2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems

Enabling Work Migration in CoMD to Study Dynamic Load Imbalance Solutions

Olga Pearce*, Hadia Ahmed†, Rasmus W. Larsen‡ and David F. Richards*
*Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA 94550, USA
Email: olga@llnl.gov, richards12@llnl.gov
†Department of Computer and Information Sciences, University of Alabama at Birmingham, USA
Email: hadia@cis.uab.edu
‡Department of Computer Science, University of Copenhagen, Denmark
Email: rasmuswriedtlarsen@gmail.com

***Abstract*—We have enabled work migration in the CoMD proxy application to study dynamic load imbalance. Proxy applications are developed to simplify studying parallel performance of scientific simulations and to test potential solutions for performance problems. However, proxy applications are typically too simple to allow work migration or to represent the load imbalance of their parent applications. To study the ability of load balancing solutions to balance work effectively, we enable work migration in one of the ExMatEx proxy applications, CoMD. We design a methodology to parameterize three key aspects necessary for studying load imbalance correction: 1) the granularity with which work can be migrated; 2) the initial load imbalance; 3) the dynamic load imbalance (how quickly the load changes over time). We present a study of the impact of flexibility in work migration in CoMD on load balance and the associated rebalancing costs for a wide range of initial and dynamic load imbalance scenarios.**

I. Introduction

Modern scientific simulations rely on parallel computers to solve state of the art problems requiring vast computational resources. The largest supercomputers have millions of independent processors, and concurrency levels are rapidly increasing. For ideal efficiency, developers of the simulations that run on these machines must ensure that computational work is evenly balanced among processors. Dynamic load balancing is a way to correct the imbalances that arise throughout the application execution, and is increasingly important for overall application performance. To enable dynamic load balancing, applications must implement a mechanism for work migration. We present a methodology for studying how flexibility in work migration impacts the trade off between ability to balance the work effectively and the associated costs.

Proxy applications can make it easier to study performance of large simulations and try new solutions. However, as the result of simplification, many proxy applications are not suitable for studying load imbalance solutions because they do not implement work migration. In this work, we extend one of the ExMatEx proxy applications, CoMD, to enable work migration, and to represent dynamic load imbalance found in large scale molecular dynamics simulations. To mimic real simulations, we develop a methodology to set up initial load imbalance, and to dynamically control the imbalance as the simulation progresses. We enable flexibility in the granularity of work migration in CoMD, which allows us to study the impact of the work migration granularity on the ability to lower the imbalance in the simulation and the associated rebalancing costs.

In this paper, we describe the control mechanisms for 1) initial load imbalance, 2) dynamic imbalance, and 3) work migration granularity we introduced in CoMD. Together, these three aspects enable us to study the effectiveness of load balancing solutions and their costs when the application presents different imbalance behaviors (high/low initial imbalance, fast/slow rate of change in imbalance). We show that our version of CoMD is useful in evaluating work migration granularity and load balance algorithm performance for applications with different imbalance behaviors.

Our contributions in this paper are:

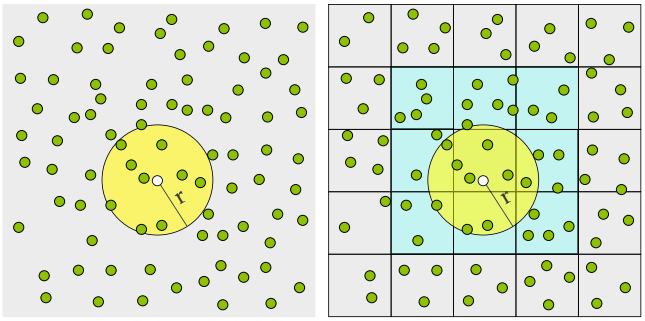
- A proxy application that allows us to study work migration granularity, load imbalance and potential solutions;
- Ability to control initial load imbalance, dynamic load imbalance, and flexibility in work migration;
- An evaluation of a load balance algorithm performance on a wide range of initial and dynamic load imbalance scenarios generated using our new proxy application.

Section II describes related work. Section III describes the original CoMD proxy application. Section IV describes how we create initial load imbalance in CoMD, and how we make load imbalance dynamic in CoMD. Section V outlines the changes we made to the implementation of CoMD to enable work migration. Section VI describes our interface for load balance algorithms. Section VII shows our results.

II. Related Work

Many simulations implement their own load balance algorithms that are tightly coupled with application data structures (i.e., ParaDiS [3]). Others rely on stand-alone libraries like graph partitioners (i.e., ParMetis [13], [14], Jostle [15], [16], and Zoltan [4], [5]). Testing whether their own or stand-alone load balance solutions can correct load imbalance is non-trivial for a production application as enabling work migration can involve significant changes to the application data structures. Our work aims at providing a testbed to enable evaluation of load balance solutions prior to performing the work necessary

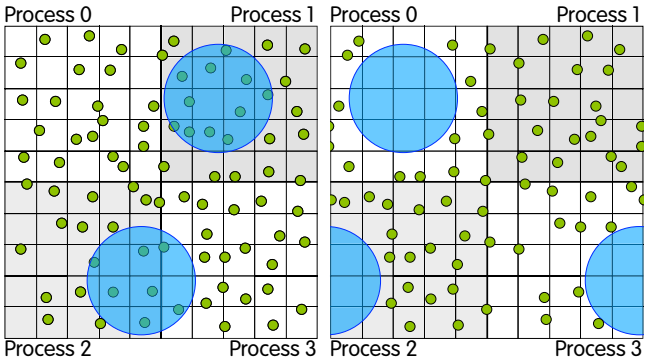
978-1-5090-5218-9/16 $31.00 © 2016 IEEE
DOI 10.1109/PMBS.2016.15



(a) Cutoff Radius Defines Atom Interactions to Compute

(b) Regular Grid Defines Proximity of Atoms in Simulated Space

Fig. 1: Cutoff Radius and Regular Grid



(a) Initial Imbalance Created by Removing Atoms within Randomly Placed Spheres

(b) Dynamic Imbalance Created by Shifting Atoms Over Time

Fig. 2: Introducing Load Imbalance in CoMD

to enable work migration, and informing decisions about granularity of work migration necessary for the load balance solutions to be successful.

Some applications have the option of using AMPI to overdecompose the domain [12] and then balance load by moving virtual processors from overloaded physical processors to the underloaded ones. This approach is application agnostic and based solely on runtime information, but can impose extra communication overhead for tightly coupled applications due to the increase in the surface to volume ratio of the smaller domains. Our work enables testing of load balance algorithms that can rebalance the application based on the input from the application.

LeanMD [9] is a parallel molecular dynamics simulation framework written in Charm++ that exhibits load imbalance. While it is useful for exploring the built-in Charm++ load balance algorithms, it does not provide the control mechanisms to set up different load balance scenarios, which are the main contribution of this work. Similarly, the AMR mini-app implemented in Charm++ and miniAMR from the Mantevo suite [7] do not have a mechanism to control the imbalance.

The Particle-in-cell (PIC) Parallel Research Kernel (PRK) [6] is a paper-and-pencil specification of the computational task to be computed. PIC PRK was developed to help measure the efficiency and effectiveness of dynamic load balance techniques. Similarly to our work, they introduce different imbalance scenarios to test load balance algorithms. Particle distribution in PRK can be exponential, sinusodial, or linear, resulting in different initial load imbalance. To simulate dynamic imbalance, particles can be uniformly injected or removed throughout execution. Besides representing a different class of simulations, our work goes a step further by explicitly parameterizing the initial and dynamic load imbalance. We also parameterize work assignment granularity to allow studying how flexibility in work assignment impacts the ability to rebalance the application.

III. CoMD: ExMatEx Proxy App

Molecular dynamics is an important class of simulations that evaluate the forces the atoms in the system exert on each other over time. CoMD [1] is an ExMatEx proxy application designed to represent this class of simulations.

Because atoms that are far apart have little effect on each other, classical MD simulations frequently use atom interaction models that define the force between two atoms to be zero when their separation distance exceeds a cutoff radius. This reduces the complexity of the force calculation to $O(n)$. Figure 1a shows a selected atom and a circle with the cutoff radius which defines the range of interaction. To simplify finding which of the atoms are within the cutoff radius, CoMD divides the simulation space into cells that are no smaller than the cutoff radius, as shown in Figure 1b. This cell definition guarantees that the atoms within the cutoff radius from an atom are either in the same cell as the atom, or in the immediately surrounding cells.

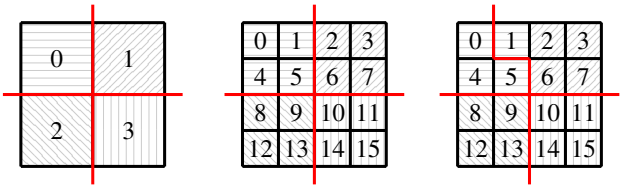
CoMD has an MPI version which implements a Cartesian spatial decomposition of atoms across processes, with each process responsible for computing forces and evolving velocities and positions of all atoms within its domain boundaries. As atoms move across domain boundaries, they are handed off from one process to the logically neighboring process. To compute forces between atoms on different domains, CoMD uses ghost cells, or copies of the immediately neighboring cells residing on the logically neighboring processes. Similar to other molecular dynamic simulations, CoMD uses periodic boundaries, allowing the atoms near the boundaries to interact with the atoms on the opposite side of the simulated space.

IV. Introducing Load Imbalance in CoMD

In this section, we describe how we introduce initial load imbalance in CoMD, and how we ensure the load imbalance changes throughout the simulation.

A. Initial Load Imbalance

The reference implementation of CoMD evenly divides the simulated space between processes. Because the atoms are



(a) Original CoMD: Single Domain per Process **(b)** Modified CoMD: Multiple Domains per Process **(c)** Modified CoMD: Enabled Work Migration

Fig. 3: Using Domain Overdecomposition to Enable Load Balancing

```
1: for timesteps do
2:     Execute application iteration on the local domain
3:     Communicate with neighbors
```

Fig. 4: Algorithm: Standard Proxy App

```
1: for timesteps do
2:     for all local domains do
3:         Execute application iteration on the local domain
4:         Communicate with neighbors (local and remote)
```

Fig. 5: Algorithm: Overdecomposed Proxy App

uniformly spaced at setup, the atoms and the number of atom interactions that need to be computed are also evenly divided between processes, making the simulation inherently balanced. To introduce imbalance, we introduce voids in the simulated structure by removing some atoms from the simulation. Figure 2a illustrates a four-process problem with the atoms shown in green. We randomly place spheres in the simulated space, shown in blue, and remove all the atoms that fall inside the spheres. While the spheres are not part of the simulation, we use the concept of the spherical voids in this work to reason about the atom spacing in the simulated space. After atom removal, the number of atoms assigned to each process is different, and so is the number of atom interactions the processes will compute. The spherical void size, the sphere count, and a random seed for generating the coordinates for the sphere center, are user-specified runtime parameters.

B. Dynamic Load Imbalance

To create a version of molecular dynamics with a dynamic load imbalance, we introduce the ability to set a non-zero center of mass velocity for the simulation. This causes the atoms to gradually shift in any of the three spatial dimensions (or combination of them) by a fixed distance at every time step. Figure 2b shows an example of moving all the atoms in Figure 2a horizontally. Now Process 0 gets fewer atoms to work with, while Process 1 gets more.

Because MD is translationally invariant, this overall shifting the atoms preserves the number of force interactions between the atoms but changes which process has to compute the interactions. We implemented the center of mass velocity as a runtime parameter; the user indicates the dimension(s) along which the atoms should be shifted, and the shift distance in Angstroms per time step for the selected dimension(s). Atoms can be shifted by a different distance in each dimension. This simulates a production application where amount of work per process changes over time throughout the application runtime.

V. Enabling Work Migration in CoMD

The reference implementation of CoMD cannot be load balanced because of several design decisions. CoMD decomposes the simulated space into the same number of domains as processes, and assigns one domain per process. The domains are rectangular prisms of the same size, and the user cannot control the size or the shape of the domains. Figure 3a shows the simulation space in CoMD decomposed into four equal-sized rectangular domains; one domain is assigned to each process. Once the domains are assigned to processes, the assignment cannot be changed, so dynamic changes in work cannot be reflected in the work assignment.

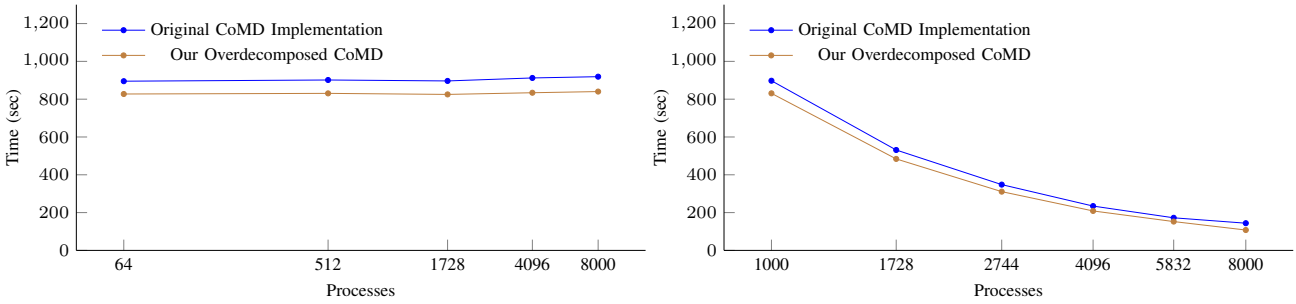
We relaxed these design decisions by overdecomposing the application into more domains than available processes. Figure 3b shows the same simulation space decomposed into 16 domains; the domains are then assigned to four processes. However, if one process has more work than others, the assignment might be more balanced if we move one of the domains to another process, as shown in Figure 3c.

In our implementation, we introduce a data structure to store the assignment of domains to processes. Our *domain graph* consists of vertices which represent domains, and edges which represent boundaries with logically neighboring domains. The domain graph is distributed between processes, and each process stores local domains and edges to local and remote logically neighboring domains.

Figure 4 shows pseudocode for the existing CoMD implementation; each process performs computation on a single domain. Figure 5 shows pseudocode for our overdecomposed implementation. Each process is now responsible for computing the forces on one or more domains, and communicating the velocity and position updates accordingly.

We evaluated the overhead of overdecomposing CoMD to ensure that we have not substantially changed the performance of the proxy application. While we have looked at the performance in significant detail [8], Figure 6 summarizes the impact on the overall runtime. Figure 6a shows weak scaling with 256K atoms per process, while Figure 6b shows strong scaling with 256M atoms overall. Both figures demonstrate that the runtime is slightly shorter with our overdecomposed implementation. This may be due to the fact that in the original implementation, the atoms are stored in long lists; these lists are broken into shorter lists in our overdecomposed approach. Because we are computing atom-atom interactions for atoms in different lists, our overdecomposed approach has the cache effect similar to that of blocking for matrix-matrix multiplication. Aside from this cache effect, our implementation does not significantly change the performance of CoMD.

We implement overdecomposition granularity as a runtime parameter to let us study the trade off between load balance quality and cost. One hypothesis is that applications with



(a) Weak Scaling, 256K Atoms Per Process

(b) Strong Scaling, 256M Atoms

Fig. 6: Total Runtime (1000 timesteps) for Original CoMD vs. Overdecomposed CoMD

moderate imbalance should be possible to balance even when work assignment granularity is large (i.e., 8 domains per process). Applications with more severe imbalance or more dynamic imbalance may require a finer granularity of work assignment; in this case, the higher cost of rebalancing may be justified by more flexibility in making a more balanced work assignment. Our proxy application allows us to easily study these trade offs.

VI. Load Balance Algorithm Interface

Our overdecomposed implementation of CoMD maintains a distributed domain graph where the vertices represent domains, and edges represent boundaries with logically neighboring domains. A vertex in the domain graph represents the smallest migratable unit in the application; its optional weight represents the amount of work in that domain. In this paper, we estimate the work in the domain as the number of interactions computed for the domain, which has been shown to be a useful estimate for molecular dynamics simulations [10].

Our distributed domain graph can be easily translated to CSR format, and used as an input to generic load balance algorithms like graph partitioners (i.e., ParMetis [13], [14], Jostle [15], [16], and Zoltan [4], [5]). The output of a load balance algorithm is an assignment of domains to processes. Our implementation sends the domain to its new process and updates the domain graph.

For the results shown in this paper, we used a simple load balance algorithm based on a spatial sort. First, the domains are spatially sorted using a Hilbert curve or Morton curve based on the region of simulated space they represent. Next, the curve is partitioned between the processes, taking domain weights into consideration. This algorithm has the complexity of a sorting algorithm in terms of the number of domains sorted. So far, we used a sequential implementation of the algorithm, necessitating reduction of the relevant information to a single process. In the future, we plan to use a variety of more sophisticated and parallel load balance methods.

Our domain graph is a suitable interface with load balance algorithms since many load balance algorithms use graphs as a representation of the application communication.

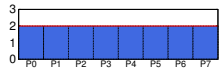
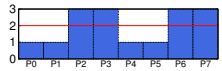
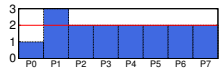
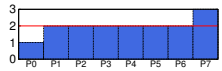
	Load on each Process	L_max	L_ave	Imbalance
(a)		2	2	0%
(b)		3	2	50%
(c)		3	2	50%
(d)		3	2	50%

TABLE I: Example Load Distributions and Their Imbalance

VII. Results

In this section, we evaluate how different work assignment (overdecomposition) granularity, initial load imbalance, and velocity of shifting the atoms in the simulation impact the ability of a load balance algorithm to correct the imbalance. We study the trade-offs of load balancing accuracy and rebalancing costs.

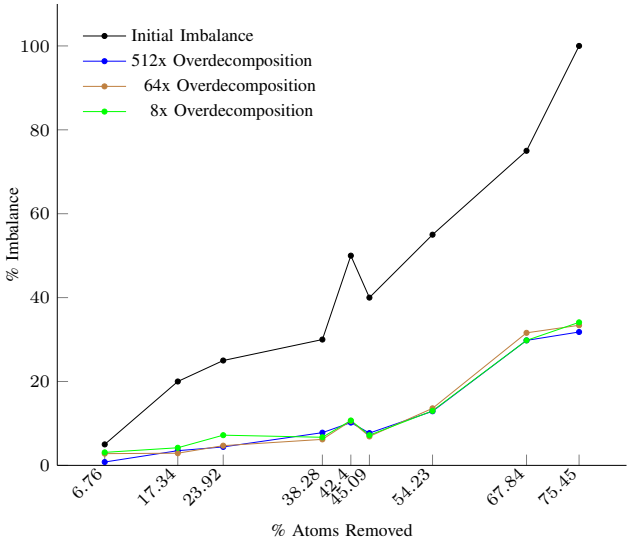
For our experiments, we use a Linux cluster with nodes consisting of two 2.8 GHz Hex-core Intel Xeon EP X5660 processors, twelve cores per node. All nodes are connected by QDR Infiniband. We use GCC 4.4.7 and MVAPICH v0.99 on top of CHAOS, an HPC variant of RedHat Enterprise Linux (RHEL), running at Linux kernel v2.6.32.

For our measurements, we use Caliper [2], a generic context annotation tool developed at LLNL. Using Caliper, we annotate the phases of execution in the application, and measure how much time the phases take at each time step. The per-timestep measurements allow us to observe the performance changes in the simulation over time.

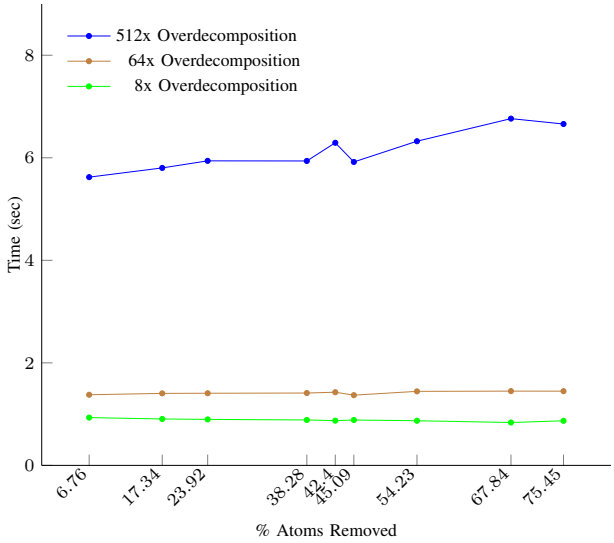
We define the load of an MPI process in a timestep as the total time minus the time waiting at MPI synchronizations:

$$L_{process} = T_{total} - T_{sync} \quad (1)$$

We use Caliper's MPI service to measure the time each process spends in MPI_Barrier, MPI_Allreduce, MPI_Alltoall, and MPI_Waitall functions at each timestep. T_{sync} in Equation 1 is simply the total time the process spends in MPI synchronizations at each timestep.



(a) Varying Load Balancing Difficulty



(b) Load Balance Algorithm Time + Redistribution Time

(c) Detailed Parameters and Statistics

Prob -lem	% Atoms Removed	NumAtoms Remaining	Number of Interactions	Spherical Void		%Imbalance Before LB	% Imbalance After LB		
				#/Process	Radius(Angstr.)		512xOV	64xOV	8xOV
1	6.76	21,750,645	467,711,537	10	19.3	5	0.8	2.8	3.1
2	17.34	19,283,754	414,158,794	2	46.3	20	3.5	2.9	4.2
3	23.92	17,747,675	380,027,720	2	52.4	25	4.4	4.7	7.2
4	38.28	14,397,791	301,885,115	8	39.5	30	7.8	6.2	6.7
5	42.40	13,431,088	283,106,849	4	52.4	50	10.2	10.6	10.7
6	45.09	12,809,523	266,446,859	10	39.5	40	7.7	6.9	7.2
7	54.23	10,677,200	220,856,392	8	46.3	55	12.9	13.6	13.0
8	67.84	7,501,551	153,090,962	8	52.4	75	29.8	31.6	29.8
9	75.45	5,727,095	115,381,879	10	52.4	100	31.8	33.4	34.1

Fig. 7: Initial Load Imbalance Scenarios in CoMD (64 Processes, 23M Atoms Initially)

For this paper, we define load *imbalance* as the scaled maximum load on any process minus the average:

$$Imbalance = \left(\frac{L_{max} - L_{ave}}{L_{ave}}\right) \times 100\%, \qquad (2)$$

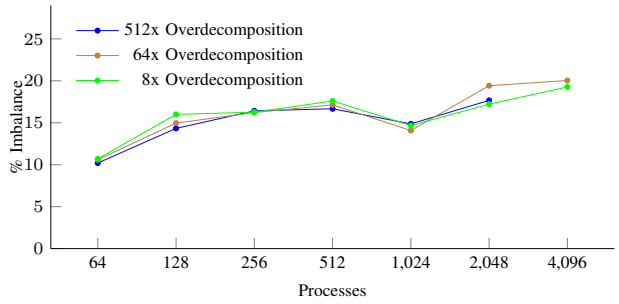
where L is the load from Equation 1. The definition in Equation 2 represents opportunity cost of load balancing, as shown in Figure I [11]. Example (b) shows process loads with 50% imbalance, and because the maximum process load is 3 units, the execution time is 3 units, which is 50% longer than the execution of the balanced example (a) (2 units).

A. Creating Initial Load Imbalance

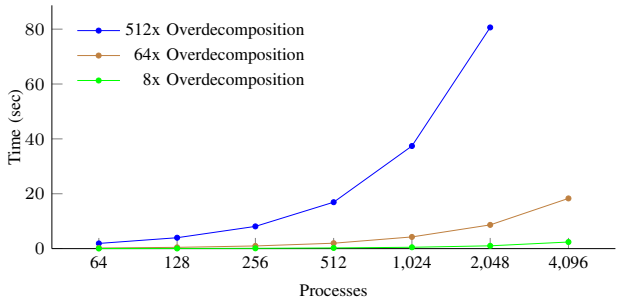
The first control mechanism for our study is introducing initial load imbalance before running the simulation. For these experiments, we started by generating 23,328,000 atoms on 64 processes. The simulated space is a cube with a side length of 650.24 Angstroms; each process starts with a cube with a side length of 162.56 Angstroms. We varied the number and size of spherical voids to generate a set of problems with different number of atoms and load imbalance for our experiments.

Figure 7 shows a set of problems with different configurations for spherical voids. We varied the number of voids from 2 to 10 per process. We varied the radius of the voids from 19.3 to 52.4 Angstroms. The parameterized atom removal resulted in a different number of atoms for each problem. Figure 7 shows the resulting initial imbalance for each problem, as defined by Equation 2. Generally, large spherical voids lead to less uniform distribution of atoms in the problem and therefore higher imbalance and difficulty in load balancing.

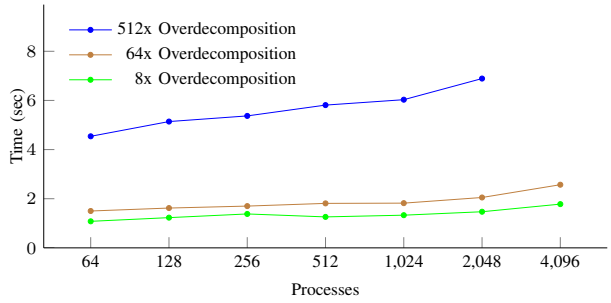
We overdecomposed the generated set of problems and load balanced them. We experimented with three granularities of overdecomposition: 8 domains per process (8x overdecomposition), 64 domains per process (64x overdecomposition), and 512 domains per process (512x overdecomposition). Decomposing the problem into more domains gives us more flexibility in terms of work assignment since it allows us to migrate smaller portions of the problem space. Figure 7 shows



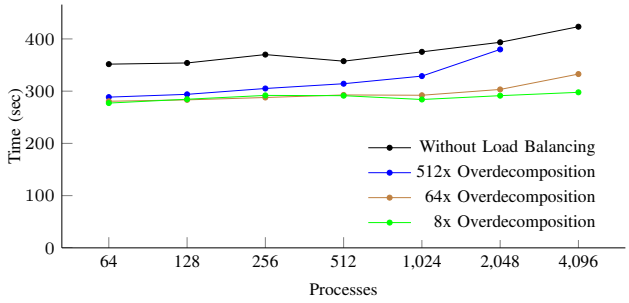
(a) Imbalance After Load Balancing



(b) Runtime of the Load Balance Algorithm



(c) Redistribution Time



(d) Total Simulation Runtime

(e) Detailed Parameters and Statistics

Num Procs	Num Atoms Remaining	Number of Interactions	% Imb bef. LB	% Imbalance After LB			Total Runtime			
				512xOV	64xOV	8x OV	No LB	512xOV	64xOV	8x OV
64	13,431,088	283,106,849	50.70	10.20	10.60	10.70	351.70	288.59	280.66	277.20
128	26,505,376	558,243,612	50.30	14.33	14.96	16	353.98	293.91	283.32	284.52
256	53,190,202	1,120,473,002	55.76	16.42	16.22	16.26	370.02	305.10	287.86	291.86
512	106,273,179	2,239,029,303	50.55	16.67	17.13	17.6	357.41	314.23	292.73	291.33
1,024	212,741,503	4,482,828,235	58.00	14.87	14.08	14.67	375.26	328.83	292.16	284.03
2,048	425,692,895	8,971,434,404	65.00	17.67	19.42	17.22	393.54	379.90	303.28	291.36
4,096	852,665,073	17,971,364,315	63.00	NA	20.04	19.27	423.40	NA	332.71	297.77

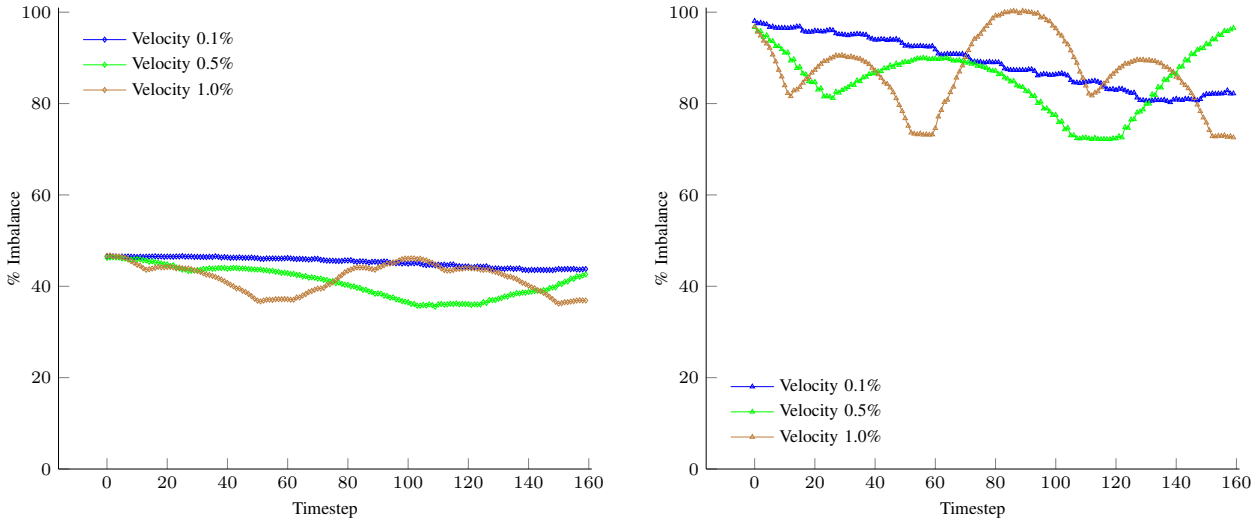
Fig. 8: Impact of Load Imbalance and Rebalancing Costs on Total Runtime (Weak Scaling)

the load imbalance for each granularity of overdecomposition after rebalancing. The problems with higher initial imbalance still had higher imbalance even after the rebalancing step, as compared to the problems with lower initial imbalance. This is due to the fact that large spherical voids result in problems with less uniform atom distribution, which are more challenging to load balance. Different granularities of overdecomposition result in roughly the same load imbalance for each problem, with smaller granularities of overdecomposition mostly achieving slightly lower imbalance due to having more flexibility in work assignment. In some instances, smaller granularity of overdecomposition does not result in lower imbalance, although the difference is negligible and is due to the fact that the work cannot be evenly divided between processes even with smaller granularity.

We used our set of problems with different initial load imbalance and parameterized overdecomposition granularities to look at the costs of rebalancing and redistributing work. Figure 7b shows that higher granularities of overdecomposition result in higher costs of rebalancing, due to the finer granularity of work assignment. Additionally, the cost of rebalancing was slightly higher in problems with more severe imbalance as more domains were migrated during rebalancing.

While Figure 7a suggests slightly better load balance with smaller granularities of overdecomposition, Figure 7b shows the higher costs of overdecomposing into smaller domains. We will look at these costs in more detail in Section VII-B as we examine weak scaling of our approach and overall runtimes of the problems.

Figure 7c details the runtime parameters (number and size of spherical voids) for each problem shown in Figures 7a and 7b. It also shows the number of remaining atoms and initial load imbalance as well as the number of interactions each problem computes in the initial timestep. We also list the resulting load



(a) Problem 5 (13M Atoms, 283M Interactions, 64 processes, 50% Initial Imbalance)

(b) Problem 9 (5.7M Atoms, 115M Interactions, 64 processes, 100% Initial Imbalance)

Fig. 9: Effect of Velocity With Which the Atom Shift on Load Imbalance

imbalance for each granularity of overdecomposition.

Our ability to vary the initial load imbalance allows us to evaluate how well a load balance method can correct the imbalance for a given overdecomposition granularity.

B. Weak Scaling

To evaluate the impact of scale on the performance and accuracy of a load balance algorithm, we performed a weak scaling study. For these experiments, we use Problem 5 in Figure 7c. This problem has 4 spherical voids with a radius of 52.4 Angstroms per process, and an initial imbalance of 50%. We weak scale Problem 5 from 64 to to 4,096 processes by proportionally increasing the simulated space and keeping the size and the number of spherical voids per processor the same. Figure 8e lists the number of atoms, interactions, and initial imbalance at each scale. Load imbalance goes up slightly at higher scales because differences in process loads become more pronounced at scale.

Figure 8 demonstrates the impact of load imbalance and load balancing costs on the total runtime of the simulation when weak scaling the problem. We ran each problem for 160 timesteps, and rebalanced in the third timestep.

Figure 8a illustrates the ability of our load balance algorithm to load balance the problem; we show the load imbalance right after rebalancing when using 8x, 64x, and 512x overdecomposition. For all three granularities of overdecomposition, the load balance algorithm is successful at reassigning work in a more balanced manner. Because the difficulty of load balancing increases with scale even in the weak scaling case, we see that the post-rebalancing imbalance is higher at higher scale. While the results are similar for all three levels of overdecomposition, 512x overdecomposition is slightly better due to more flexibility in work assignment. However since 512x overdecomposition deals with fine-grained data, it uses more memory for bookkeeping and runs out of memory on 4,096 processes when using our sequential load balance algorithm.

Figure 8b shows the runtime of our load balance algorithm. Because we only have a sequential algorithm in place at the moment and need to gather/scatter its input/output, its execution time grows as the problem is weak scaled. The execution time grows much faster with scale for finer granularity of overdecomposition because the number of domains in the problem is the input size to the load balance algorithm.

Figure 8c shows the time for redistributing the domains in the simulation after the load balance algorithm determines the new domain assignments. The higher the overdecomposition, the more domains there are in the problem, and therefore the longer it takes to redistribute the simulation.

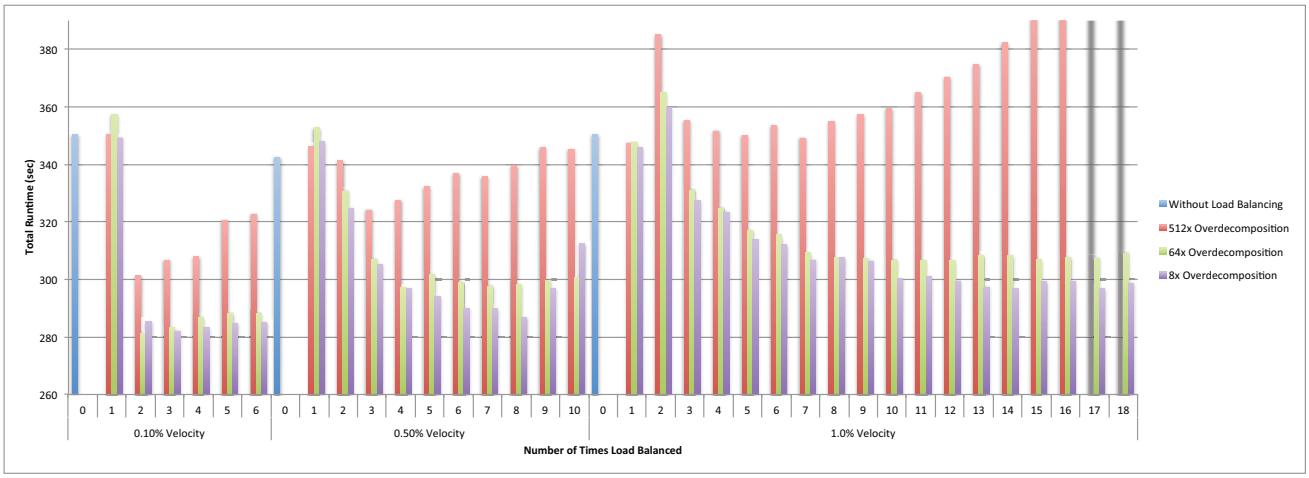
Figure 8d shows the total runtime of the simulation over 160 timesteps. Problems run without load balancing take the longest to finish. Problems run with 8x overdecomposition plus load balancing are the fastest especially at higher scale.

Because we only have a sequential load balance algorithm that requires reduction of all necessary data to a single process, 512x overdecomposition used too much memory and resulted in a slow execution time of the load balance algorithm, outweighing the benefits of achieving lower imbalance.

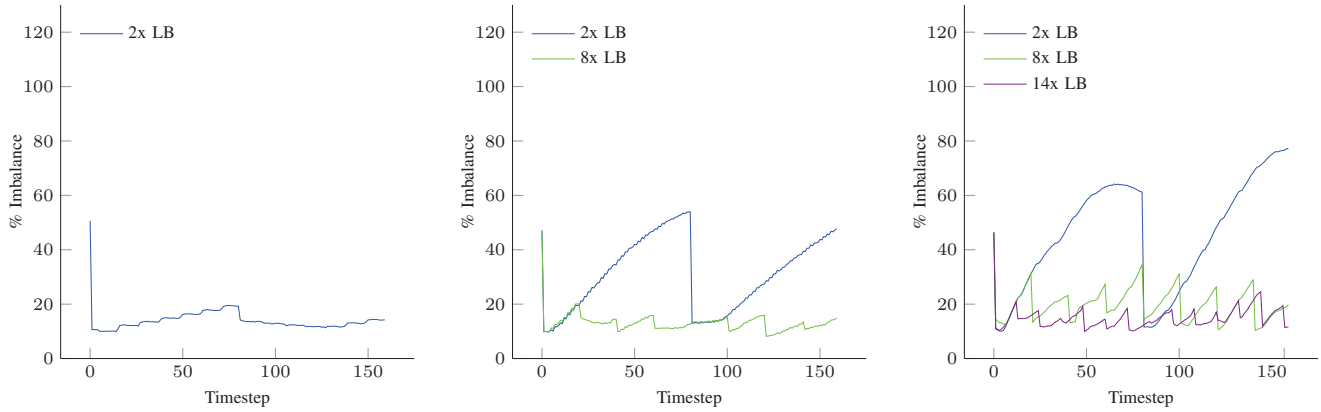
Our proxy application allowed us to study performance and accuracy of a load balance algorithm as the number of processes increased, as well as the associated costs.

C. Dynamic Imbalance

To allow dynamic work changes and to study their effect on the ability to load balance the simulation, we experiment with our control mechanism of shifting atoms in the simulated



(a) Total Runtime for Different Velocity, Frequency of Load Balancing, and Overdecomposition (Problem 5)



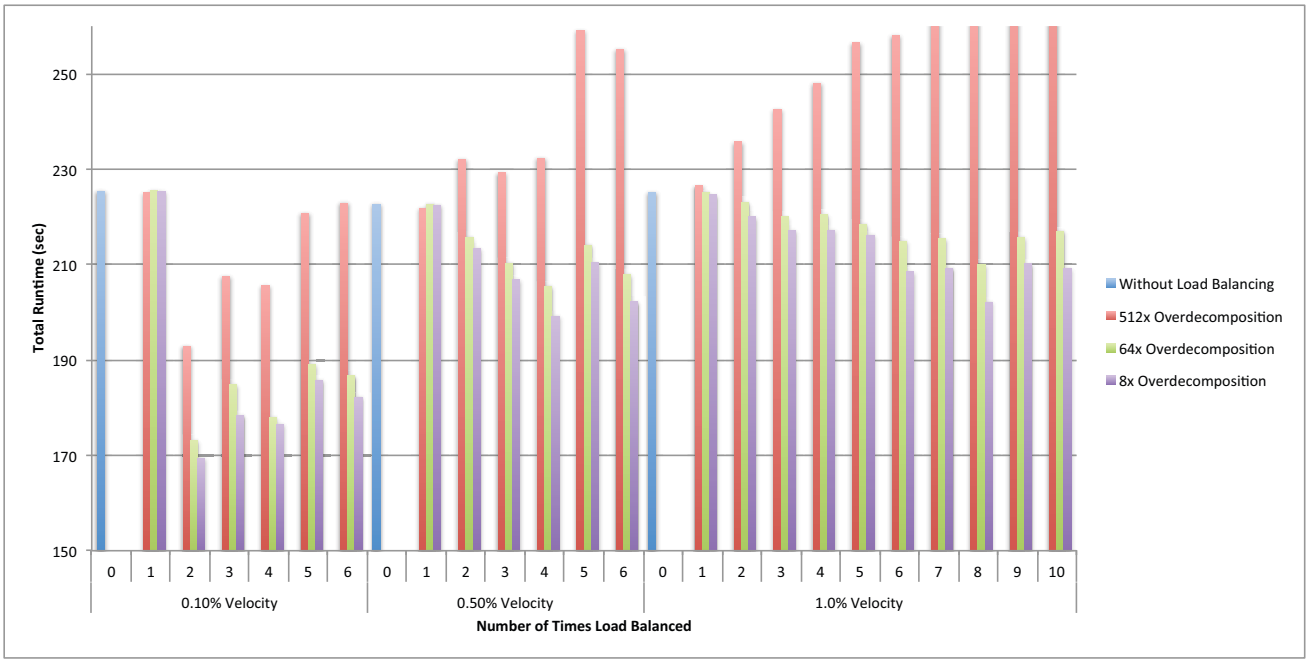
(b) 0.1% Velocity **(c)** 0.5% Velocity **(d)** 1.0% Velocity

Fig. 10: Effect of Frequency of Rebalancing on Load Imbalance, Problem 5 (13M Atoms, 283M Interactions, 64 processes, 50% Initial Imbalance)

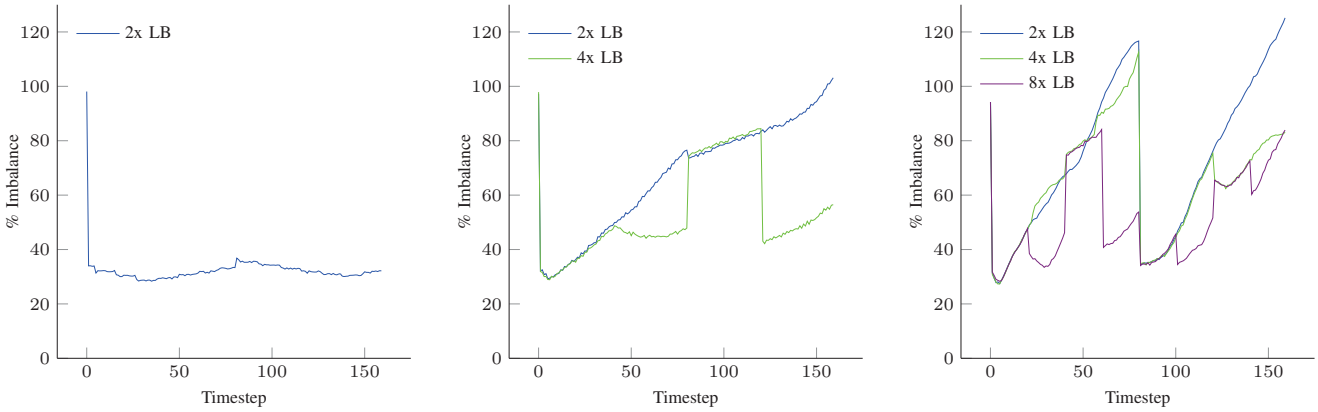
space. For these experiments we compare Problems 5 and 9 in Figure 7c on 64 processes. Problem 5 has 4 spherical voids with the radius of 52.4 Angstroms per process; it has 13,431,088 atoms and computes 283,106,849 atom-pair force interactions, and 50% initial imbalance. Problem 9 has 10 spherical voids with the radius of 52.4 Angstroms per process; it has 5,727,095 atoms and computes 115,381,879 atom-pair force interactions, and 100% initial imbalance. In addition to being more imbalanced, Problem 9 performs significantly less computation than Problem 5. For both problems, we use three granularities of overdecomposition, and run the load balance algorithm at even intervals with different frequencies.

Figure 9 shows the effect of shifting the atoms in the simulation space on the load balance. We ran all problems for 160 timesteps. We show three center of mass velocities: 0.1%, 0.5%, and 1%, where 1% velocity means each timestep the atoms were shifted by 1% of the length of the simulation space in one dimension of the problem. The change in the imbalance is small when the atoms are shifted slowly, and a faster shift of atoms results in a larger variation in imbalance. The imbalance can both decrease and increase because the atoms in the problem will shift from one process to the next, changing the amount of computation each process performs. For Problem 5, the variation in imbalance as the atoms shift is not large, because each process does a significant amount of work even as the atoms shift, as shown in Figure 9a. For Problem 9, the variation in imbalance is larger, because there are fewer atoms overall and the impact of them shifting to other processes is greater, as shown in Figure 9b.

Figures 10 and 11 show the total runtimes for Problems 5 and 9, and how imbalance in Problems 5 and 9 evolves when they are rebalanced with different frequency. Here we define frequency of rebalancing as the number of times the problem was rebalanced during the execution, so a frequency of 2x means the problem was rebalanced twice, namely at timestep 1 and timestep 81. We ran the problems for 160 timesteps. We



(a) Total Runtime for Different Velocity, Frequency of Load Balancing, and Overdecomposition (Problem 9)



(b) 0.1% Velocity **(c)** 0.5% Velocity **(d)** 1.0% Velocity

Fig. 11: Effect of Frequency of Rebalancing on Load Imbalance, Problem 9 (5.7M Atoms, 115M Interactions, 64 processes, 100% Initial Imbalance)

show rebalancing at equal intervals with different frequencies that result in the best runtime (as demonstrated in Figure 10a) for 0.1%, 0.5%, and 1.0% velocities of shifting atoms. We show the runtimes for different velocities of shifting atoms, and different frequencies of rebalancing. Minimizing the total runtime requires appropriately evaluating the tradeoffs between the costs and benefit of rebalancing with different frequency; we intend to use our version of CoMD for developing and evaluating models which handle the trade off for different imbalance scenarios and rebalancing costs.

Figure 10a shows the total runtimes for Problem 5. Figure 10b shows that load imbalance increases slowly when center of mass velocity is 0.1%, and Figure 10a confirms that load balancing twice results in the lowest execution time (64x overdecomposition). The imbalance increases faster when the shift velocity is 0.5%, as shown in Figure 10c, so load balancing 8 times results in the lowest execution time (8x overdecomposition). Load imbalance increases most dramatically when the atoms are shifted with velocity of 1.0%, as shown in Figure 10d, necessitating rebalancing 14 times (8x overdecomposition).

Figure 11a shows the total runtimes for Problem 9. Figure 11b shows that load imbalance increases slowly when shift velocity is 0.1%, and load balancing twice results in the

lowest execution time (8x overdecomposition). The imbalance increases faster when the shift velocity is 0.5%, as shown in Figure 11c, so load balancing 4 times results in the lowest execution time (8x overdecomposition). For this instance of the problem (0.5% velocity), the state of the application at step 80 is such that the load balance algorithm does not improve the imbalance; in fact, load imbalance increases; we are still investigating why this happens. Load imbalance increases most dramatically when the atoms are shifted with velocity of 1.0%, as shown in Figure 11d, requiring rebalancing 8 times for the best runtime (8x overdecomposition). Similarly to the 0.5% velocity case, for 1.0% velocity case, there are points in the problem when rebalancing increases the load imbalance (i.e., at timestep 40); we are still investigating the causes. Because there is less work per process in Problem 9 as compared to Problem 5, higher imbalance can be tolerated and fewer rebalancing steps can be amortized over the duration of the problem.

TABLE II: Details for Best Case Execution for Different Velocities

Problem \ Velocity		0.1%	0.5%	1.0%
Problem 5 (50%)	Overdecomp.	64x	8x	8x
	LB Frequency	2x	8x	14x
	Time (sec)	281.1	286.8	296.9
Problem 9 (100%)	Overdecomp.	8x	8x	8x
	LB Frequency	2x	4x	8x
	Time (sec)	169.4	199.16	202.1

Table II shows details for the lowest runtimes in each velocity/frequency category. As velocity increases, the benefit of increasing the rate of rebalancing can outweigh the rebalancing cost, resulting in overall improvement in total runtime of the simulation.

Our ability to shift atoms in the simulation over time allows us to study how dynamic work changes effect our ability to load balance the simulation for a given overdecomposition granularity.

VIII. Conclusions

We extended an ExMatEx proxy application, CoMD, to be a suitable testbed for assessing the ability of load balance algorithms to correct dynamic load imbalance. We designed a methodology to parameterize three key aspects necessary for studying load imbalance correction: the granularity with which work can be migrated, the initial load imbalance, and how quickly the imbalance changes throughout the simulation. We presented a study demonstrating our ability to use our modified version of CoMD to evaluate the effectiveness of a load balance algorithm and the associated rebalancing costs for a wide range of initial and dynamic load imbalance scenarios.

IX. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-704368).

References

[1] Official website for CoMD. http://www.exmatex.org/comd.html.

[2] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimnez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*, Salt Lake City, Utah, USA, November 13-18, 2016.

[3] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *SC'04*, November 2004.

[4] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, J. Teresco, J. Faik, J. Flaherty, and L. Gervasio. New Challanges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[5] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *International Conference on Supercomputing (ICS)*, May 2000.

[6] E. Georganas, R. F. V. der Wijngaart, and T. Mattson. Design and Implementation of a Parallel Research Kernel for Assessing Dynamic Load-Balancing Capabilities. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016.

[7] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[8] R. W. Larsen and O. Pearce. Enabling Load Balancing in CoMD through Overdecomposition. Technical Report LLNL-TR-678978, Lawrence Livermore National Laboratory, November 2015.

[9] V. Mehta. LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[10] O. Pearce, T. Gamblin, B. R. de Supinski, T. Arsenlis, and N. M. Amato. Load Balancing N-Body Simulations with Highly Non-Uniform Density. In *International Conference on Supercomputing (ICS)*, June 2014.

[11] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *International Conference on Supercomputing (ICS)*, June 2012.

[12] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kalé. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, October 2010.

[13] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations. In *SC'00*, November 2000.

[14] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *International Euro-Par Conference on Parallel Processing*, August 2000.

[15] C. Walshaw and M. Cross. Parallel Optimization Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[16] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.