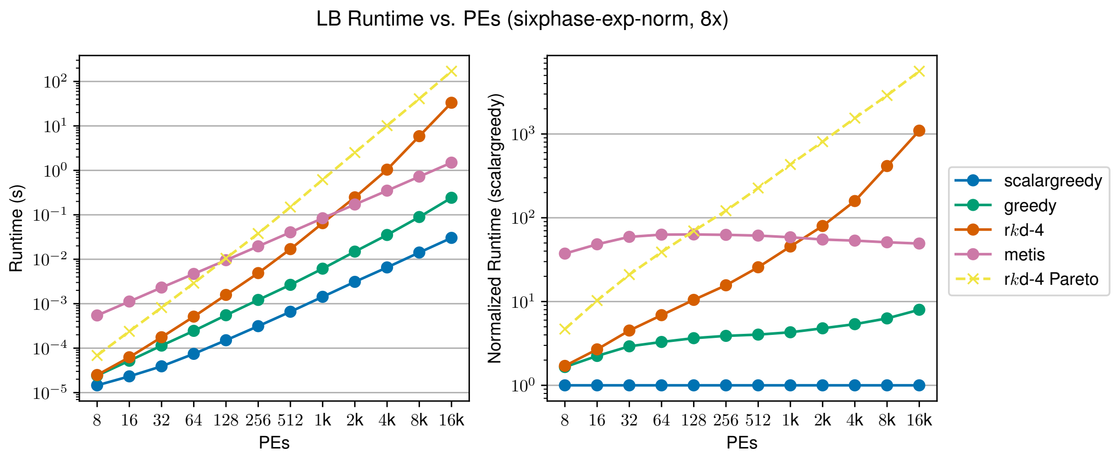
Figure 7.5: Pareto Performance with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional Pareto



(b) Four Dimensional Pareto



(c) Six Dimensional Pareto

Figure 7.6: Pareto Performance with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

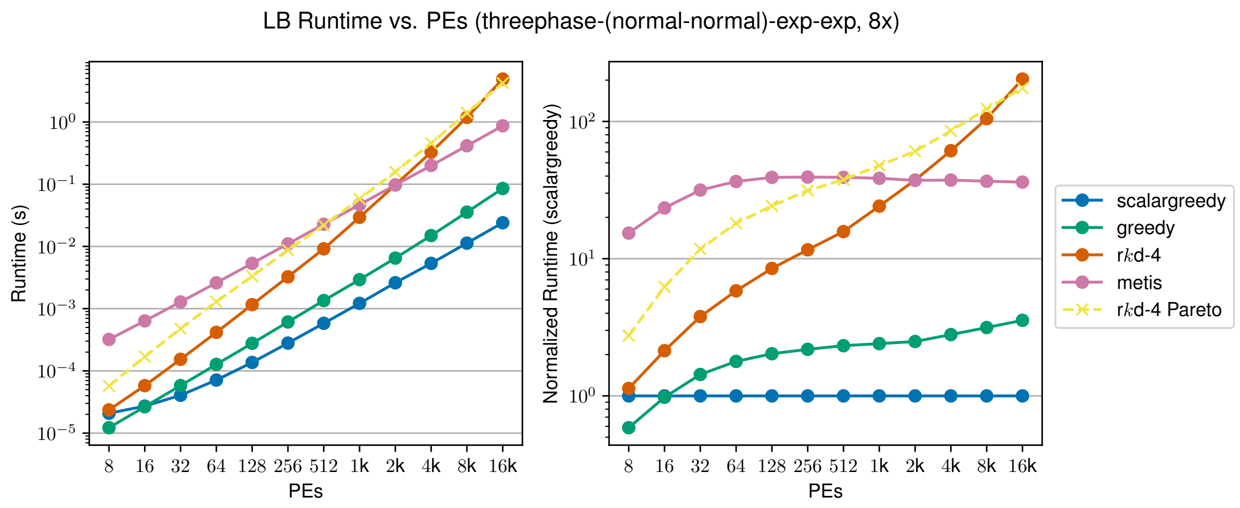


Figure 7.7: Pareto Performance with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

The Pareto variant is also slower than all other tested strategies across all tested configurations with the exception of METIS at low core counts or with low dimensionality.

Thus, while the Pareto variant does provide speedups over the baseline rk-d strategy in certain low dimensionality, large scale situations, the improvements are relatively meager and insufficient to allow for viable use at scale.

7.5 EARLY EXIT

7.5.1 Motivation

The Pareto variant attempts to provide performance improvement by reducing the search space while maintaining the same quality and solution as the baseline rk-d version, analogous to a lossless compression scheme that reduces space requirements without sacrificing fidelity to the input data. However, in load balancing, we can tolerate loss in the quality of solutions (in fact, as discussed in Section 7.1, every practical load balancing strategy is already using some heuristic or approximation as balancing is NP-hard). Thus, we can afford to use the load balancing equivalent of lossy compression, strategies that degrade quality as compared to the baseline but which have performance improvements beyond "lossless" schemes.

To that end, we designed another variant of rk-d called *early exit*. In the early exit variant, rather than thoroughly searching through the entire PE search space to determine each object placement, the search is terminated early, trading off quality for performance. The search termination criterion is that some fixed number of “suitable” PEs have been found, from which the “best” is used as the destination for the object being considered.

7.5.2 Implementation

Algorithm 7.2 describes the operation of the early exit strategy; the highlighted lines indicate the new additions as compared to the baseline.

We take a suitable solution to be a PE such that each dimension of the post-placement load vector of the PE does not exceed the global maximum value for that dimension across all PEs. Or, in other words, a solution that does not increase the makespan of the mapping. Of course, such a solution may not exist, in which case early termination does not occur and the search space is examined to its full extent. To determine if a PE satisfies this criterion, we maintain a maximum load vector $load_{max}$, against which we compare the load vectors of candidate PEs. Every time a suitable solution is found, the variable *limit* is decremented, and when it hits zero, the algorithm terminates and returns the best candidate PE found so far.

The value of *limit* corresponds to the number of suitable solutions that must be found before initiating an early exit. This value can be provided by the user; larger values perform a more thorough search of the space at the cost of decreased performance. Results for $limit = 1$, 5, and 10 are shown in the provided figures (denoted as Early-{1,5,10}).

In our examination of this variant, there are a few questions we would like to answer:

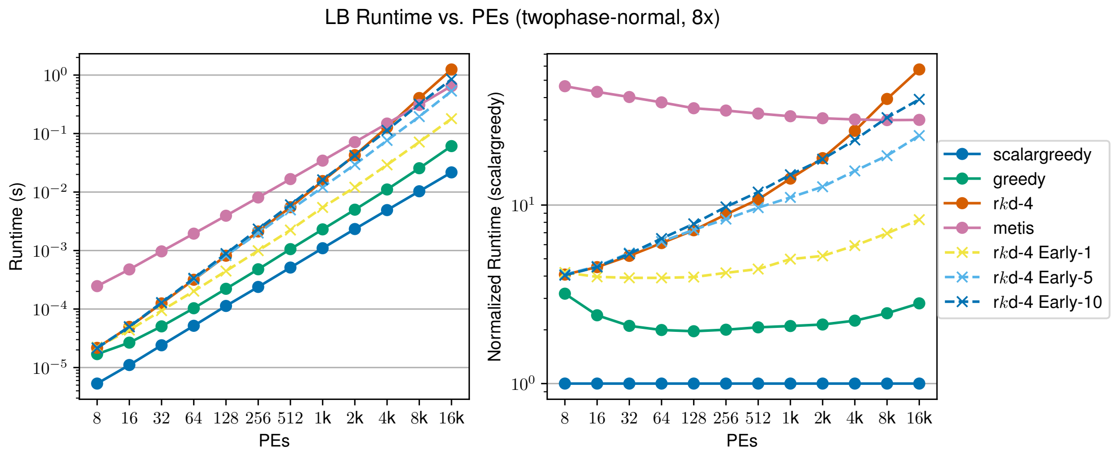
1. Is the suitability criterion too strict to provide a performance benefit?
2. How do different values of *limit* affect the scalability of the strategy?
3. Given that our suitability criterion limits early exit to cases where the makespan does not increase, is there any appreciable benefit in using $limit > 1$?

7.5.3 Performance

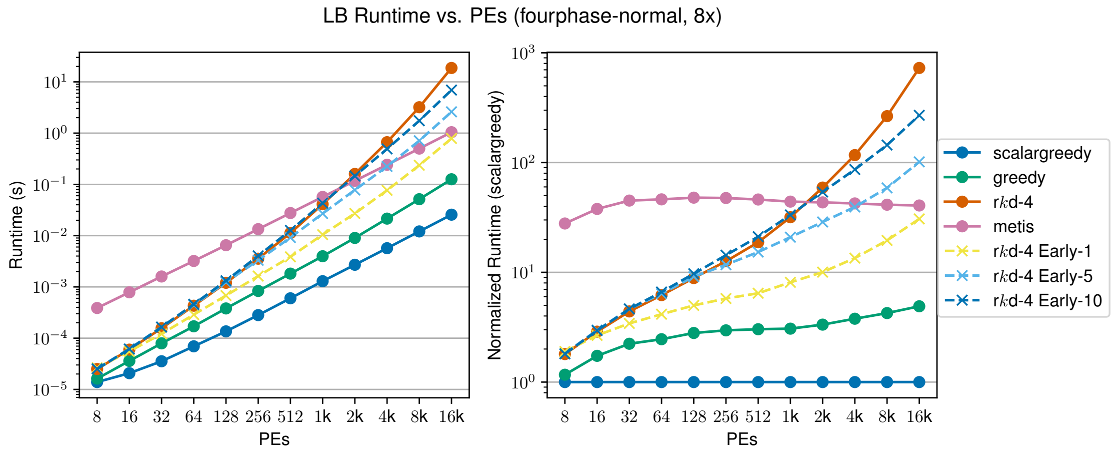
Figures 7.8 to 7.10 compare the runtime performance of the early exit variant against the baseline load balancers. The same pattern emerges in each of the tested load distributions and dimensionalities: the early exit strategy performs only negligibly differently from the standard rk-d strategy at eight PEs, but as the number of PEs increases, the early exit variant begins to outperform the regular version. This performance delta grows with scale, as one

Algorithm 7.2: rk-d Early Exit Algorithm

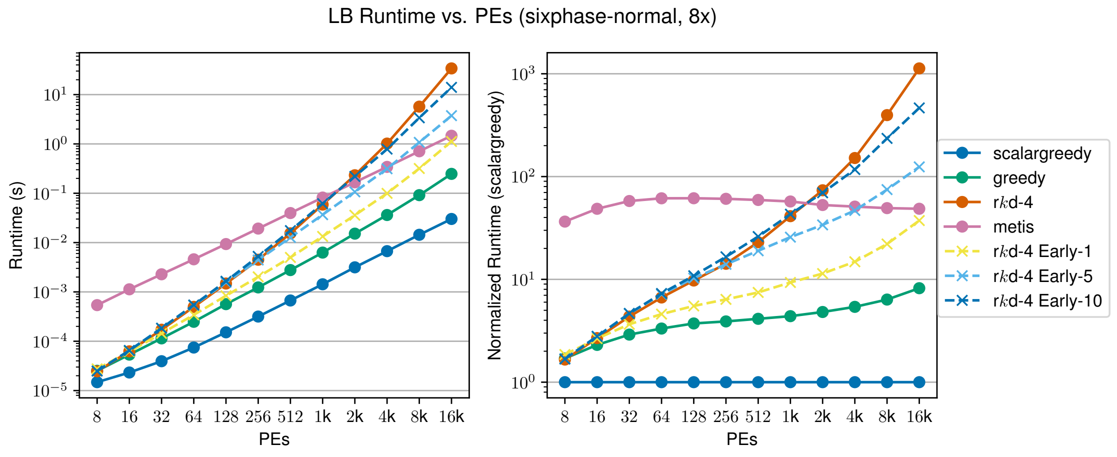
```
Input: Set of objects $O$, set of PEs $P$
Output: New mapping in $sol$
$load_{max} \leftarrow \langle 0, \ldots, 0 \rangle$;
$tree \leftarrow MakeTree(P)$;
forall $o \in sorted(O)$ do
    $p_{min}, \_ \leftarrow FindMinNormPEEarly(tree, o)$;
    $tree \leftarrow Remove(tree, p_{min})$;
    $sol.Assign(o, p_{min})$;
    $load_{max} \leftarrow \langle \max(load_{max}[i], p_{min}[i]) \rangle \ \forall i \in 1, \ldots, d$;
    $tree \leftarrow Add(tree, p_{min})$;
end
Function FindMinNormPEEarly($tree, o, limit, bounds = \langle 0, \ldots, 0 \rangle$):
    if $tree.left \neq NULL$ then
        $p_{best}, limit \leftarrow FindMinNormPEEarly(tree.left, o, limit, bounds)$;
    end
    if $limit > 0 \wedge \|tree.data + o\| < norm_{best}$ then
        $norm_{best} \leftarrow \|tree.data + o\|$;
        $p_{best} \leftarrow tree.data$;
        if $(tree.data + o)[i] \leq load_{max}[i] \ \forall i \in 1, \ldots, d$ then
            $limit \leftarrow limit - 1$;          /* Candidate found, so reduce limit */
        end
    end
    if $limit > 0 \wedge tree.right \neq NULL$ then
        $oldBound \leftarrow bounds[tree.dim]$;
        $bounds[tree.dim] \leftarrow tree.data[tree.dim]$;
        if $\|bounds + o\| < norm_{best}$ then
            $p_{best}, limit \leftarrow FindMinNormPEEarly(tree.right, o, limit, bounds)$;
        end
        $bounds[tree.dim] \leftarrow oldBound$;
    end
    return $p_{best}, limit$
```



(a) Two Dimensional with Early Exit

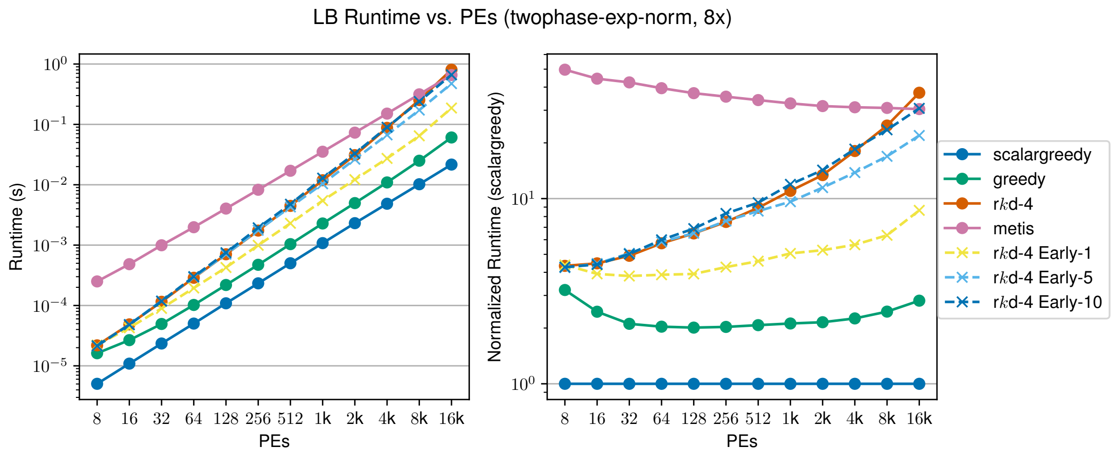


(b) Four Dimensional with Early Exit

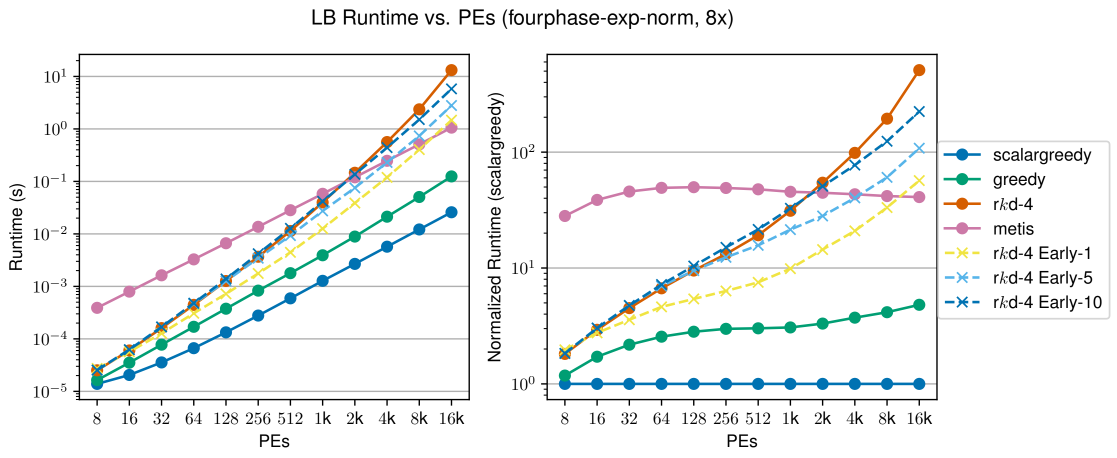


(c) Six Dimensional with Early Exit

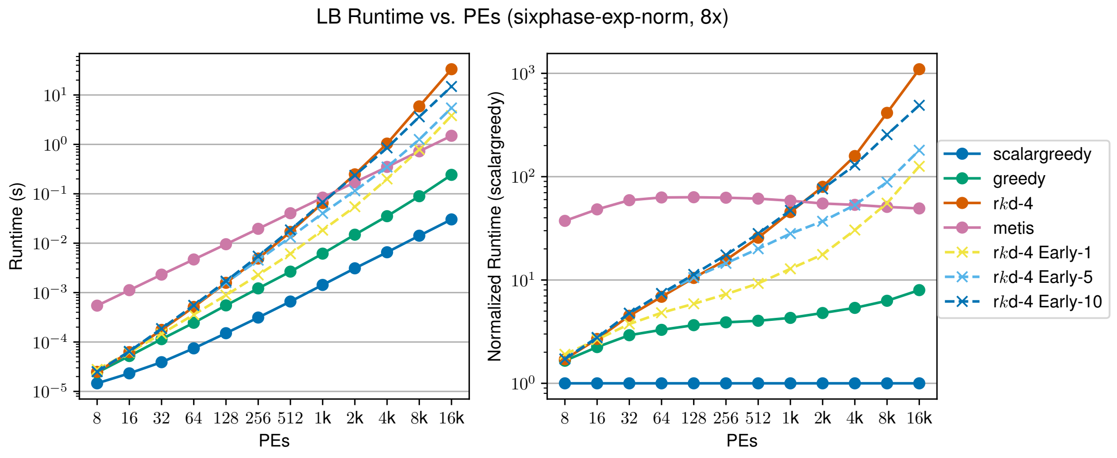
Figure 7.8: Early Exit Performance with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional with Early Exit



(b) Four Dimensional with Early Exit



(c) Six Dimensional with Early Exit

Figure 7.9: Early Exit LB Performance with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

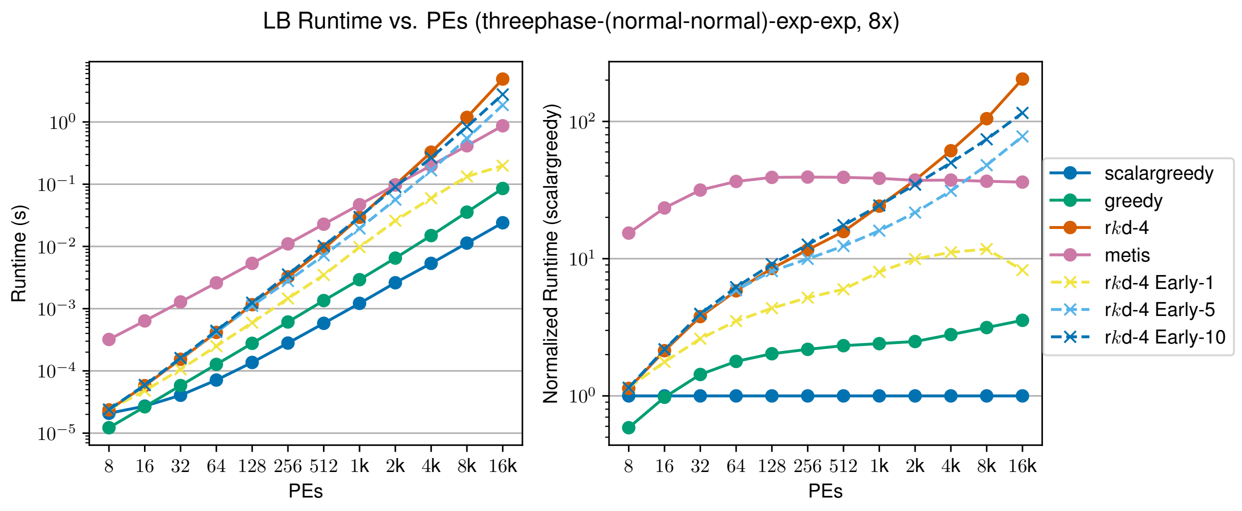


Figure 7.10: Early Exit LB Performance with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

would expect, since the fixed count early termination condition means that the proportion of total PEs that the early exit variant *avoids searching* grows with the number of PEs. In the experimental data, this culminates with the early exit variant with $limit = 1$ being approximately an order of magnitude faster than the regular rk-d strategy at 16,384 PEs for each of the tested load distributions. Thus, as opposed to the Pareto variant, the early exit variant does indeed provide an effective method to improve load balancing performance at scale.

With regards to our earlier questions, based on these results, it is clear that the strictness of the suitability criterion is not excessively restrictive, the early exit variant is able to improve upon the performance of rk-d. In fact, Figure 7.11, described in more detail below, shows that the early exit pathway is used by almost every placement for the two tested six dimensional cases.

Varying the value of *limit* affects the observed performance of the early exit variant. A smaller value causes performance to decouple from that of the base version at a smaller PE count and results in a lower runtime across all tested scenarios. The best performance is achieved when $limit = 1$, but all tested values result in performance benefits at large scale, reducing the slope of the performance curve at the high end of scaling.

Interestingly, unlike the results of the Pareto variant, here the load distribution has an effect on the ranking of strategies by performance at the high end of scale. Namely, in the six dimensional tests at 16,384 PEs, early exit with $limit = 1$ is faster than METIS for the normally distributed case, taking 1.12s and 1.46s, respectively, while early exit is slower than METIS for the case with alternating exponentially and normally distributed loads, taking 3.82s and 1.49s, respectively. A similar reversal occurs for the same strategies and distributions in the four dimensional case.

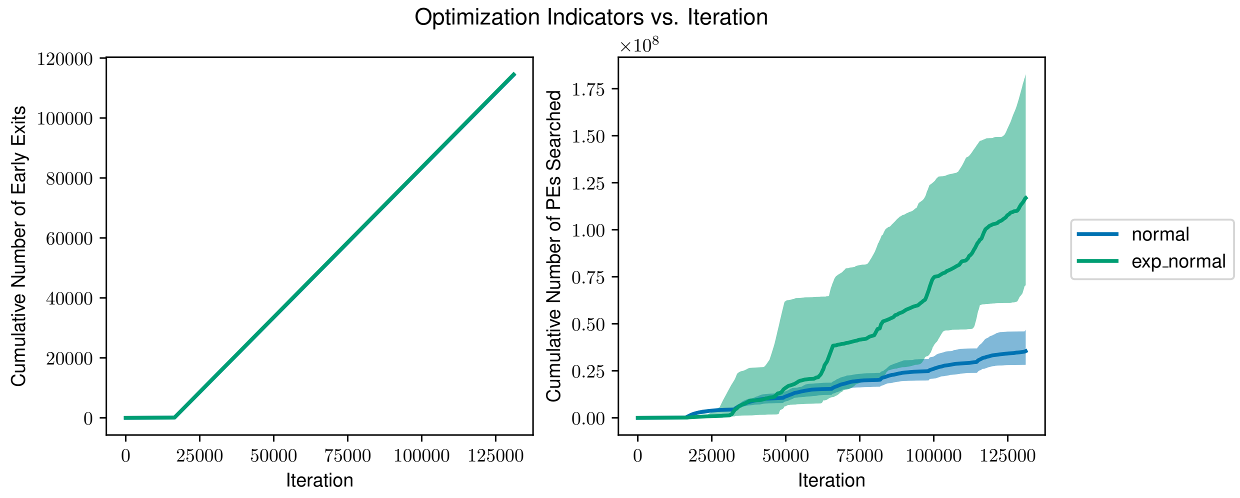


Figure 7.11: Comparison of Cumulative Early Exit Performance Indicators for 6-Normally and 6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE, 16,384 PEs

Figure 7.11 compares the median, minimum, and maximum of the cumulative number of early exits and cumulative number of PEs searched across one hundred runs of the early exit variant with $limit = 1$ for the two different six dimensional distributions at 16,384 PEs. As the figure shows, the progressions of the number of early exits for both load distributions are almost identical, with a median of 114,535 for the (exponentially, normally) distributed case and 114,387 for the normally distributed case (note that in our actual implementation, the first $|P|$ objects are each placed on separate PEs without performing a search, clearly visible in the figure as the flat section during the first $|P|$ iterations, so only $8 * 16,384 - 16,384 = 114,688$ iterations are eligible for the early exit optimization in this configuration). However, there is a sizable gulf in the number of PEs searched, with a final cumulative median of 1.17×10^8 for the (exponentially, normally) distributed load vectors and 3.55×10^7 for the normally distributed load vectors.

The difference in the number of PEs considered is likely caused by disparities between the distributions of each dimension. When the dimensions have different distributions or use different parameters for the same distribution, the search algorithm tends to more thoroughly search PE along the "smaller" or less variable dimensions than in uniform cases. This occurs because changes in such dimensions have a smaller effect on the norm than changes in the “larger” or more variable dimensions, due to the non-linearity of the norm calculation (excepting the 1-norm, which we do not consider since it is equivalent to scalar load balancing and leads to poor quality mappings). This phenomenon is exacerbated by the fact that the early exit variant only allows early termination when it also finds a new candidate that improves upon the PE with the current best post-placement norm, meaning that if a candidate with a relatively small norm that violates the early exit suitability criterion is found, often many criterion satisfying PEs will be searched but rejected because they do not improve upon the current best post-placement norm. As the number of dimensions increases, this becomes increasingly likely and may exponentially increase the number of rejected PEs as it searches through each such dimension.

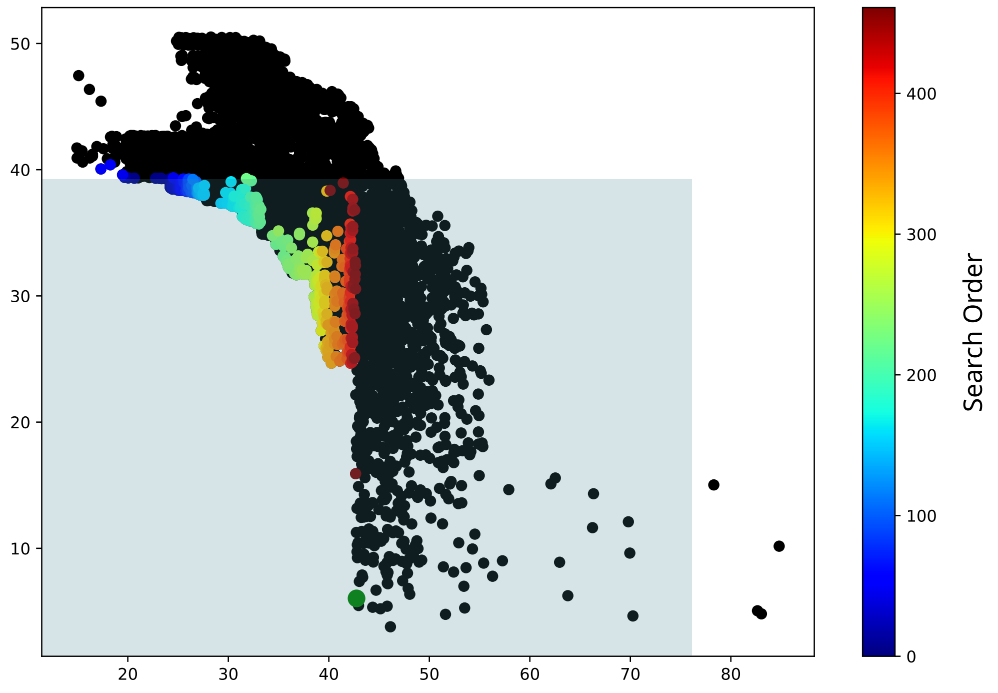


Figure 7.12: Early Exit Long Search Example

This effect is illustrated in Figure 7.12. The color gradient indicates searched PEs and their search order, dark blue PEs were searched first, and dark red PEs were searched last. Black PEs were never searched. The dark green PE with a larger marker at the bottom of the plot is the PE chosen for placement. The figure shows a single iteration of the algorithm processing a two dimensional (exponentially, normally) distributed problem, placing an object of load $\langle 8.51, 11.36\rangle$ and the maximum load vector is $\langle 84.81, 50.51\rangle$. Thus, any PE with load less than or equal to $\langle 76.30, 39.15\rangle$ satisfies the suitability criterion for this object, shown as the shaded gray region in the figure. However, because of the extant structure of the tree, the search algorithm begins searching in the top left of the plot, where it finds a candidate PE that exhibits the problematic combination of a small norm and violation of the early exit criterion. The algorithm then searches through many PEs in the satisfactory region before finally finding a PE with a smaller post-placement norm and returning it.

7.5.4 Quality

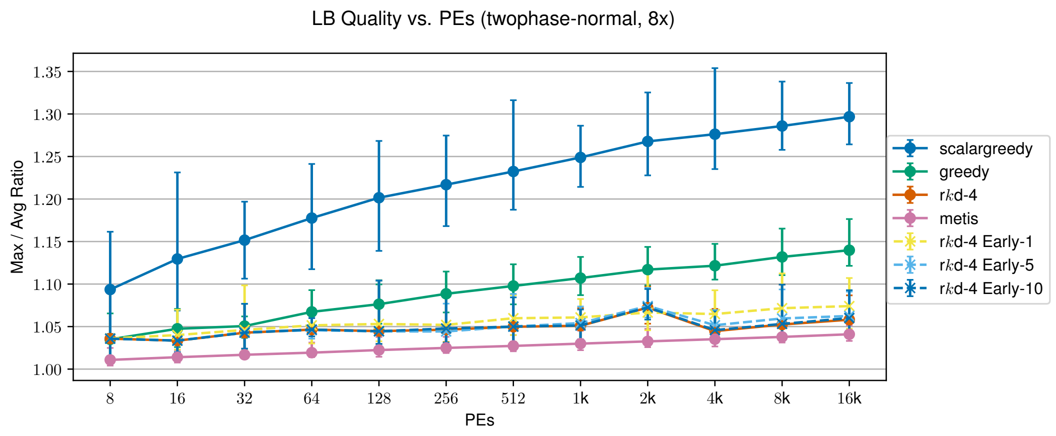
Because the early exit variant has different return conditions than the regular rk-d strategy, the resulting mappings differ from those of the regular version in most cases.

Figures 7.13 to 7.15 compare the quality of the early exit variant against the baseline load balancers, scored via the sum-based metric defined in Equation (3.1). Broadly, the quality of the mappings produced by early exit are very similar to those produced by regular rk-d, with only minor degradations.

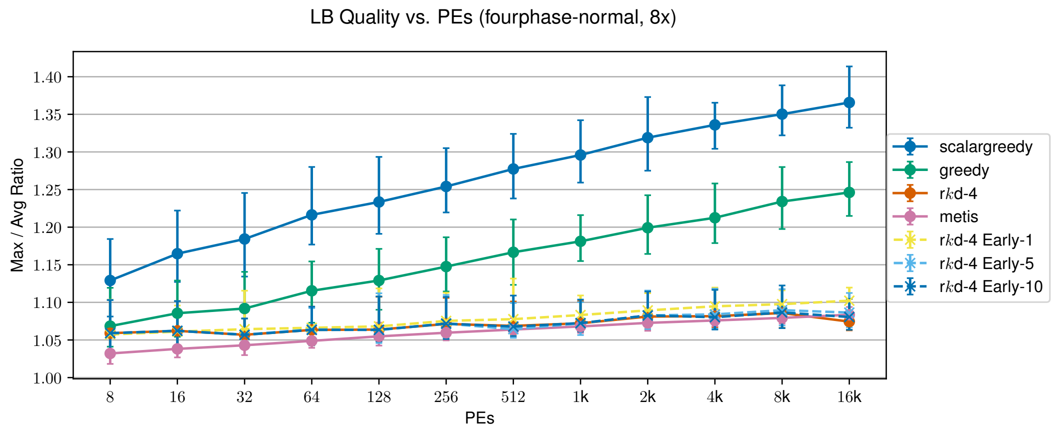
Table 7.2 shows quality results for the early exit variant normalized to the median of regular rk-d. The results in the table are aggregated across all experimental configurations, meaning the reported minimum value is the minimum across all of the different tested load distributions, likewise for the maximum, and the reported median is the median of each load configuration's median. Lower values are better, indicating a more balanced mapping.

At worst, the maximum $Max : Avg$ ratio observed for early exit is approximately 15% worse than the corresponding median result given by the rk-d baseline, and the median ratio is at most 2% worse than the baseline.

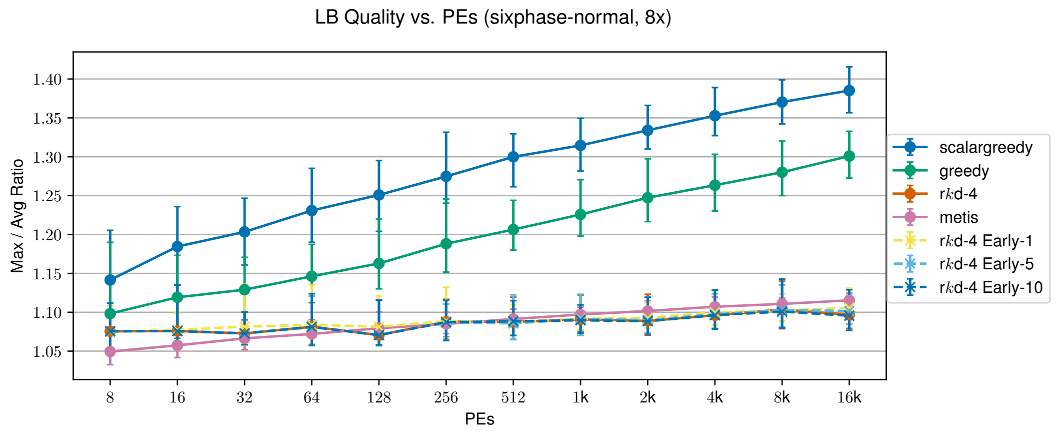
Table 7.2 indicates that the value of *limit* affects the quality of the results, with quality generally improving as *limit* increases and the search space is more thoroughly searched, meaning PEs with smaller post-placement norms are found before performing the early exit. Ignoring the small scale regime < 128 PEs where the early exit performance optimization is much less impactful, moving from $limit = 1$ to 5 to 10 improves the maximum observed normalized $Max : Avg$ ratio from 15% worse than the median of rk-d to 7% and then to 6%, respectively. However, the median solutions for each tested value of *limit* have a much tighter spread, with only the $limit = 1$ case differing from the baseline, and even then only by 2% at worst. The same applies for the minimum values, with all of the early exit strategies are within 1% of each other.



(a) Two Dimensional with Early Exit

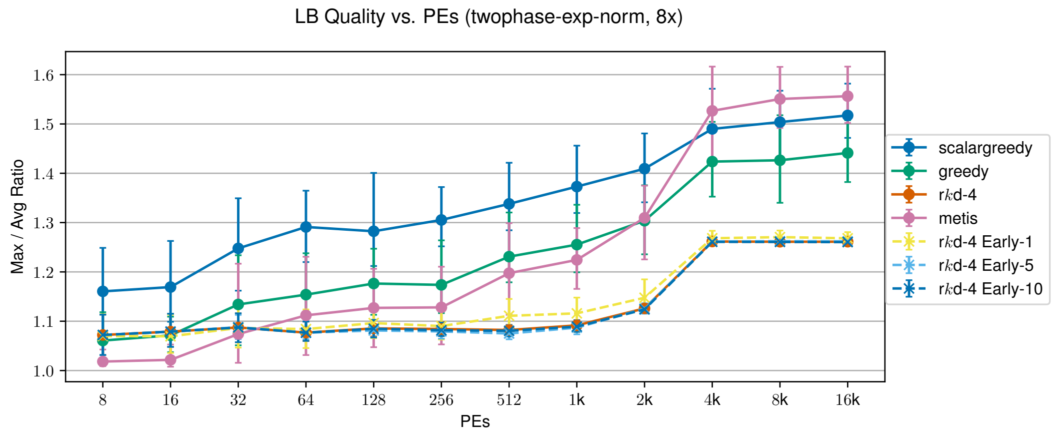


(b) Four Dimensional with Early Exit

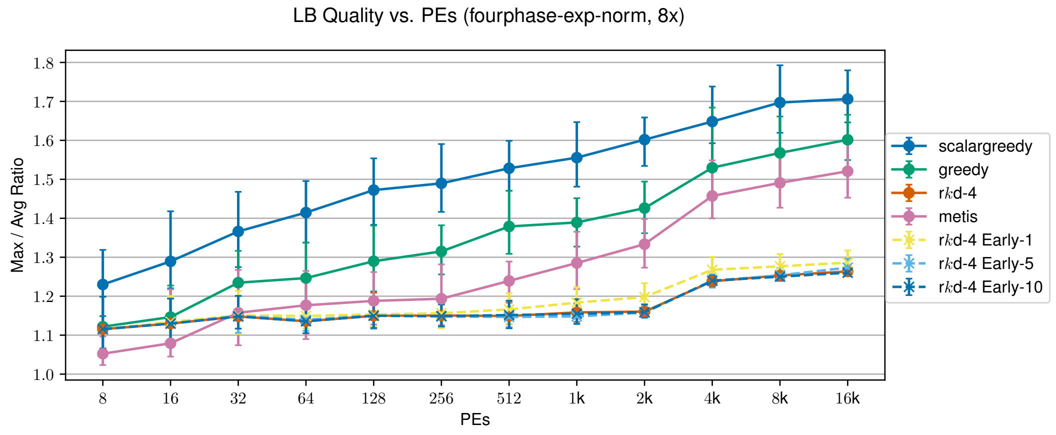


(c) Six Dimensional with Early Exit

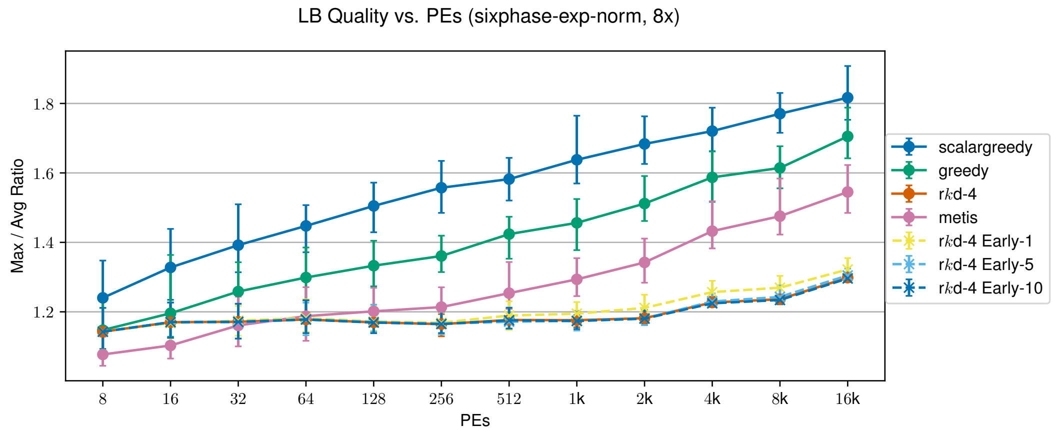
Figure 7.13: Early Exit Quality Comparison with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional with Early Exit



(b) Four Dimensional with Early Exit



(c) Six Dimensional with Early Exit

Figure 7.14: Early Exit LB Quality Comparison with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

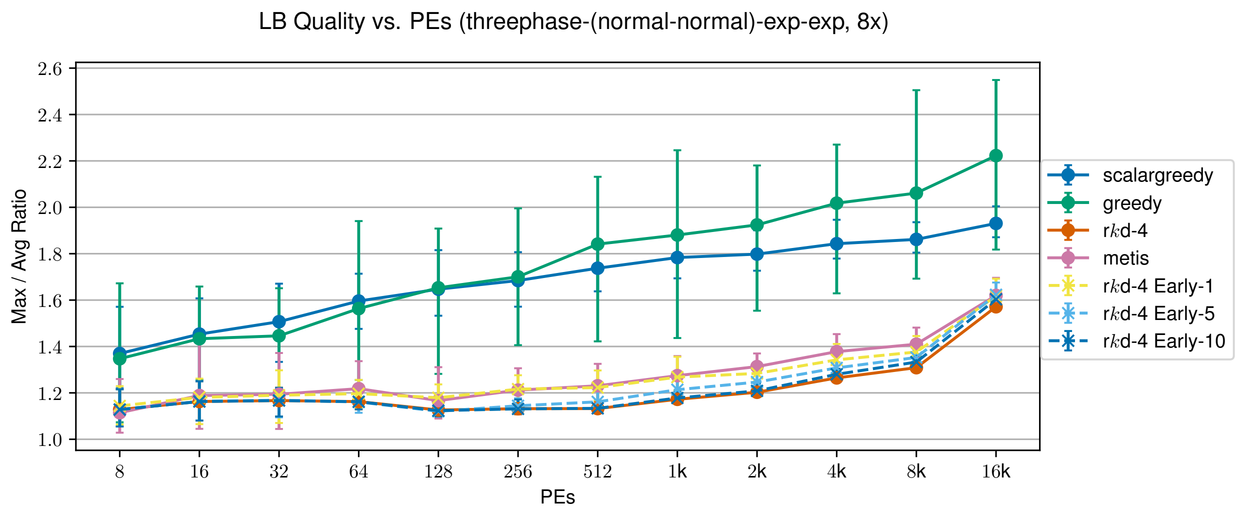


Figure 7.15: Early Exit LB Quality Comparison with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

These results indicate that the early exit variant generally provides high quality results while also providing a performance benefit over normal rk-d, as discussed in Section 7.5.3. While there are some cases where our testing reveals that quality may suffer relative to the baseline, this degradation is generally small and can be tuned by the user via altering the *limit* parameter.

7.6 HIERARCHICAL LOAD BALANCING

7.6.1 Motivation

The previous two optimizations, Pareto search and early exit, are based on modifying the behavior of the load balancing strategy itself, attempting to improve performance by reducing the search space via algorithmic adjustments. However, altering the algorithm is not the only way to reduce the search space, one can also change the inputs to the strategy, providing a smaller set of PEs, for example.

Figure 7.16 shows the performance impact of using a smaller set of PEs with the baseline LB strategies. As opposed to previous scaling plots in this chapter where the number of objects is a fixed multiple of the number of PEs, here the number of objects remains fixed at 128k for all PE counts (128k objects corresponds to the value used at 16k PEs in the other plots, which use eight objects per PE). Analyzing the figure, we see that all of the strategies decrease in runtime as the number of PEs decreases and that the rk-d strategy has

Strategy		**PEs**											
		8	*16*	*32*	*64*	*128*	*256*	*512*	*1K*	*2K*	*4K*	*8K*	*16K*
rk-d	*Min*	0.94	0.93	0.94	0.96	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	Median	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Max	1.10	1.08	1.06	1.07	1.06	1.04	1.04	1.03	1.05	1.05	1.05	1.03
Early-1	*Min*	0.94	0.92	0.91	0.97	0.98	0.96	0.98	0.98	0.97	0.99	0.99	0.99
	Median	1.00	1.00	1.01	1.01	1.01	1.01	1.02	1.02	1.02	1.02	1.01	1.02
	Max	1.09	1.09	1.11	1.13	1.13	1.13	1.14	1.15	1.10	1.12	1.10	1.08
Early-5	*Min*	0.94	0.93	0.93	0.96	0.97	0.96	0.98	0.98	0.98	0.98	0.98	0.98
	Median	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Max	1.10	1.08	1.05	1.06	1.06	1.06	1.07	1.07	1.07	1.06	1.06	1.07
Early-10	*Min*	0.94	0.93	0.94	0.95	0.97	0.97	0.98	0.97	0.98	0.98	0.98	0.98
	Median	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Max	1.10	1.08	1.06	1.07	1.06	1.04	1.04	1.04	1.05	1.04	1.05	1.04

Table 7.2: Quality of Early Exit with Varying *limit* Normalized to Median of rk-d

the highest rate of change as the number of PEs changes.

While reducing the number of PEs improves performance, in doing so, we solve a smaller problem than the one we need to solve, so how can we apply this approach to the LB problem at hand? Such smaller problems can be *hierarchically* composed to provide a solution to the original problem. To do so, we *coarsen* the original input data, combining multiple PEs from the original set of PEs to a single virtual "PE". The number of PEs to combine is configurable, specified by setting the value of a *coarsening factor* variable f_{coarse}. The coarsening process thus creates a new set of virtual "PE"s of size $|P|/f_{coarse}$. Objects placed on these virtual "PE"s will eventually be mapped to one of the corresponding original PEs.

After coarsening, we use this new smaller virtual "PE" dataset as an input argument along with the entire set of objects for an invocation of the LB strategy. Then, we use the resulting object mapping and the corresponding *uncoarsened* PEs as input for multiple invocations of the strategy, one for each of the $|P|/f_{coarse}$ coarsened "PE"s. Each of these invocations takes as input the set of objects assigned to the coarsened "PE", a set of average size $|O|/(|P|/f_{coarse})$, and the subset of the original PEs that were combined to form the corresponding coarsened version, of size f_{coarse}; each of these input sets is orthogonal to those of all other such invocations. Then, the union of the resulting mappings from these invocations is a solution to original problem, mapping all of the objects to non-virtual PEs.

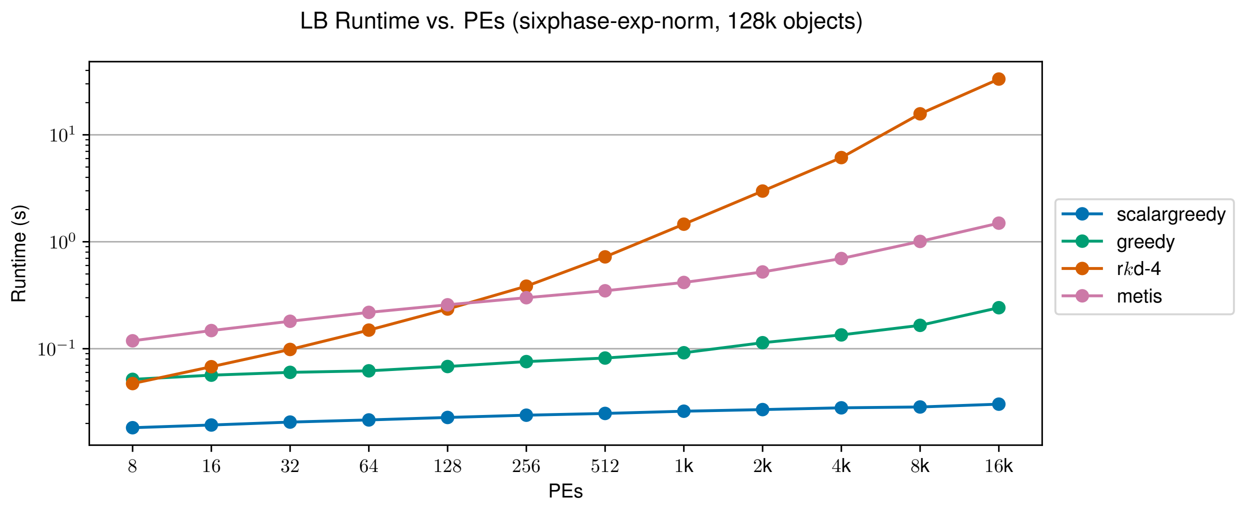


Figure 7.16: Performance Comparison with 6-(Exponentially, Normally) Distributed Vectors, 128k Objects

Parallelism

Because the invocations which use the results of the invocation with coarsened data all operate on orthogonal sets of PEs and objects, another benefit of this hierarchical execution scheme is that they can run in *parallel*, and, further, can be run in a *distributed* manner. A centralized scheme where the strategy is only invoked once with all of the relevant input data can only utilize multiple cores if the algorithm itself is parallelized. Doing so can be tricky from both the correctness and performance perspectives, especially for algorithms in which each iteration operates on the same shared data structure, such as via the modification of a tree or a heap, as LB strategies are wont to do. Also, it requires invasive modification of the logic of the LB strategy, while the hierarchical execution scheme requires no code changes. Finally, even if the algorithm is parallelized, it can likely only take advantage of shared memory parallelism, limiting the scope of the potential performance benefit. Strategies can be restructured to run in a distributed way, but this is a decidedly non-trivial paradigm shift, necessitating the addition of potentially costly network communication and additional logic to coordinate the decision making process among all of the distributed participants.

To explore the viability of parallelism for non-hierarchical vector load balancing, we implemented a shared memory parallel version of the regular rk-d strategy which operates in a batched manner, first searching for norm-minimizing PEs mappings for a batch of objects in parallel without actually placing any of them, then placing each one serially, performing another search on the current updated state of the tree if any previous object in the batch was mapped to the same PE. The quality results were in line with the serial version (albeit slightly different due to a different exploration order), but we were never able to observe a significant performance benefit.

7.6.2 Related Work

Ahmad *et al.* introduce the idea of hierarchical task placement in an attempt to create a hybrid of the centralized and fully distributed approaches. In [84], they describe a hierarchical (which they call "semi-distributed") task placement scheme wherein the system is partitioned into orthogonal regions. The scheduler first attempts to place tasks within their originating region, but will place tasks into other regions if the load of the lightest loaded local node exceeds some threshold.

Zheng *et al.* develop tree-based hierarchical load balancing methods with Charm++ in [85]. These hierarchical load balancing methods are used to improve the load balancing performance at large scale for NAMD in [86]. Our implementation builds upon this work, modernizing and reimplementing it to increase its flexibility and add support for vector load balancing.

Bak *et al.* create a hierarchical load balancing approach combining the periodic load balancing of Charm++ with intra-node work stealing via OpenMP tasks in [87].

Teresco *et al.* use a hierarchical balancing scheme to apply different load balancers at different levels of the system, using a graph partitioner which minimizes communication between nodes, but potentially leaves some compute imbalance at the inter-node level, and then applying a faster geometric partitioner for intra-node balancing in [88].

Lifflander *et al.* compare the performance and quality of a hierarchical persistance-based balancing scheme to a centralized scheme and balancing via retentive work stealing [89], finding that the hierarchical scheme delivers good performance, especially at scale, and deficiencies in quality can be mitigated via coupling it with a performance work stealing scheme.

7.6.3 Implementation

To support hierarchical execution, we implemented a new load balancing framework called*TreeLB*. TreeLB fully decouples the algorithmic part of the strategy from execution structure, statistics collection, and object migration. This separation simplifies the implementation of load balancing algorithms and provides tremendous flexibility in terms of how strategies are utilized.

TreeLB, as the name suggests, consists of a tree-based execution structure. Leaf nodes correspond to PEs, and each non-leaf node in the tree is responsible for the set of PEs formed by the union of the domains of its children. The top level always consists of a solitary root node and the bottom level always consists of $|P|$ nodes, one for each PE, but the number of and branching factor for the intermediate levels in the tree are configurable by the user.

In the parlance of programming languages, TreeLB is essentially an abstract class. It is not in itself a load balancing strategy, rather, it offers a way to create a customized LB strategy by composing different load balancing algorithms into a tree of the specified structure. Each non-leaf level of the tree requires the user to select a load balancing algorithm for use at that level.

At a high level, a TreeLB-based strategy proceeds as follows:

1. PE level load information is gathered at the leaves.
2. Load data are passed up to the parent node. The data are potentially coarsened at each level, and data continue to be passed up until reaching the root or a disabled level (coarsening and disabled levels are explained in the description of per-level LB options given below).
3. The highest enabled level runs its specified load balancing algorithm with the data passed in from its children.
4. The resulting mapping is passed down the tree until reaching the leaves. Uncoarsening and further invocations of load balancing occur at each level if coarsening is enabled.
5. Leaf nodes process the new mapping, issuing migration requests for resident objects that have a new PE assignment.

There are two ways in which these per-level load balancers may be used:

Step Interval The user may specify a interval for each level of the tree more than one level removed from the leaves. By default, the load data from the PEs are passed all the way up to the top of the tree whenever load balancing is called, running the algorithm specified at the root and proceeding back down the tree as configured. However, when a step interval is provided for a level, data and control flow only pass to that level and beyond if the number of the load balancing step is divisible by the given interval. When the step number is not divisible by the specified interval of the parent level, each node at the current level uses the load balancer of the current level to independently and concurrently balance load among the PEs in its subtree. This enables use cases such as performing LB across the PEs of a physical node at high frequency (migrations are local and thus cheap, but only intra-physical node imbalance can be corrected), and performing global LB at low frequency (migrations are more expensive, but system wide imbalance can be fixed).

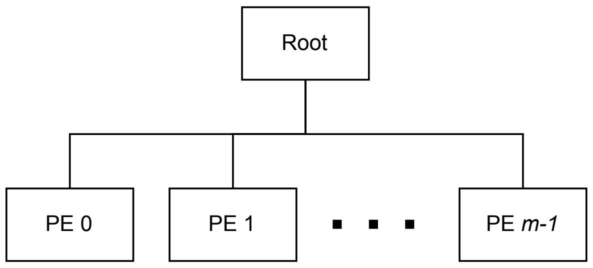
Coarsening When the user enables coarsening at a given level, load data are coarsened when passed up the tree, and the resulting mapping received from its parent is uncoarsened when being passed down the tree. When a level performs uncoarsening, it has to execute its given LB algorithm to map the objects it receives to the uncoarsened PEs (we assume here that we are only coarsening PEs; it is also possible to coarsen objects). When coarsening is used, load balancing consists of a series of sequential phases starting at the top of the coarsening hierarchy (usually but not necessarily the root, depending on the configuration) and progressing down the tree, since uncoarsening cannot run until the parent provides a mapping.

These two options are not exclusive, users may both enable coarsening and specify intervals, in which case coarsening is performed and data are passed up when the parent level is enabled, otherwise coarsening does not occur and load balancing proceeds for the subtree as described above.

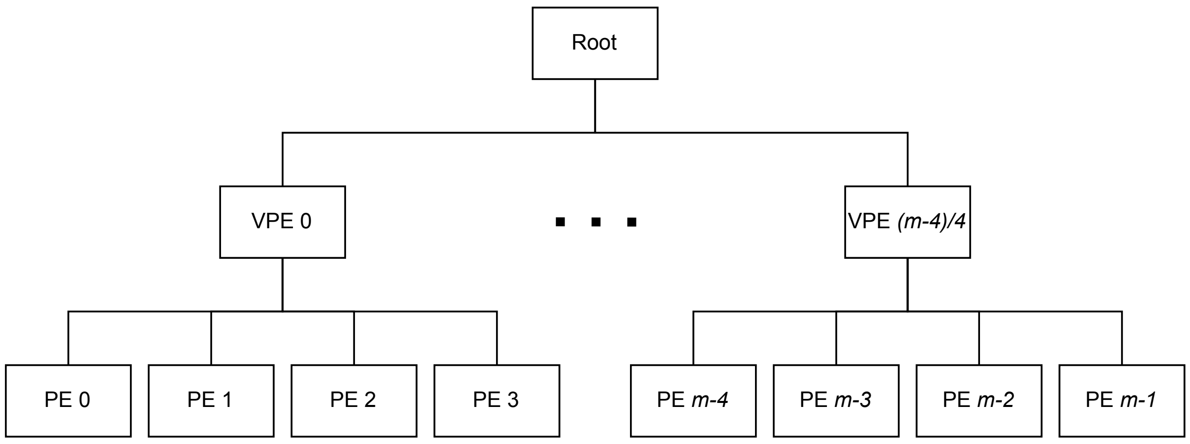
Figure 7.17 shows example two level and three level trees. The two level tree in Figure 7.17a runs in a centralized manner, collecting all load data onto a single root PE, executing the selected load balancing algorithm on that PE with a global view of the load data, and finally broadcasting the new mapping back to each of the PEs. The three level tree in Figure 7.17b runs in a hierarchical manner, the intermediate level performing coarsening and/or subtree-only balancing depending on the configuration. We have assigned labels to the intermediate level of the tree assuming that it is being used for coarsening, denoting each node with its virtual "PE" (VPE) index, each corresponding to four actual PEs.

Trees consisting of more than three levels are also supported, although the logarithmic nature of tree decomposition and the allowance of arbitrarily large branching factors means that it is unlikely that trees with many levels will be useful. Additionally, using a tree with fewer levels and/or a larger branching factor tends to improve load balancing quality, as increasing subtree sizes at low levels provides the load balancer a broader view of the state of imbalance and increased scope to mitigate it. We corroborate this claim via the results in Section 7.6.5.

TreeLB is configured by passing in a configuration file at runtime, such as the example shown



(a) Two Level Tree



(b) Three Level Tree

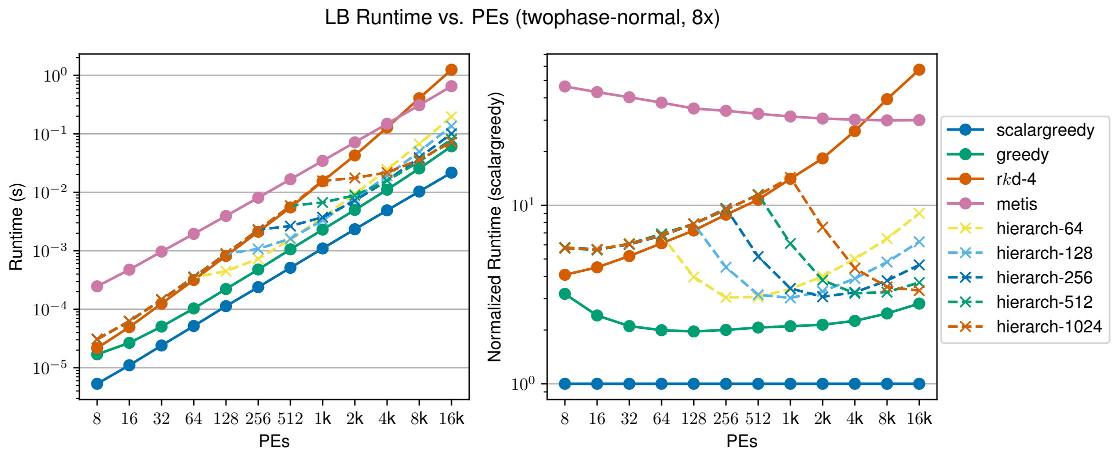
Figure 7.17: TreeLB Structure Diagrams

in Listing 7.1. This configuration creates a three level tree, the size of the intermediate level automatically determined by the number of processes in the job. The top and intermediate levels use the rk-d strategy with the 4-norm and coarsened hierarchical execution is enabled.

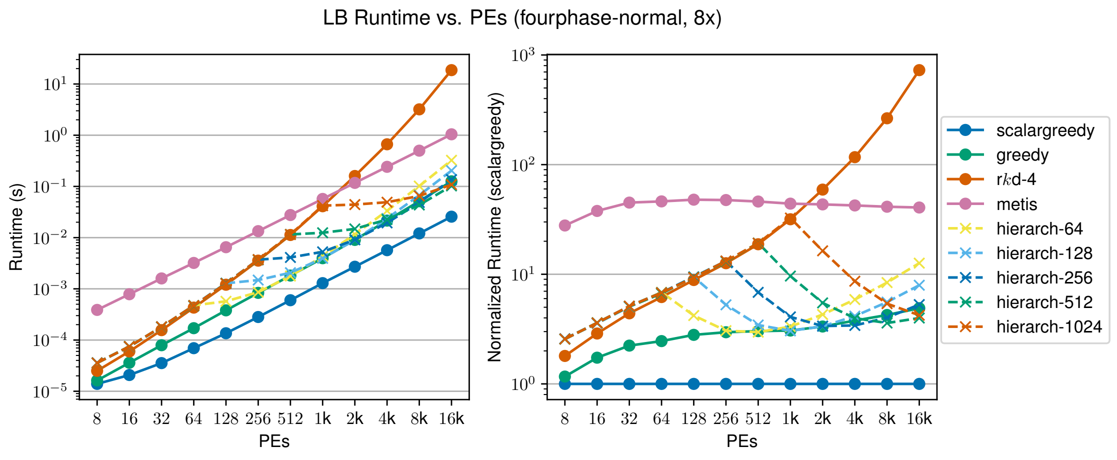
7.6.4 Performance

Figures 7.18 to 7.20 compare the performance of hierarchical execution against the baseline strategies. The hierarchical executions all use three level PE-intermediate-root trees configured to use the normal rk-d balancing algorithm with the 4-norm at the intermediate and root levels. Coarsening is enabled at the intermediate level, and the coarsening factor f_{coarse} varies from 64 to 1024, as labeled in the legend.

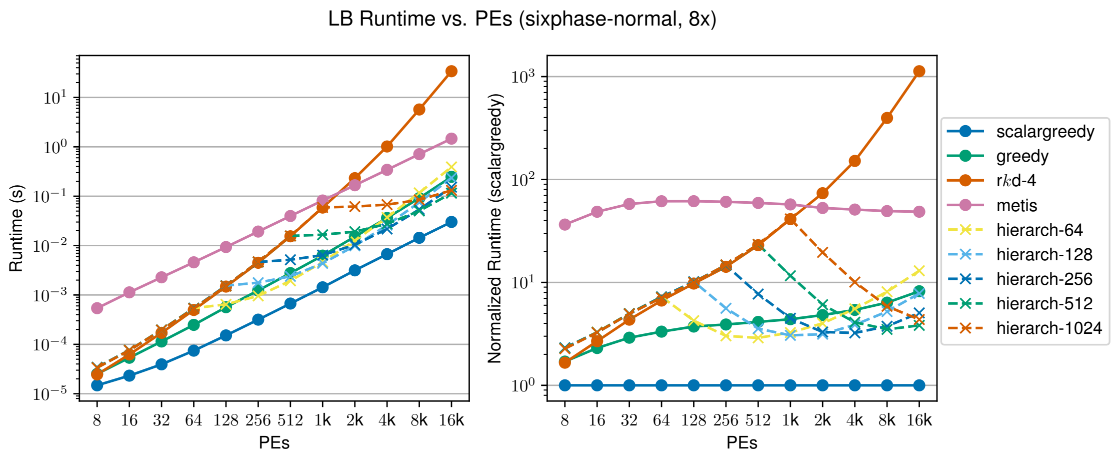
To reiterate from Section 7.6.3, because these experiments are using three level trees with hierarchical coarsening, load balancing consists of two sequential phases. In the first phase, the root level balancer runs using coarsened PE data (e.g. in the hierarch-256 case, the root level balancer assigns every object to one of $|P|/256$ virtual PEs; each virtual PE corresponds



(a) Two Dimensional with Hierarchical Execution

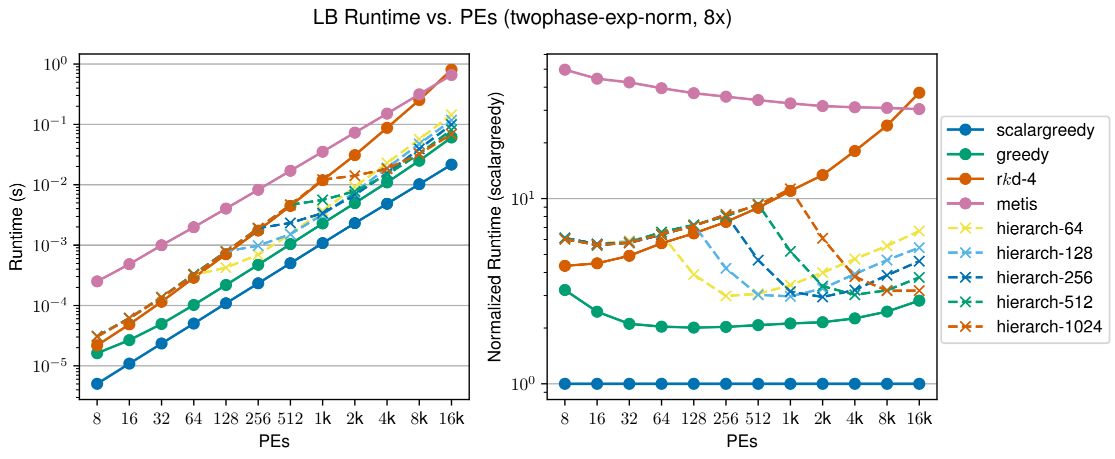


(b) Four Dimensional with Hierarchical Execution

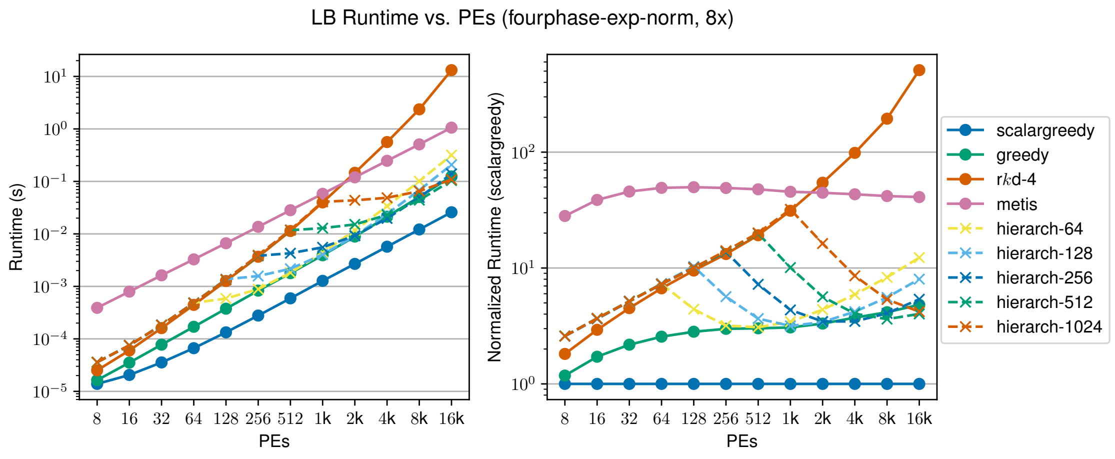


(c) Six Dimensional with Hierarchical Execution

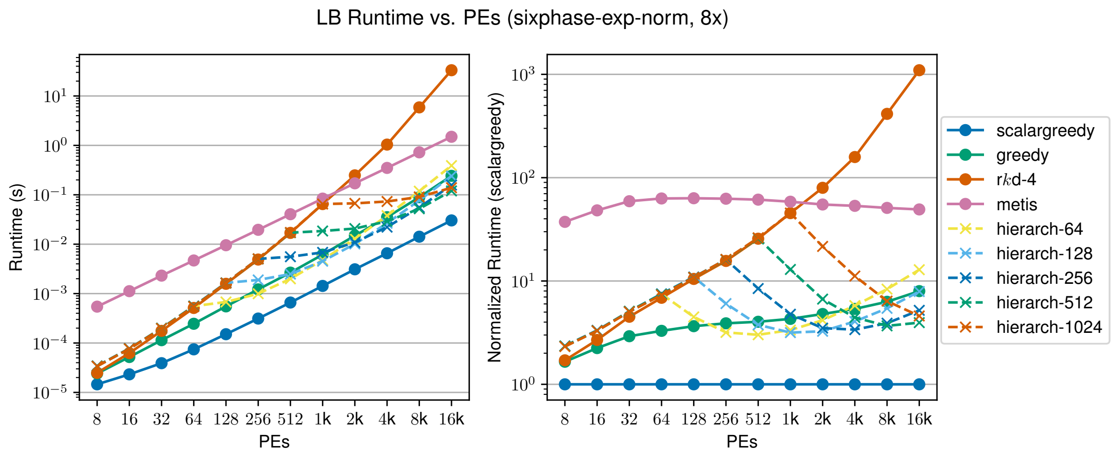
Figure 7.18: Hierarchical Performance with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional with Hierarchical Execution



(b) Four Dimensional with Hierarchical Execution



(c) Six Dimensional with Hierarchical Execution

Figure 7.19: Hierarchical LB Performance with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

```
{
  "tree": "PE_Process_Root",
  "root":
  {
    "pe": 0,
    "strategies": ["rkd4"]
  },
  "process":
  {
    "strategies": ["rkd4"],
    "coarsen": true
  }
}
```

Listing 7.1: Example TreeLB Configuration File

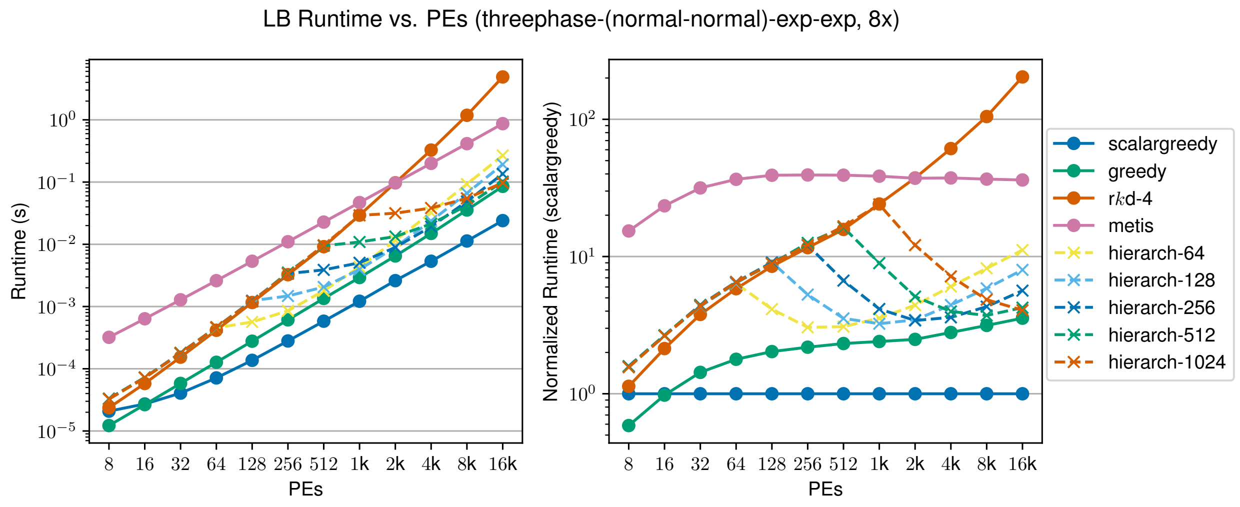


Figure 7.20: Hierarchical LB Performance with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

to 256 real PEs). After the first phase completes, the mapping from objects to virtual PEs is given to the intermediate level. Each node in the intermediate level then concurrently uncoarsens, running the intermediate level load balancer to remap the objects assigned to its virtual PE to real PEs (e.g. continuing the earlier case, each of the $|P|/256$ nodes of the intermediate level remaps its objects across the 256 PEs in its subtree).

Thus, the reported time for the hierarchical executions in Figures 7.18 to 7.20 is the sum of the time taken for the root level load balancer invocation and the maximum time taken by any of the invocations of intermediate level load balancers. Even though we are making more calls to load balancing algorithms when executing hierarchically, performance is markedly than the centralized rk-d strategy, as each invocation works on a smaller set of input data and concurrency at the intermediate level allows for parallel execution.

The performance of the hierarchical and centralized rk-d are effectively at par when $|P| \leq f_{coarse}$, modulo minor overhead for hierarchical execution in the small scale of 8-32 PEs. This is expected, as coarsening leads to only a single PE at the root level, resulting in a trivial mapping, and the intermediate level only consists of a single node, which performs load balancing with the same inputs as the centralized case receives. However, when $|P| > f_{coarse}$, the performance of the hierarchical and centralized versions decouple, with the runtime of the hierarchical version growing much more slowly than the centralized version. This same pattern occurs across all tested load configurations and values of f_{coarse}. The runtime growth rate of the hierarchical version slowly increases as we scale upward from the decoupling point, but both growth rate and absolute runtime remain well under the centralized version universally post-decoupling. The performance gulf between the centralized and hierarchical versions exceeds two orders of magnitude for both tested six dimensional load vector cases at 16,384 PEs.

Interestingly, for the four and six dimensional cases at large scale, hierarchical rk-d is faster than even the centralized greedy strategy (but it never outperforms the centralized scalar greedy strategy). The hierarchical strategies also outperform the METIS strategy in all tested cases. This would likely not hold at larger PE counts, as the runtime of hierarchical cases with small f_{coarse} is growing faster than METIS at large scale.

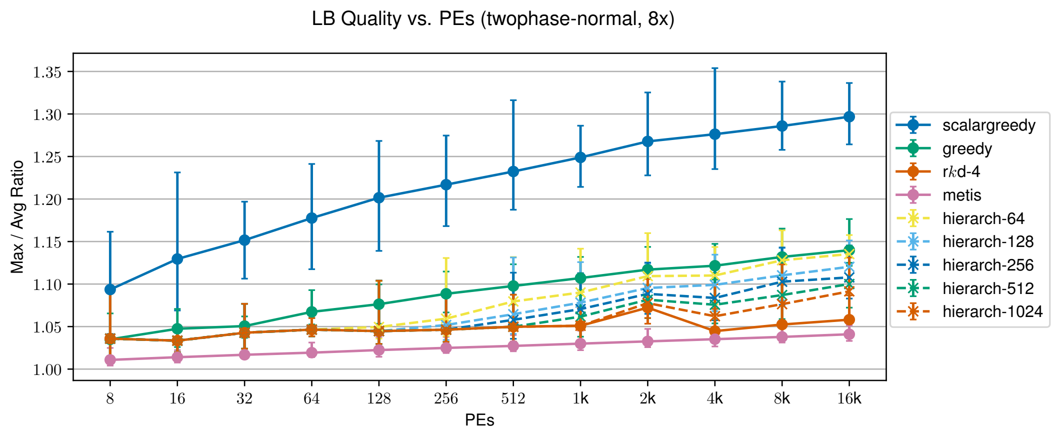
7.6.5 Quality

Figures 7.21 to 7.23 compare the quality of the results of hierarchical executions of rk-d to those of the baseline strategies.

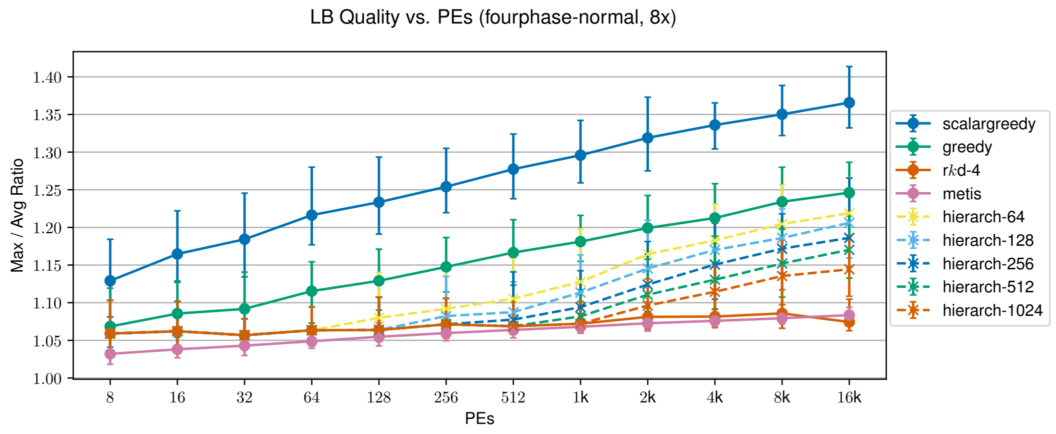
Using coarsened hierarchical load balancing trades off performance for quality. No single invocation of the load balancing strategy ever sees a detailed global view of the entire PE space, its scope either limited to only a subset of PEs or the coarsened representation of the whole PE space. The size of both of these views is dependent on the coarsening factor f_{coarse}. As f_{coarse} increases, the root sees fewer virtual PEs, but each leaf invocation of the load balancer has a larger subset of PEs to assign its allotment of objects to, increasing the quality of the resulting mapping.

The dimensionality of the problem also affects the decrease in quality. As our results show, as the number of dimensions increases, the quality of the hierarchical LB mappings decreases relative to those of the the centralized version.

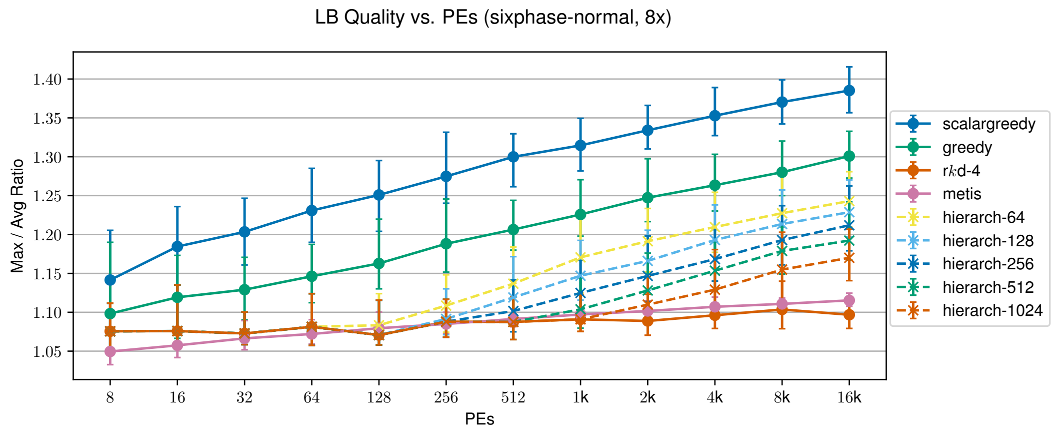
For the normally distributed load vectors, the quality of the mappings from the hierarchical



(a) Two Dimensional with Hierarchical Execution

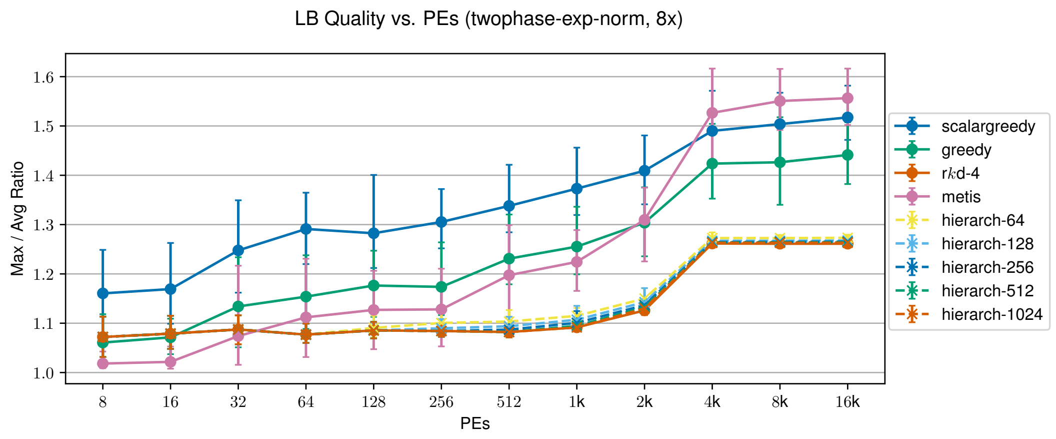


(b) Four Dimensional with Hierarchical Execution

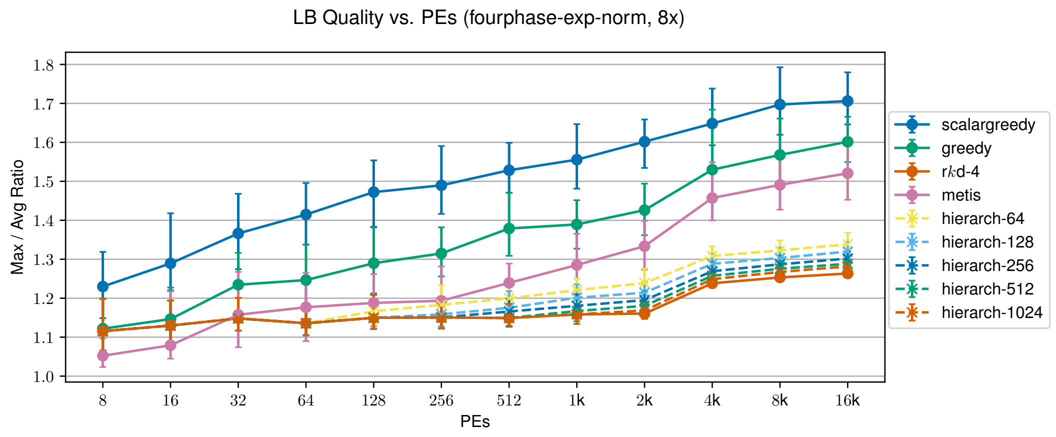


(c) Six Dimensional with Hierarchical Execution

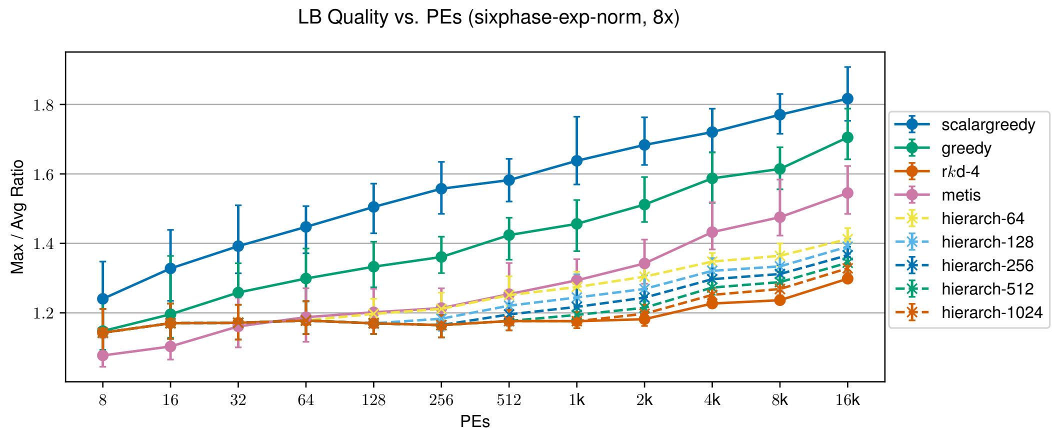
Figure 7.21: Hierarchical LB Quality Comparison with 2/4/6-Normally Distributed Vectors, 8 Objects/PE



(a) Two Dimensional with Hierarchical Execution



(b) Four Dimensional with Hierarchical Execution



(c) Six Dimensional with Hierarchical Execution

Figure 7.22: Hierarchical LB Quality Comparison with 2/4/6-(Exponentially, Normally) Distributed Vectors, 8 Objects/PE

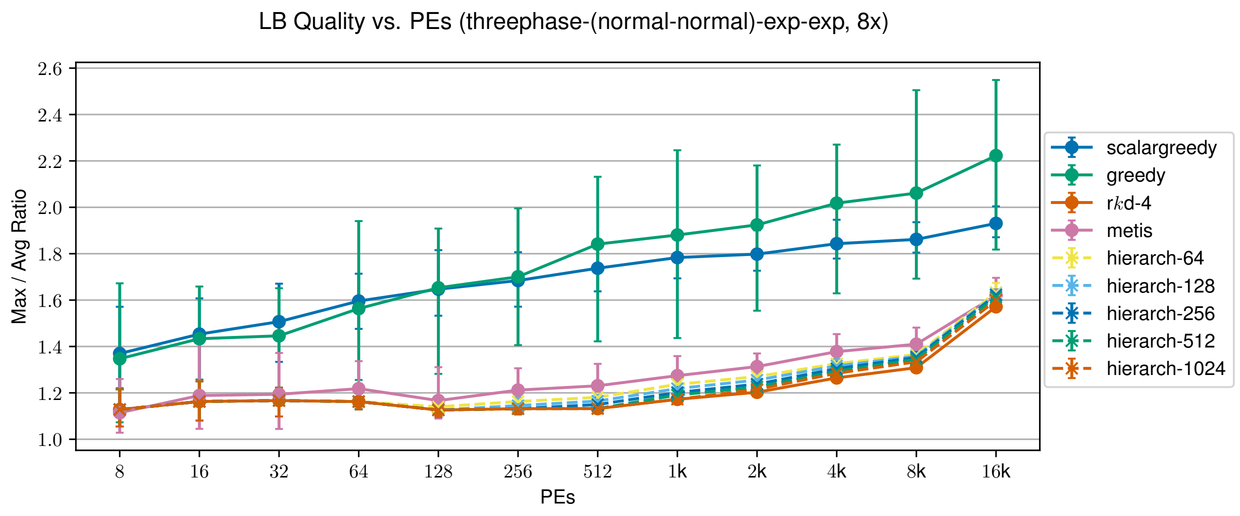


Figure 7.23: Hierarchical LB Quality Comparison with 3-(Normally/Normally, Exponentially, Exponentially) Distributed Vectors, 8 Objects/PE

executions becomes progressively worse beyond its decoupling point, with the $Max : Avg$ ratio increasing roughly logarithmically (note that these are semi-log plots). Even with this degradation, the quality is still better than that of the greedy strategy in all tested cases.

The quality discrepancies for the (exponentially, normally) and (normally/normally, exponentially, exponentially) cases are smaller, with all hierarchical executions delivering better quality mappings than every other tested strategy outside of centralized rk-d.

Overall, hierarchical LB provides a dramatic performance improvement at the cost of a low to moderate degradation in quality, depending on the distribution of the load vector. Both performance and quality depend on the selected value of f_{coarse}, providing users with a degree of tunability to adjust this tradeoff according to their preferences.

7.7 CONCLUSIONS

Performance is a critical aspect of load balancing for parallel applications using HPC. Such applications are often specifically crafted and tuned to eke out every bit of performance possible from the machine. Even if the use of load balancing improves the application mapping and resource utilization, it is usually untenable if that comes at the cost of lengthy strategy runtime; the benefit provided by balancing should be higher than its cost. Such considerations are particularly acute for vector load balancing, given the increased complexity of the problem. Luckily, the optimizations explored in this chapter show that the performance of load balancing can be improved by a variety of techniques.

Interestingly, the results shown in this chapter indicate that the performance characteristics of the tested load balancing strategies largely do not vary across different distributions of load vectors (with the notable exception of the early exit variant as dimensions increasingly vary), whereas differences in the dimensionality of load vectors have a significant impact on performance.

One potential future line of work is to analyze the impact of combining the different approaches from this chapter. Searching through only the Pareto frontier, terminating early when promising results are found, and executing with hierarchical coarsening are all mutually compatible optimizations and thus can be applied simultaneously.

CHAPTER 8: CONCLUSIONS AND FUTURE DIRECTIONS

8.1 CONCLUDING REMARKS

In this thesis, we analyze the causes, structure, and mitigation of complex load imbalance in parallel applications. We propose using *vector load balancing* as a way to improve application performance in a variety of scenarios. Specifically, we provide an overview of the concepts and history of scalar load balancing and its application in the parallel programming framework Charm++ in Chapter 2, formulate the mathematical background and design new load balancing strategies that support vectors of load in Chapter 3, discuss the factors giving rise to phase-based applications and their need for vector load balancing in Chapter 4, consider the benefits of runtime support for malleable vector scheduling and vector load balancing for programs that use accelerators in Chapter 5, examine scenarios with disparate load measurements and the addition of constraint support to vector load balancing schemes in Chapter 6, and finally analyze the scalability of vector load balancing and introduce performance optimizations in Chapter 7. We show that vector load balancing offers advantages that scalar load balancing fundamentally cannot provide and validate our techniques with simulated and practical runs of benchmarks, miniapps, and production applications. Vector load balancing improves upon the applicability, utility, and resilience of scalar load balancing, enriching the landscape of parallel computing and enabling ever greater performance as software and machines grow in scale and complexity.

8.2 FUTURE DIRECTIONS

8.2.1 Alternative Data Structures

As discussed in Section 3.2.2, we tested several different implementations for a norm-based load balancer before ultimately settling on using the relaxed variant of the k-d tree as the core data structure. However, there are other potential data structure candidates that we did not fully evaluate in our prototyping process. The family of data structures useful for this problem are called *metric trees*, introduced by Uhlmann in [90], of which the k-d tree and its variants are members. This family of trees should be more thoroughly explored in the context of vector load balancing; such experimentation may be able to improve upon the performance of the rk-d tree-based version.

One particular data structure of interest is the *range tree*, developed by Bentley in [91]. Range trees allow for multidimensional range-based queries, returning all entries which fall within the specified range. This data structure, coupled with a search optimization called *fractional cascading* [92], may be usable for implementing a high-performance alternative version of the norm-based load balancing algorithm.

8.2.2 Communication-Aware and Migration-Aware Load Balancing

In this thesis, we focus on load balancing in the "pure form", i.e. we consider only the data contained in the load vector, ignoring other factors that affect overall performance in practice, such as the cost of migrating objects when applying load balancing decisions and the effect of object mappings on communication cost. Empirically, for certain classes of applications such as those with heavy communication, these other factors often are more important to total execution time than improving the mapping from a computational load perspective.

Thus, new load balancers that take vector loads into account while also considering these factors of communication and migration cost should be constructed. Existing communication-aware load balancers using techniques such as layout based on space-filling curves (SFC), orthogonal recursive bisection (ORB), and graph partitioning (such as METIS) can be extended to use vector loads when making placement decisions, changing the resulting load distribution without sacrificing the contiguity of their partitions. A similar integration is possible with migration-aware load balancers, as well, avoiding migration when each dimension is within some factor of the average load in that dimension, for example.

However, the additional limitations imposed by this communication or migration awareness can make it difficult for a vector load balancer to realize benefits over a scalar load balancer, as certain mapping choices that improve computational or other vector balance may worsen the communication locality. We implemented a vector-aware version of an ORB strategy to use with ChaNGa [16], but testing showed no improvement over the extant scalar version. One potential solution to this lack of improvement is to relax the strictness of the partitioning requirements; rather than strictly bisecting along a hyperplane as in ORB, one may bisect more fuzzily, allowing elements within some threshold distance of the splitting hyperplane to be placed in either partition depending on their load vector, potentially worsening communication cost while improving computational load balance. This technique of trading off one factor for another is essentially the main idea behind vector load balancing itself, and it seems likely that it can provide benefits here as well.

8.2.3 Dimensionality Reduction

As discussed in Chapter 7, a significant difficulty with vector load balancing is managing the performance of load balancing strategies. The hierarchical schemes developed in that chapter reduce strategy runtime by running the LB strategy multiple times at different levels of a hierarchical tree and shrinking the set of PEs considered by each invocation via partitioning and coarsening, reducing the size of the PE search space. While this hierarchical approach is effective in reducing runtime, there are other complementary techniques that can also provide speedups by reducing search time, such as *dimensionality reduction*.

Rather than reducing the number of PEs to be searched, dimensionality reduction shrinks the search space and search complexity by reducing the number of dimensions of the problem itself. One of the core concepts in dimensionality reduction is the difference between the *representational dimension* and *intrinsic dimension* of a dataset [93]. The representational dimension is the actual number of dimensions a given dataset uses in its representation, whereas the intrinsic dimension is the number of dimensions that an embedding of the same dataset requires in order to maintain distances between points. For example, a dataset $D = \{p \in \mathbb{R}^3 | p_z = 1.0\}$ where the z coordinate of every point is the same has a representational dimension of three, but an intrinsic dimension of two (assuming the data are not arranged along a single line or at a single point; in these degenerate cases, the intrinsic dimension would be one or zero, respectively).

While embedding a dataset into its intrinsic dimension maintains the distance between points in the dataset, meaning it does not affect the results of computations such as clustering or nearest-neighbor search, it does not necessarily preserve the distance between points and the origin, and so is not a norm-preserving transformation in general.

For load balancing applications, since we are concerned about the sums and norms of load vectors, we prefer to utilize norm-preserving dimensionality reduction methods, such as principal component analysis (PCA) [94], which only uses norm-preserving rotational transformations. Since the resulting components of PCA are sorted by the amount of variance they represent in the underlying data, a dimensionality reduction scheme may keep only the first few components, potentially reducing the representational dimension of the new dataset below the intrinsic dimension of the original. While this can lead to a loss of information, the transformation ensures that this loss is small, and the degree of loss can be quantified by calculating how much variance is explained by the discarded dimensions.

Using such methods as a preprocessing step before load balancing should maintain reasonable fidelity between the reduced and original data while also ameliorating the difficulties of searching through a multi-dimensional space. Cases such as the fourteen dimensional load vector presented by EMPIRE-VT in Section 4.5.3 may need to use dimensionality reduction for load balancing at large scale.

8.2.4 Dimension Selection

Muszala *et al.* study using an application derived load metric for NAMD in [95], replacing the automatically measured timings of objects with an application specified parameter, the total atomic mass contained within a object, after identifying that it correlates with timed load. They show this to be beneficial in that it allows load balancing to happen immediately at launch time since no timesteps have to be instrumented to discover the load distribution and it removes the overhead of calling timers around the execution of every method. However, even with these benefits, using atomic mass as the load metric does not always outperform the original CPU time-based metric.

In [96], Mikida explores load balancing parallel discrete event simulations (PDES) using different application derived properties as the load metric. For optimistic PDES [97], CPU time is often not an accurate representation of load because it involves speculative execution, leading to rollbacks and reevaluation. Because of this speculation, an object that spends more time executing than another may not actually be doing more useful work than the other. The efficacy of the different load metrics tested in this work vary depending on the underlying model, there is not one universal best choice.

The unifying factor of these two examples is the use of application-derived metrics in load balancing to provide improvements over the default measurement-based load balancing. However, in both cases, the applicability and choice of alternative metrics is determined by manually modifying the application to use the new metric, and empirically testing and performing offline analysis of the resulting performance. This approach requires multiple training runs with the same configuration and input dataset and needs to be repeated for new inputs, wasting the limited compute time available to users on HPC systems.

One potential way to improve these cases is via *dimension selection.* In dimension selection, multiple potential load metrics are provided to the runtime system through a load vector, allow the runtime system to automatically perform the formerly manual evaluation of alternate metrics. The RTS can then dynamically detect if any of the provided metrics correlate well with CPU load and switch to using that instead to remove timer overhead, as done in the case with NAMD, or the RTS can test the impact of using different metrics in place of CPU load on the performance of the resulting mapping, automating the selection scheme done with PDES. Note that this would work with scalar load balancing; a load vector is presented to the RTS, but only one of these candidate loads is chosen to be used by the balancer.

Separately, there may be situations where using multiple metrics simultaneously with vector load balancing provides a benefit over what scalar load balancing can provide. Selecting multiple metrics that correlate with runtime or application progress and using them to construct a load vector for holistic balancing may more fully capture the properties of an application than a single one of them can. As suggested in [98], this is likely the case in PDES, where both differences in the amount of work per event and an uneven distribution of events contribute to load imbalance.

The application of online machine learning techniques to discover and predict which combinations of metrics can be used to represent load is another interesting area of exploration. In particular, one goal is to train a model to derive a function $f : \mathbb{R}^d \to \mathbb{R}^k$ with $k < d$, to map an input load vector to an output load vector of a reduced dimension, both reducing the dimensionality of the problem and doing so by combining the dimensions of the original input load vector in arbitrary ways.

8.2.5 Broadening Applicability

While we have developed the methods in this thesis for use with Charm++, the insidious problem of load imbalance afflicts parallel applications at large, not merely those using Charm++. None of the algorithms and techniques for vector load balancing explored here are inherently limited for use only with Charm++, and we have already demonstrated their applicability for VT and MPI (via Adaptive MPI) applications.

In personal discussions, the developers of AMReX [99] have expressed interest in using our vector load balancing methods to address imbalance they have observed in their own phase-based AMR applications, and there are many other applications that may benefit from the use of vector LB, but do not use Charm++ and are too large or established to be easily ported.

To this end, we intend to encapsulate our techniques and strategies into a library for general use. The library will contain only the load balancing algorithms themselves, with an API taking sets of objects/tasks and PEs as input and returning a balanced mapping as output. In order to use this library, users will have to implement statistics collection and migration support for their specific application or framework.

Outside of the context of load balancing parallel applications, multi-objective partitioning has been suggested for or used in fields as disparate as legislative redistricting [100]–[102], component layout for FPGAs [103], and offloading for mobile pervasive systems [104]. A commodity library allowing for the easy use of our methods by researchers outside of the HPC community may be beneficial for other such diverse uses, as well.

REFERENCES

[1] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[2] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, "Design of dynamic load-balancing tools for parallel applications," in *Proceedings of the 14th international conference on Supercomputing*, 2000, pp. 110–118.

[3] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of mpi+openmp applications," in *International Conference on Parallel Processing, 2004. ICPP 2004.*, 2004, 195–202 vol.1. DOI: `10.1109/ICPP.2004.1327921`.

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999, ISSN: 0004-5411. DOI: `10.1145/324133.324234`.

[5] R. H. Halstead, "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 9–17, ISBN: 0897911423. DOI: `10.1145/800055.802017`.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, 2 Feb. 2011. DOI: `10.1002/cpe.1631`.

[7] A. Duran, E. Ayguadé, R. M. Badia, *et al.*, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[8] L. Kale, B. Acun, S. Bak, *et al.*, *The Charm++ Parallel Programming System*, Aug. 2019. DOI: `10.5281/zenodo.3370873`.

[9] L. V. Kale and G. Zheng, "Chapter 1: The Charm++ Programming Model," in *Parallel Science and Engineering Applications: The Charm++ Approach*, L. V. Kale and A. Bhatele, Eds., 1st, Boca Raton, FL, USA: CRC Press, Inc., 2013, ch. 1, pp. 1–16, ISBN: 1466504129, 9781466504127. DOI: `10.1201/b16251`.

[10] A. Sinha and L. Kale, "A load balancing strategy for prioritized execution of tasks," in *[1993] Proceedings Seventh International Parallel Processing Symposium*, 1993, pp. 230–237. DOI: 10.1109/IPPS.1993.262887.

[11] R. K. Brunner and L. V. Kalé, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*, World Scientific Publishing, 2000, pp. 167–181.

[12] 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.

[13] 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 *2010 22nd International Symposium on Computer Architecture and High Performance Computing*, ISSN: 1550-6533, Oct. 2010, pp. 71–78. DOI: 10.1109/SBAC-PAD.2010.18.

[14] G. Zheng, M. S. Breitenfeld, H. Govind, P. Geubelle, and L. V. Kale, "Automatic dynamic load balancing for a crack propagation application," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 06-08, Jun. 2006.

[15] J. C. Phillips, D. J. Hardy, J. D. C. Maia, *et al.*, "Scalable molecular dynamics on cpu and gpu architectures with namd," *The Journal of Chemical Physics*, vol. 153, no. 4, p. 044 130, 2020. DOI: 10.1063/5.0014475.

[16] H. Menon, L. Wesolowski, G. Zheng, *et al.*, "Adaptive techniques for clustered n-body cosmological simulations," *Computational Astrophysics and Cosmology*, vol. 2, no. 1, pp. 1–16, 2015.

[17] W. Hong, "Exploiting inter-operation parallelism in xprs," *ACM SIGMOD Record*, vol. 21, no. 2, pp. 19–28, 1992.

[18] M. N. Garofalakis and Y. E. Ioannidis, "Scheduling issues in multimedia query optimization," *ACM Computing Surveys*, vol. 27, 1995.

[19] M. N. Garofalakis and Y. E. Ioannidis, "Multi-dimensional resource scheduling for parallel queries," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 365–376, Jun. 1996, ISSN: 0163-5808. DOI: 10.1145/235968.233352.

[20] C. Chekuri and S. Khanna, “On Multidimensional Packing Problems,” *SIAM Journal on Computing*, vol. 33, no. 4, pp. 837–851, Jan. 2004, Publisher: Society for Industrial and Applied Mathematics, ISSN: 0097-5397. DOI: 10.1137/S0097539799356265.

[21] L. Epstein and T. Tassa, “Vector assignment problems: A general framework,” *Journal of Algorithms*, vol. 48, no. 2, pp. 360–384, 2003, ISSN: 0196-6774. DOI: https://doi.org/10.1016/S0196-6774(03)00055-5.

[22] A. Meyerson, A. Roytman, and B. Tagiku, “Online Multidimensional Load Balancing,” en, in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 8096, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 287–302, ISBN: 978-3-642-40327-9 978-3-642-40328-6. DOI: 10.1007/978-3-642-40328-6_21.

[23] X. Zhu, Q. Li, W. Mao, and G. Chen, “Online vector scheduling and generalized load balancing,” en, *Journal of Parallel and Distributed Computing*, vol. 74, no. 4, pp. 2304–2309, Apr. 2014, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2013.12.006.

[24] Y. Azar, I. R. Cohen, and D. Panigrahi, “Randomized algorithms for online vector load balancing,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’18, USA: Society for Industrial and Applied Mathematics, Jan. 2018, pp. 980–991, ISBN: 978-1-61197-503-1.

[25] I. Cohen, S. Im, and D. Panigrahi, “Online Two-Dimensional Load Balancing,” in *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, A. Czumaj, A. Dawar, and E. Merelli, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 168, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 34:1–34:21, ISBN: 978-3-95977-138-2. DOI: 10.4230/LIPIcs.ICALP.2020.34.

[26] S. Taylor, J. Watts, M. Rieffel, and M. Palmer, “The concurrent graph: Basic technology for irregular problems,” *IEEE Parallel Distributed Technology: Systems Applications*, vol. 4, no. 2, pp. 15–25, 1996, Conference Name: IEEE Parallel Distributed Technology: Systems Applications, ISSN: 1558-1861. DOI: 10.1109/88.494601.

[27] J. Watts, M. Rieffel, and S. Taylor, “A load balancing technique for multiphase computations,” in *Proc. of High Performance Computing ‘97*, 1997, pp. 15–20.

[28] F. Pellegrini, “Scotch and PT-Scotch Graph Partitioning Software: An Overview,” in *Combinatorial Scientific Computing*, O. S. Uwe Naumann, Ed., Chapman and Hall/CRC, 2012, pp. 373–406. DOI: 10.1201/b11644-15.

[29] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–11. DOI: 10.1109/IPDPS.2007.370258.

[30] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 28–28. DOI: 10.1109/SC.1998.10018.

[31] R. Barat, C. Chevalier, and F. Pellegrini, "Multi-criteria graph partitioning with scotch," in *2018 Proceedings of the seventh SIAM workshop on combinatorial scientific computing*, SIAM, 2018, pp. 66–75.

[32] K. D. Devine, E. G. Boman, R. T. Heaphy, *et al.*, "New challenges in dynamic load balancing," en, *Applied Numerical Mathematics*, vol. 52, no. 2-3, pp. 133–152, Feb. 2005, ISSN: 01689274. DOI: 10.1016/j.apnum.2004.08.028.

[33] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[34] A. Duch, V. Estivill-Castro, and C. Martínez, "Randomized K-Dimensional Binary Search Trees," vol. 1533, Dec. 1998, pp. 199–208, ISBN: 978-3-540-65385-1. DOI: 10.1007/3-540-49381-6_22.

[35] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998, ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1997.1404.

[36] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[37] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Design Automation Conference*, 1982, pp. 175–181. DOI: 10.1109/DAC.1982.1585498.

[38] J. Bordner and M. L. Norman, *Computational cosmology and astrophysics on adaptive meshes using charm++*, 2018. DOI: 10.48550/ARXIV.1810.01319.

[39] J. H. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[40] P. P. Ewald, "Die berechnung optischer und elektrostatischer gitterpotentiale," *Annalen der Physik*, vol. 369, no. 3, pp. 253–287, 1921. DOI: `https://doi.org/10.1002/andp.19213690304`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.19213690304`.

[41] T. Darden, D. York, and L. Pedersen, "Particle mesh ewald: An n log (n) method for ewald sums in large systems," *The Journal of chemical physics*, vol. 98, no. 12, pp. 10 089–10 092, 1993.

[42] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger, "Object-Based Adaptive Load Balancing for MPI Programs," in *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, May 2001, pp. 108–117.

[43] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, Oct. 2003, pp. 306–322.

[44] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93, Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883, ISBN: 0818643404. DOI: `10.1145/169627.169855`.

[45] L. Kalé and A. Sinha, "Projections: A preliminary performance tool for charm," in *Parallel Systems Fair, International Parallel Processing Symposium*, Newport Beach, CA, Apr. 1993, pp. 108–114.

[46] 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 *2020 Workshop on Exascale MPI (ExaMPI)*, Nov. 2020, pp. 31–40. DOI: `10.1109/ExaMPI52011.2020.00009`.

[47] M. T. Bettencourt, D. A. Brown, K. L. Cartwright, *et al.*, "EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code," en, *Communications in Computational Physics*, vol. 30, no. 4, pp. 1232–1268, Jun. 2021, ISSN: 1815-2406, 1991-7120. DOI: `10.4208/cicp.OA-2020-0261`.

[48] T. T. P. Team, *The Trilinos Project Website*.

[49] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2014.07.003`.

[50] J. Lifflander, N. L. Slattengren, P. P. Pébaÿ, P. Miller, F. Rizzi, and M. T. Bettencourt, "Optimizing Distributed Load Balancing for Workloads with Time-Varying Imbalance," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, ISSN: 2168-9253, Sep. 2021, pp. 238–249. DOI: `10.1109/Cluster48925.2021.00039`.

[51] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in charm++," in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, © 2016 IEEE. Reprinted, with permission, 2016, pp. 40–43. DOI: `10.1109/ESPM2.2016.011`.

[52] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[53] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture."

[54] S. Donfack, S. Tomov, and J. Dongarra, "Dynamically balanced synchronization-avoiding lu factorization with multicore and gpus," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, 2014, pp. 958–965. DOI: `10.1109/IPDPSW.2014.109`.

[55] C. R. Ferreira and M. Bader, "Load balancing and patch-based parallel adaptive mesh refinement for tsunami simulation on heterogeneous platforms using xeon phi coprocessors," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '17, Lugano, Switzerland: Association for Computing Machinery, 2017, ISBN: 9781450350624. DOI: `10.1145/3093172.3093237`.

[56] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *European Conference on Parallel Processing*, Springer, 2009, pp. 863–874.

[57] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, IEEE Computer Society Press, 2012, p. 66.

[58] A. Duran, E. Ayguadé, R. M. Badia, *et al.*, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[59] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, Orlando, FL, USA: ACM, 2014, 273:273–273:283, ISBN: 978-1-4503-2670-4. DOI: `10.1145/2544137.2544163`.

[60] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load balancing in a changing world: Dealing with heterogeneity and performance variability," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: ACM, 2013, 21:1–21:10, ISBN: 978-1-4503-2053-5. DOI: `10.1145/2482767.2482794`.

[61] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," M.S. thesis, Dept. of Computer Science, University of Illinois, 2008.

[62] D. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012.

[63] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[64] D. J. Miller, P. M. Watts, and A. W. Moore, "Motivating future interconnects: A differential measurement analysis of pci latency," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '09, Princeton, New Jersey: Association for Computing Machinery, 2009, pp. 94–103, ISBN: 9781605586304. DOI: `10.1145/1882486.1882513`.

[65] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017. DOI: `10.1109/MM.2017.37`.

[66] V. García-Flores, E. Ayguade, and A. J. Peña, "Efficient data sharing on heterogeneous systems," in *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 121–130. DOI: `10.1109/ICPP.2017.21`.

[67] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, 2010, pp. 1–12.

[68] A. L. Fazenda, a. L. Mendes, L. V. Kale, J. Panetta, and E. R. Rodrigues, "Dynamic load balancing in gpu-based systems for a mpi program," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 154–161. DOI: `10.1109/HPCSim.2014.6903681`.

[69] R. Hagan and Y. Cao, "Multi-gpu load balancing for in-situ visualization," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, The Steering Committee of The World Congress in Computer Science, Computer ..., 2011, p. 1.

[70] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, San Antonio, TX, USA: Association for Computing Machinery, 2011, pp. 277–288, ISBN: 9781450301190. DOI: 10.1145/1941553.1941591.

[71] A. Acosta, V. Blanco, and F. Almeida, "Towards the dynamic load balancing on heterogeneous multi-gpu systems.," in *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '12, USA: IEEE Computer Society, 2012, pp. 646–653, ISBN: 9780769547015. DOI: 10.1109/ISPA.2012.96.

[72] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-gpu clusters," in *Proceedings of the Conference on High Performance Graphics*. Goslar, DEU: Eurographics Association, 2010, pp. 57–66.

[73] J. Teresco, J. Fair, and J. Flaherty, "Resource-aware scientific computation on a heterogeneous cluster," *Computing in Science & Engineering*, vol. 7, no. 2, pp. 40–50, 2005. DOI: 10.1109/MCSE.2005.38.

[74] J. Faik, "A model for resource-aware load balancing on heterogeneous and non-dedicated clusters," AAI3183624, Ph.D. dissertation, USA, 2005, ISBN: 0542251361.

[75] A. Merkel and F. Bellosa, "Balancing power consumption in multiprocessor systems," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06, Leuven, Belgium: Association for Computing Machinery, 2006, pp. 403–414, ISBN: 1595933220. DOI: 10.1145/1217935.1217974.

[76] O. Sarood and L. V. Kale, "A 'cool' load balancer for parallel applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: Association for Computing Machinery, 2011, ISBN: 9781450307710. DOI: 10.1145/2063384.2063412.

[77] O. Sarood, P. Miller, E. Totoni, and L. V. Kalé, ""cool" load balancing for high performance computing data centers," *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1752–1764, 2012. DOI: 10.1109/TC.2012.143.

[78] E. L. Padoin, M. Diener, P. O. A. Navaux, and J.-F. Méhaut, "Managing power demand and load imbalance to save energy on systems with heterogeneous cpu speeds," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2019, pp. 72–79. DOI: `10.1109/SBAC-PAD.2019.00024`.

[79] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8. DOI: `10.1109/CLUSTER.2013.6702627`.

[80] B. Acun, A. Langer, E. Meneses, *et al.*, "Power, reliability, and performance: One system to rule them all," *Computer*, vol. 49, no. 10, pp. 30–37, 2016. DOI: `10.1109/MC.2016.310`.

[81] M. H. Bremer, J. D. Bachan, and C. P. Chan, "Semi-static and dynamic load balancing for asynchronous hurricane storm surge simulations," in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2018, pp. 44–56. DOI: `10.1109/PAW-ATM.2018.00010`.

[82] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, IEEE, 1998, pp. 28–28.

[83] J. Bruno, E. G. Coffman, and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Commun. ACM*, vol. 17, no. 7, pp. 382–387, Jul. 1974, ISSN: 0001-0782. DOI: `10.1145/361011.361064`.

[84] I. Ahmad and A. Ghafoor, "A semi distributed task allocation strategy for large hypercube supercomputers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90, New York, New York, USA: IEEE Computer Society Press, 1990, pp. 898–907, ISBN: 0897914120.

[85] G. Zheng, A. Bhatelé, E. Meneses, and L. Kalé, "Periodic hierarchical load balancing for large supercomputers," English (US), *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 371–385, Nov. 2011, ISSN: 1094-3420. DOI: `10.1177/1094342010394383`.

[86] C. Mei, Y. Sun, G. Zheng, *et al.*, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, Seattle, Washington: ACM Press, 2011, p. 1, ISBN: 978-1-4503-0771-0. DOI: `10.1145/2063384.2063466`.

[87] S. Bak, H. Menon, S. White, M. Diener, and L. V. Kalé, "Multi-level load balancing with an integrated runtime approach," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, IEEE, 2018, pp. 31–40, ISBN: 978-1-5386-5815-4. DOI: 10.1109/CCGRID.2018.00018.

[88] J. D. Teresco, J. Faik, and J. E. Flaherty, "Hierarchical partitioning and dynamic load balancing for scientific computation," in *Applied Parallel Computing. State of the Art in Scientific Computing*, J. Dongarra, K. Madsen, and J. Waśniewski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 911–920, ISBN: 978-3-540-33498-9.

[89] 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, Delft, The Netherlands: Association for Computing Machinery, 2012, pp. 137–148, ISBN: 9781450308052. DOI: 10.1145/2287076.2287103.

[90] J. K. Uhlmann, "Satisfying general proximity / similarity queries with metric trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991, ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(91)90074-R.

[91] J. L. Bentley, "Decomposable searching problems," *Information Processing Letters*, vol. 8, no. 5, pp. 244–251, 1979, ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90117-0.

[92] B. Chazelle and L. J. Guibas, "Fractional cascading: I. a data structuring technique," *Algorithmica*, vol. 1, pp. 133–162, 1986.

[93] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, Sep. 2001, ISSN: 0360-0300. DOI: 10.1145/502807.502808.

[94] K. P. F.R.S., "Liii. on lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901. DOI: 10.1080/14786440109462720.

[95] S. Muszala, G. Alaghband, J. Hack, and D. Connors, "Natural Load Indices (NLI) in NAMD2 load balancing algorithms," en, UCAR/NCAR, Tech. Rep., Jul. 2014, Artwork Size: 578 KB Medium: application/pdf, 578 KB. DOI: 10.5065/D6TD9V99.

[96] E. P. Mikida, "Adaptive techniques for scalable optimistic parallel discrete event simulation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2019.

[97] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990, ISSN: 0001-0782. DOI: 10.1145/84537.84545.

[98] E. Mikida and L. Kale, "Adaptive methods for irregular parallel discrete event simulation workloads," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '18, Rome, Italy: Association for Computing Machinery, 2018, pp. 189–200, ISBN: 9781450350921. DOI: 10.1145/3200921.3200936.

[99] W. Zhang, A. Almgren, V. Beckner, *et al.*, "AMReX: A framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019. DOI: 10.21105/joss.01370.

[100] E. Rincón-García, M. Gutiérrez-Andrade, S. de-los-Cobos-Silva, P. Lara-Velázquez, A. Ponsich, and R. Mora-Gutiérrez, "A multiobjective algorithm for redistricting," *Journal of Applied Research and Technology*, vol. 11, no. 3, pp. 324–330, 2013, ISSN: 1665-6423. DOI: https://doi.org/10.1016/S1665-6423(13)71542-6.

[101] D. B. Magleby and D. B. Mosesson, "A new approach for developing neutral redistricting plans," *Political Analysis*, vol. 26, no. 2, pp. 147–167, 2018. DOI: 10.1017/pan.2017.37.

[102] R. Swamy, D. M. King, and S. H. Jacobson, "Multiobjective optimization for politically fair districting: A scalable multilevel approach," *Operations Research*, vol. 0, no. 0, null, 0. DOI: 10.1287/opre.2022.2311.

[103] N. Selvakkumaran, A. Ranjan, S. Raje, and G. Karypis, "Multi-resource aware partitioning algorithms for fpgas with heterogeneous resources," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04, San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 741–746, ISBN: 1581138288. DOI: 10.1145/996566.996768.

[104] S. Ou, K. Yang, and A. Liotta, "An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems," in *Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)*, 2006, 10 pp.–125. DOI: 10.1109/PERCOM.2006.7.