Load Balancing Solutions in Distributed Large-Scale Applications

- literature study -

Jelle van Dijk

1 INTRODUCTION

The growing demand for computational resources is driving the development of large-scale distributed systems. Each year these systems increase in size, complexity and availability. If we want to benefit from this growing supply of computational resources we need to design parallel applications that can efficiently utilise distributed hardware.

Parallelism is a well-known technique to speed-up application execution: divide the work among multiple workers, have them work in parallel, and combine their results in the end. Most parallel applications attempt to achieve parallel performance as close to linear as possible, i.e., doubling the processors should half the execution time. To do this we need to utilise as much of the available hardware as possible. Good hardware utilisation improves application performance, as well as scaling behaviour when more hardware becomes available.

Achieving good hardware utilisation on a homogeneous system running a simple kernel is trivial: give every processor an evenly sized chunk of the work. For real-world applications, however, it quickly becomes more complicated than this. Systems are often not homogeneous, and workloads don't always divide in chunks that are, and stay, fair. Still, even more complex applications, like scientific simulations or distributed machine learning codes, want the benefits of good hardware utilisation. Therefore, finding efficient ways to distribute the work among all workers becomes a key challenge in parallelising such applications.

To improve hardware utilisation in parallel applications we look at *load balancing*. Load balancing is the process of (re)distributing work across multiple processors, with the goal of roughly equalising the load per processor. Load balancing methods can vary greatly between applications. To achieve a balanced load some applications might only need to split the input evenly across the workers. Some applications - like, like, for example, processing users accessing a website - may need a system that balances the load at runtime [1].

All parallel applications need some form of load balancing. This means that a wide variety of application domains have to address the challenge of load balancing. Even tough load balancing is a known problem in many domains, and we have found studies discussing different load balancing methods in the same domain [21, 27], we have found no work that compares load balancing across domains. Therefore, in this literature study we compare load balancing solutions from different domains across computer science. Specifically, we aim to ask the following research question:
What are the similarities and differences between load balancing solutions in different domains and applications?

To answer this question we compare 10 papers from 5 different domains. Each paper discusses a load balancing solution for a application on a distributed platform. We present these papers in Section 3. In Section 4, we compare the load balancing methods described by the core papers in order to answer the research question.

2 BACKGROUND

In this study we focus on load balancing in distributed systems. Specifically, in our context, distributed systems are platforms consisting of multiple independent processing units. Although all platforms we discuss share this common feature, the platforms may differ in, for example, size, heterogeneity and communication costs. To keep a consistent terminology across the different domains, we will refer to each individual processing unit as a worker.

The goal of multiple workers is to jointly solve the same workload, i.e., the problem at hand. To do so each worker is given a task, which is a part of the workload. For example, consider an application when an entire dataset must be processed. Each worker will have to process a part of the dataset. For the remainder of this study, we refer to a *job* as an entire application, whose workload is split into tasks to be executed by the workers. In the selected papers, jobs are stand alone, i.e., they do not require any communication with other jobs. Tasks from the same job do use communication to process the workload. Moreover, how a job/task is executed on a single worker is considered beyond the scope of this analysis - i.e., we do not discuss the scheduling done per worker by the operating system.

2.1 Terminology from literature

In order to make clear comparisons between methods, it is important to have a common terminology. This is especially useful when combining methods originating in different domains, which (may) use different terminology to talk about the same concepts. A taxonomy for load balancing in distributed systems, proposed by Casavant and Kuhl [6], can help us in creating this common terminology. Casavant and Kuhl [6] proposes to categorise different methods for load balancing in distributed systems. From this work we will mention some important concepts that will be used in describing and comparing the different methods later on.

2.1.1 Static or Dynamic load balancing. The most general classification we can use to distinguish different load balancing methods is to define them as *static* or *dynamic*. A *static load balancer* will distribute the workload (fairly) among the workers during a preprocessing step. This means that each worker will receive a fixed part of the workload, which is expected to stay constant throughout execution of the application. Static load balancing is used when a workload can be distributed fairly based on the application and input data alone. If this is not the case, or if the load can change during execution, i.e, a initially fair distribution will grow imbalanced over time, dynamic load balancing is used. *Dynamic load*

balancing can redistribute the workload at runtime. This means that a perfectly balanced load distribution is not strictly needed before execution: even if/when the load distribution is imbalanced, or grows imbalanced over time, the dynamic load balancer will correct it.

Dynamic load balancing is usually done in three steps. First, performance information is gathered for each worker, e.g., execution time or amount of communication. Next, the empirical data is used to estimate the load for each worker. In the third and final step, the workload is migrated based on the load estimations.

Using a dynamic load balancing system will, in most cases, result in more overhead than a static approach. However, this overhead can be negligible compared to the performance improvement received from balancing the load.

Finally, it is important to realise that dynamic load balancers also require a static step for initial work distribution. A good static load balancer can greatly reduce the amount of work done by the dynamic load balancer.

2.1.2 Response time. . Response time is an important performance metric for load balancing. The response time of a job or task is the total time taken for this job or task to finish, which includes running the job, waiting for resources, and communication with the workers. In general, the load balancing methods we analyse in this study aim to minimise the average response time of all the jobs or tasks in the system[1].

2.2 Paper Selection

To answer the proposed research question we look for published research describing load balancing for distributed application across different domains. To find relevant papers we use the academic search engine *google scholar* combined with the search queries presented in Table 1. A paper is considered relevant if, there is a direct mention of *load balance*, load balancing is applied specific application or domain and the related application is distributed. To limit the results we also only consider papers published after the year 2000. This gives a selection of 17 papers across 11 domains, see Table 2.

Table 1: Search Queries used to find relevant papers

Search Query
Load balancing survey
Load balance workload characterisation
Load balancing scientific workloads
Load balancing in scientific simulations
Large-scale distributed application
Large-scale parallel application
Load balancing in large-scale applications

Because 17 papers is to much to address in this study we further narrow down the selection. The *cosmological Simulation* domain is eliminated because of its overlap with *n-body simulations*. All but one of the domains that are represented by a single paper are also eliminated. This gives us a selection of 10 papers in 5 distinct domains represented by multiple papers, or a single paper discussing multiple load balancing methods. The final selection of domains is *fog computing*, *graph processing*, *n-body simulations*, *molecular dynamics* and *particle-in-cell simulations*.

3 PAPER ANALYSIS

In this section, we discuss each of the core papers individually, including a brief analysis of the strengths, weaknesses, and characteristics of each load-balancing method we encountered. In Section 4 we will compare the methods described in this section.

3.1 Fog Computing

Cloud computing offers computing resources on demand, an efficient alternative to owning and maintaining hardware. However, centralised clusters of computing resources are not ideal for latency-sensitive applications. Fog computing aims to tackle this problem by providing a cloud-like model on the edge of the network, i.e., closer to the end user. In fog computing, the edge of network is assumed to consist of many heterogeneous geographically distributed nodes [4]. These workers are capable of running a wide variety of applications. In this study we focus on the execution of stand-alone applications, here after called jobs. Jobs are scheduled once, and do not require any communication with other jobs.

Scheduling each application on the worker closest to user can lead to significant performance degradation due to load imbalance. If an applications needs to wait for resources, scheduling it on a worker further away might improve performance. This, together with handling the heterogeneity of the system, are some of the challenges that need to be addressed in load balancing for fog computing.

3.1.1 Load balancing aware scheduling algorithms for fog networks [25]. Singh and Auluck [25] propose multiple load balancing aware methods for scheduling latency-sensitive applications in the fog. The goal of these scheduling methods is to minimise the average response time of the jobs on the system. To achieve this, three heuristic methods are proposed. *Minimal Distance* (MD) places the work on the closest worker; if this worker is already overloaded, the jobs will wait until the resources are available. *Minimal Load* (ML) scheduling will place work on the worker with the lowest current utilisation. Lastly, there is *Minimal Hop Distance and Load* (MHDL) scheduling, where the workers that are one hop away are ranked based on utilisation, and the work is placed on the least utilised worker. If all workers at this distance are fully loaded, or if they are not available for some other reasons, the hop distance is increased by one and the procedure is repeated.

Singh and Auluck [25] experimentally compare the performance of the different methods by running simulations with a dataset based on the real usage of a cloud/fog system. Experiments are run with varying communication delays and a different number of workers. These experiments show that, across all the different experiments, MHDL has, on average, the lowest response time. MD performs better in situations where the communication delay is high. With high communication, waiting for resources becomes less significant, meaning that choosing the worker closet to the user is beneficial.

Singh and Auluck [25] also propose a non-heuristic scheduling method. This method looks at all possible scheduling configurations

Table 2: Collected literature divided into relevant domains.

Domain	Ref	Title	Author
Fog Computing	[25]	Load balancing aware scheduling algorithms for fog networks	Singh and Auluck
Graph processing	[14]	Mizan: a system for dynamic load balancing in large-scale graph processing	Khayyat et al.
	[29]	Gemini: A Computation-Centric Distributed Graph Processing System	Zhu et al.
N-body simulations	[17]	K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical N-body simulations	Marzouk and Ghoniem
	[20]	Load balancing n-body simulations with highly non-uniform density	Pearce et al.
Cosmological Simulation	[13]	Massively parallel cosmological simulations with ChaNGa	Jetley et al.
	[26]	The cosmological simulation code gadget-2	Springel
Molecular Dynamics	[12]	GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation	Hess et al.
	[3]	Dynamic topology aware load balancing algorithms for molecular dynamics applications	Bhatelé, Kalé, and Kumar
	[5]	Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters	Bowers et al.
Particle-in-Cell	[10]	The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing	Germaschewski et al.
	[19]	OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations	Nakashima et al.
IoT	[21]	A comprehensive and systematic review of the load balancing mechanisms in the Internet of Things	Pourghebleh and Hayyolalam
Agent-based Simulations	[8]	Distributed Load Balancing for Parallel Agent-Based Simulations	Cosenza et al.
Numerical Simulations	[18]	Dynamic Load Balancing for Parallel Numerical Simulations Based on Repartitioning with Disturbed Diffusion	Meyerhenke
Web Servers	[27]	Comparison of Load Balancing Strategies on Cluster-based Web Servers	Teo and Ayani
Machine learning	[15]	Distributed machine learning load balancing strategy in cloud computing services	Li et al.

and picks the optimal one, i.e., where the average response time of all the jobs is minimised. This method produces lower average response times, if the scheduling overhead is ignored. But the the work required for computing the best possible configuration makes this method perform significantly worse than the heuristic methods. For example, when more than 70 jobs have to be scheduled, the non-heuristic scheduling method is unable to compute a scheduling solution before the cut of time of $2x10^7 msec$.

The scheduling methods explored in [25] are designed to handle heterogeneous platforms. This is firstly done through the use of utilisation to describe the load of a worker. The use of utilisation gives each worker a load, without assuming computational power of that worker. Moreover, each worker is given a security tag, jobs containing sensitive information are scheduled based on these tags. This tag system enables the load balancer to take specific worker characteristics into account.

In summary, according to Singh and Auluck [25], a heuristic scheduler shows more usability in a real life situation. Using utilisation to determine the load per worker results in good load balancing. However for the best average response time, both utilisation and distance have to be taken into account.

3.2 Graph Processing

Graphs are a very powerful abstraction to model real world relationships between objects. Using graphs and graph processing we can solve a wide range of problems, e.g., finding a shortest path between cities or ranking web pages based on importance. As the amount of data we have grows, graphs representing the real world also grow. If we want to keep up with the growth in data, we need faster graph processing systems. Parallelism offers a solution for increasing graph processing speed. In order to utilise the distributed platforms, graph processing problems need to be disturbed across workers.

However, simply dividing the graph over multiple workers is not good enough if we want good performance. This is due to graph characteristics (i.e., connectivity, topology, density, clustering coefficient, etc.) that make it difficult for graph processing systems to balance load across workers, resulting in poor performance scaling. Balancing load is difficult because the data-driven nature of graph algorithms make it hard to predict computational work based on vertices and edges. The other problem is that graphs are often unstructured with poor data locality, meaning that creating a fair distribution based on data partitioning is also non-trivial[16]. For these reasons, load balancing for graph processing is a non-trivial

task, that needs to be addressed if we want to keep up with the growing supplies of data.

3.2.1 Mizan: a system for dynamic load balancing in large-scale graph processing [14]. Mizan is a graph mining system built on top of Pregel, a large-scale graph processing model. Unlike some other Pregel implementations [11, 7, 23], Mizan does not assume any prior knowledge of the graph structure of algorithm behaviour. Mizan's execution (much like Pregel's) is based on the Bulk Synchronous Parallel (BSP) model, meaning that there is a computation step followed by a global synchronisation step. In Mizan, the synchronisation step is also used as a load balancing step: during synchronisation, load imbalance is detected and, if necessary, the workload is rebalanced.

Load balancing in Mizan is based on statistical analysis of the system's performance. During the computation step, each worker measures the number of incoming messages, number of outgoing messages, and response time, i.e., the time it takes to process a message per vertex. During the bulk synchronisation step load imbalance is detected by looking at the summarised statistics, see Figure 1 for a schematic representation of the program flow.

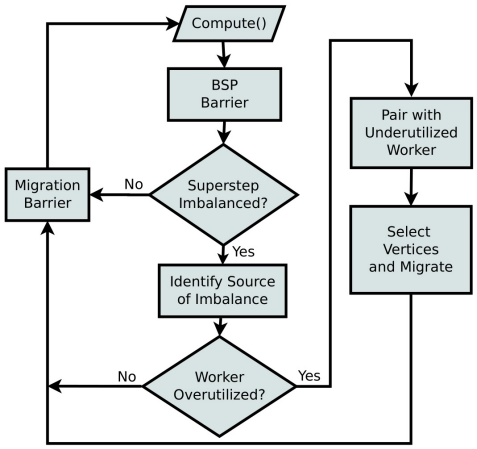

Figure 1: Summary of Mizan's Migration Planner, taken from [14]

Based on the gathered performance statistics, each worker calculates a z-score, i.e., its deviation from the average performance. If the z-score is above a predefined threshold, the worker is either over- or under-utilised; overutilised workers indicate load imbalance in the system. If load imbalance is detected, load is balanced by means of vertex migration between pairs of workers. Specifically, workers are paired based on utilisation, such that overutilised ones can migrate vertices to underutilised ones. For each pair, half of the difference in workload is migrated to the underutilised worker. Which vertices are migrated depends on the *migration goal*, i.e., which of the metrics is targeted to improve performance. The migration goal is determined by looking at the correlation between the number of messages and the response time, which indicates the performance bottleneck. If one of the message metrics is strongly correlated with the response time, it becomes the migration goal; otherwise, the goal defaults to response time.

To test performance, Khayyat et al. [14] used the following experimental setup. Experiments where run two different clusters, one with 21 workers and the other with 1024. Three load balancing methods are compared: static pre-processing, work stealing, and the load balancing method discussed above. The input data is a randomly generated graph, and a generated graph representing a real life social graph. Khayyat et al. [14] show that Mizan can scale linearly up to 1024 workers. The Mizan load balancing method outperforms pre-processing oriented methods, with up to 84% performance improvement.

3.2.2 Gemini: A Computation-Centric Distributed Graph Processing System [29]. Zhu et al. [29] found that shared memory graph processing systems were still able to outperform many of the distributed implementations. To address this they present Gemini, a distributed graph processing code.

Load balancing in Gemini is done through static chunk distribution. A chunk is a continuous piece of the input data. Chunks are continuous to preserve as much of the locality in the original data as possible. Each worker is assigned one chunk. For all vertices connected to the chunk, i.e., all vertices not in the chunk that are reachable by following one edge, mirror vertices are created. These mirror vertices will gather and collect the updates of the connected vertices and send it to the worker holding the original vertex. For an example of chunk distribution see Figure 2

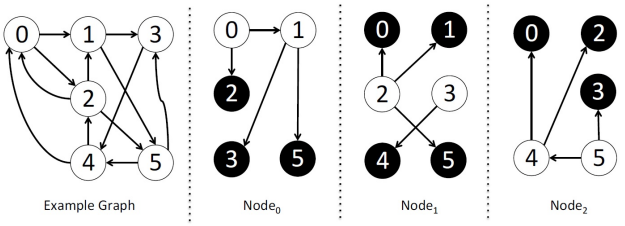

Figure 2: Example of possible chunk partitioning of a graph. White and black nodes represent original and mirrored nodes respectively. Image from [29].

Zhu et al. [29] show that deciding where to split based solely on either the number of vertices or edges is not ideal. Focusing on evenly distributing one of the two will create an imbalanced load. In order to decide the cut-off point, a dedicated metric, based on both the amount of vertices and the amount of edges, is used:

$$\alpha \cdot |V_i| + |E_i^D|$$

Here $|V_i|$ and $|E_i^D|$ represent the number of vertices and the number of outgoing edges in chunk i respectively. The variable α is set to $8 \cdot (p-1)$ by default, but can be altered depending on the system specifications. The cutoff point is chosen such that this value is balance between all workers.

Performance is evaluated on a 8 node cluster. Experiments performed by running a pagerank algorithm on twitter data from 2010 show that Gemini's method for chunk balancing reduces runtime by up to 1.8 times compare to focusing on either the number of vertices or edges. Compared to a few other partitioning schemes, most notably random and hash based partitioning, Gemini's chunk-based

partitioning still performs significantly better. Zhu et al. [29] do acknowledge that the different algorithms and data sources might not show the same results. The results show that, on large graphs, scalability can get close to linear up to 8 nodes. On smaller graphs, however, a slow-down can be observed.

3.3 N-body simulations

N-body simulations aim to track the evolution of dynamic particle systems. The particles, or bodies, often interact with each other, e.g., through gravitational forces. Simulating these interactions allows us to show how a system will evolve over time. A good example of an n-body simulation is a simulation of celestial objects, such as planets in a solar system.

Computing the forces between each and every pair of particles gives us a computational complexity of $O(N^2)$. To reduce this complexity, many algorithms exist that estimate the evolution of a system based on grouping particles in clusters. Treecode algorithms, like the Barnes-Hut algorithm, subdivide the particles in regions, which are organised in a tree, where a parent is a cluster from a larger region, and its children are the sub-clusters from the subregions resulting after the division. By simulating particles interacting with non-leaf nodes, or clusters, instead of the individual particles, the computation complexity of the algorithm is reduced [2]

Such treecode algorithms create a load balancing problem: not all particles have the same number of interactions. This means that simply giving each worker the same number of particles will not create a balanced load. Coupled with the possibility of non-uniform particle distribution, this means that simple particle or spacial partitioning gives no guarantee of load balance [17].

3.3.1 K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical N-body simulations [17]. Marzouk and Ghoniem [17] introduce and n-body system for vortex simulations. To partition the input over the workers, a k-means clustering method is proposed. K-means clustering attempts to iteratively optimise a cost function by mapping N observations to k clusters. The N observations are the input particles of the simulation, these are mapped to the k workers. The mapping and cluster location is chosen to minimise the following cost function[1]:

$$J = \sum_{j=1}^{k} \sum_{i}^{|X_j|} |X_i - y_j|^2$$

Here, X_j and y_i represent the particles in cluster j and the cluster centroid of cluster j, respectively. The goal of k-means clustering is to minimise the squared distance between each particle and its cluster centre. This creates a cluster of localised particles on each worker. This localisation, coupled with a treecode estimation algorithm, will reduce communication and improve simulation efficiency.

Even though k-means clustering creates nice localised clusters, there is no guarantee of load balance. Cluster sizes are not necessarily equal, and no method for interaction balancing between workers is present. On top of the decomposition problems, vortex simulations present another issue which increased load imbalance: in order to maintain proper resolution throughout the simulation, new particles are introduced during each timestamp. These particles will introduce load imbalance, even if we manage to create a balanced load at the start of the simulation.

To solve these load imbalance problems, Marzouk and Ghoniem [17] implemented an extra dynamic load balancing system. The same k-means clustering method is applied with a small alteration to the cost function:

$$J = \sum_{j=1}^{k} \sum_{i}^{N_j} S_{k_j} |X_i - y_j|^2$$

Here, S_{k_j} is the scaling factor for cluster j. The scaling factor is based on the workers deviation from the mean execution time of the last timestep. If a worker is overloaded, the scaling factor is increased. The result of a higher scaling factor is that fewer particles are assigned to that worker, and more to underloaded workers, see Figure 3. After each simulation time step, the clusters are calculated again, and particles are redistributed. Due to the nature of k-means clustering, each recalculation of the clusters will require less computation, due to the faster convergence. Because k-means clustering only finds a local optimum, the cluster can be very dependent on the initial cluster guesses. To avoid bad clustering, new initial cluster are chosen if load imbalance exceeds a predefined threshold.

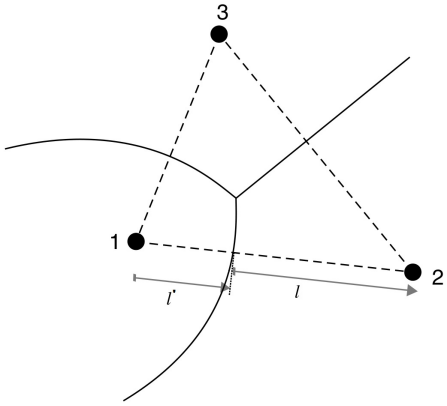

Figure 3: Two-dimensional schematic of scaled k-means cluster boundaries, with three clusters. The numbered solid circles are cluster centroids. Clusters have scaling factors s_i; here, $s_1 > s_2 = s_3$ and $l > l'$. Adaptation of an image from [17].

The performance of the k-means clustering load balancing method is measured on a machine with 1024 workers. Load imbalance is defined as the time taken by the longest running worker divided by the average time per worker, for one timestep. Marzouk and Ghoniem [17] show that on a system with 1024 workers, running a simulation with $N \approx 1.2million$ nodes, load imbalance typically stays below 1.5. Parallel efficiency can be as high as 98%, while for simple domain distribution is around 20%.

Marzouk and Ghoniem [17] also observe that load imbalance and parallel efficiency, although related, are not necessarily the

[1] Simplified version. For the original cost function see [17].

same. In a single time step, when the dynamic load balancing has not had enough time to fully balance the load, load imbalance can be poor compared to a simpler decomposition method, but parallel efficiency can still be high, because of the improved localisation provided by the k-means clustering.

3.3.2 Load balancing n-body simulations with highly non-uniform density [20]. The load balancing method for n-body simulations proposed by Pearce et al. [20] is based on the observation that both particle and spatial decomposition will result in load imbalance when the bodies are non-uniformly distributed in space. Instead, interactions are a good indication of load: computing interactions usually dominates computation, and each interaction requires approximately the same amount of computation. However, interaction partitioning can become an expensive operation, due to the high number of interactions in a system.

To reduce the overhead of evenly partitioning interactions, Pearce et al. [20] propose adaptive sampling to create evenly sized work units. A work unit contains a set of interactions, and is represented by a single interaction and its centre point, the mid point between all particles involved in the interaction.

The work units are generated such that all of them contain approximately the same amount of interactions. This is achieved through adaptive sampling. From each particle p_j $max(1, s \times |i_{p_j}| / count_{avg})$ particles are sampled, where s, $|i_{p_j}|$ and $count_{avg}$ represent the adaptive sampling threshold, the number of interactions for particle p_j, and the average number of interactions per particle, respectively. These interactions will be the basis of new work units. Each non-sampled interaction is assigned to the work unit where the centre point is closest to it's own centre point. Work units containing a higher than average number of interactions are subdivided into multiple work units. The adaptive sampling threshold s can be changed to adjust the coarseness of the work units.

From all the work units and particles, a hypergraph [2] is created. In this hypergraph, particles are represented by edges, and work units by vertices. A hypergraph partitioning algorithm then divides the work across the workers. The goal of this partitioning algorithm is to create partitions with an even number of vertices in each partition, and to minimise the number of edges spanning multiple partitions. Because work units are created to contain the same amount of work, this will balance the load across the workers.

To evaluate the performance of their load balancing system, Pearce et al. [20] use the following definition of load balance:

$$\text{imbalance} = \frac{\text{maximum load} - \text{average load}}{\text{average load}}$$

Here, load is the amount of interactions per worker. All experiments are run on 8 to 2048 workers, with 32K particles. Results show that choosing the correct sample rate is important. A sampling rate of around or below 1% seem to work best; increasing too much above 1% will create more load balancing overhead, due to the higher number of work units, but the imbalance will not decrease as significantly. Pearce et al. [20] show a performance increase of up to 26% compared to a simple Barnes-Hut implementation.

[2] A hypergraph is a generalisation of a graph where each edge can connect multiple vertices together

3.4 Molecular Dynamics

Molecular dynamics simulations are used to analyse the interaction of atoms and molecules overtime. Due to the high number of interactions and the small time steps (10^{-15} secs) simulating a millisecond of a biomolecule can take years on a single processor [3]. To allow for the simulation of larger molecules over a longer time parallel execution of the code becomes unavoidable.

The challenge comes from the behaviour of the particles. Non-uniform particle distribution can complicate both spacial and particle decomposition. Both a chunk of spatial domain or a set of particles is not guaranteed to relate to the same amount of work. As well as difficulty in partitioning the input, a molecular dynamics load balancer needs to take particle movement into account. The movement of particles can shift load overtime or slow down computation through reducing favourable locality [12].

3.4.1 GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation [12]. GROMACS is a distributed moleculere dynamics code, Hess et al. [12] show the work done to improve parallel performance of GROMACS through, among other things, better load balancing. Three main causes of load imbalance are observed by Hess et al. [12]: (1) an uneven distribution of particles in the simulation space, (2) the difference in interactions density, i.e., some particle interactions take more time to compute, and (3) the statistical fluctuation of the number of particles in a domain.

The first two causes of load imbalance might be addressed by a simple static load balancer, although it is non trivial to build one. However, to address particle fluctuation, a dynamic load balancing method is employed. First, each worker is given a 3D chunk of the input domain. These chunks start out evenly-sized, and are composed of charge groups. A charge group is a set of particles which together are neutrally charged, and it also represents the unit that is migrated between workers. During execution, these charge groups can be migrated between workers to achieve a balanced load. Approximately every 10 time steps, execution monitoring information is gathered from all workers. This information is used to determine how the chunks can be resized to achieve a better load balance. Resizing is done staggeringly across the dimensions, i.e., x boundaries are moved first, followed by y and z. How far the boundaries are moved is based on the amount of load imbalance; however, no clear definition of load balances given by the authors.

A different form of load imbalance is introduced with the Particle Mesh Ewald (PME) calculations. The PME calculations use 3D fast Fourier transforms, which require all-to-all communications. This means that running the PME calculations in a distributed manner introduces significant communication overhead, which is bad for parallel performance and scaling. To combat this issue, GROMACS provides the ability to assign dedicated PME workers. By using a smaller set of workers, the communication is reduced significantly. However, choosing the wrong number of PME workers can create load imbalance. Hess et al. [12] advise that 1 in 4 workers is dedicated to the PME calculation.

The performance evaluations are done using up to 128 workers. A benchmark with 24119 atoms and a 2:1 PME processor ratio shows that scaling is close to linear up to 38 workers. The dynamic load balancing improves execution time by a factor of 1.5 on these

38 workers. The overhead introduced by repartitioning the domains is 2-5%, and depends on the length of a time step.

3.4.2 Dynamic topology aware load balancing algorithms for molecular dynamics applications [3]. Bhatelé, Kalé, and Kumar [3] discuss the load balancing methods employed in the molecular dynamics code NAMD. NAMD uses two types of partitioning, spacial and force partitioning, to create a fair workload distribution. Spatial domain decomposition is done by splitting the simulation space into cells called patches. The force computation between each patch is assigned to a force object. Both patches and force objects are assigned to the available workers. A worker that is assigned a patch stores the current state of the patch and communicates this to other workers when required. A worker assigned a compute object will perform the force computation between two relevant patches.

To fairly distributes the patches and force objects between workers, two load balancers are employed. First, the *comprehensive load balancer*, used at startup, distributes all patches and compute objects as fair as possible. During the simulation, the *refinement load balancer* moves compute objects from overloaded to underloaded workers. The load of a worker is determined by the number of compute objects assigned to it, Bhatelé, Kalé, and Kumar [3] give no clear method for determining when a worker is over or underloaded. The two load balancers, comprehensive and refinement, use the same heuristic-based scheduling policy for distributing the compute objects. Underloaded workers have priority - meaning that whenever possible compute objects are assigned to an underloaded worker. If multiple or no underloaded workers are present, the compute object is assigned to a worker with favourable communication to the relevant patches. A worker has a favourable communication pattern if the relevant patch is located close-by, or when the patches are already used by one of the other computation objects on the worker.

Running a benchmark of 92,227 atoms on a machine with up to 1024 workers, Bhatelé, Kalé, and Kumar [3] show execution time of namd is reduced by 5.8 times going from 128 to 1024 workers. Benchmarks run by Bowers et al. [5] shows that namd scales well up to 256 workers. These results are further explored in the following section.

NAMD uses empirical data and heuristics to balance the load. The use of compute objects enables the redistribution of workload without redistributing all data associated with a particulate patch. The method explored by Bhatelé, Kalé, and Kumar [3] is not only applicable to molecular dynamics. The use of patches and compute objects abstracts away the actual simulation. Applications where this abstraction is applicable could make use of this work; however, for other applications, different heuristic priorities could result in better performance, e.g., prioritise reducing communication.

3.4.3 Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters [5]. Desmond is a scalable molecular dynamics simulation code proposed by Bowers et al. [5]. To parallelise the simulation, Desmond divides the simulation space into equally sized 3d domains, one for each worker. As with the other molecular dynamics codes, most of the execution time is spend on computing the forces between particles. To determine which worker computes the forces in a specific interaction, the midpoint method is used. In the midpoint method, the worker where the midpoint of an interaction resides computes the forces of the interaction. This is in contrast to a more traditional spacial decomposition, where one of the workers storing the relevant particles is chosen to do this computation. This method uses less communication bandwidth compared to a traditional spatial decomposition approach. Moreover, this method can be seen as a hybrid between spatial and force decomposition.

Bowers et al. [5] claim that this method of static load balancing is sufficient. This is because the distribution of particles in biomolecular systems is approximately uniform. Variations of the midpoint method exist, but Bowers et al. [5] found these to only have a minor performance influence due to the additional overhead.

Bowers et al. [5] show the performance of Desmond by running a simulation with 92.224 atoms, all experiments are performance on a system with 1056 workers. The results show up to 55% parallel efficiency. Based on the parallel efficiency Bowers et al. [5] claim that the proposed static load balancing method provides good-enough load balancing.

To determine the performance of Desmond compared to other molecular dynamics codes, the same 92.224 atom benchmark as before is used for both Desmond and NAMD. Hess et al. [12] added the results for GROMACS, see Figure 4. It is important to note that the setup and machines used for GROMACS are slightly different compared to Desmond and NAMD. These results do not provide a clear comparison of the effectiveness of the different load balancers. We can see that, for this simulation, the scaling of NAMD quickly drops when using 256 workers. Both Desmond and GROMACS seem to scale similarly up to 32 workers. Desmond keeps this scaling up to at least 128 workers; however, because GROMACS is not tested on more then 64 workers it is hard to say anything about further scaling. Based on these results it does seem like, at least for this setup, the static load balancing of Desmond is able to keep up with the dynamic load balancing approaches of NAMD and GROMACS.

3.5 Particle-in-Cell simulations

Particle-in-cell simulations are used to for plasma simulations, high energy physics, cloud modelling and more, for a long time [9]. These simulations are using particles and interactions to model the change in system overtime. However where the performance of n-body simulations is dominated by particle-to-particle interactions, the performance of the the particle-in-cell simulations we discuss is dominated by particle-to-field interactions [10, 19].

The particle-to-field interactions have some interesting properties that make load balancing challenging. Even tough particle-in-cell applications simulate particles, unlike with n-body simulations each particle particle brings with it an equal amount of work. This is because the particle mostly react with the field and not eachother. However, this does not mean that providing each worker with the same amount of particles creates a perfectly balanced load. This imbalance is caused by the communication required to find the correct field point. Evenly dividing the field can cause imbalance if the particles are not uniformly distributed [10].

3.5.1 The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing [10]. Germaschewski et al. [10] discuss the load balancing methods used in the Plasma Simulation

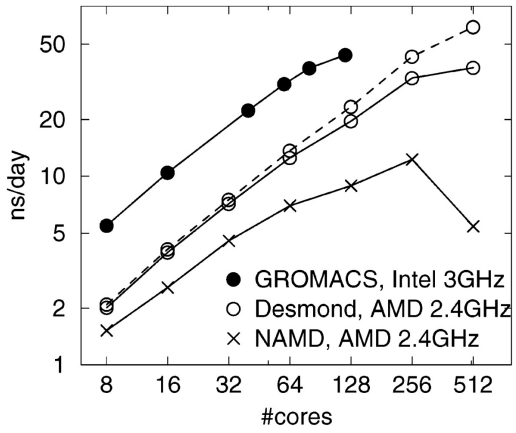

Figure 4: Performance results for NAMD, GROMACS and Desmond. Results are from a 92.224 atom simulation on a dual-core system. The dashed line for Desmond shows performance using a tuned Infiniband library. NAMD and Desmond results come form [5], the GORMACS results and the plot come from [12]. Please note that runs are done with slightly different machines and setups.

Code (PSC). PSC uses spacial domain decomposition with dynamic rebalancing. Spacial domain decomposition is chosen because it offers a significant reduction in communication compared to particle decomposition. The drawback of spacial domain decomposition is that rebalancing is required to account for the migration of particles across domains for input with non-uniform particle distribution.

The load balancer implemented in PSC uses patches, i.e., small pieces of the spatial domain. These patches are distributed across the workers in order to balance the workload. Patches are generated by splitting a space-filling Hillbert-Peano curve into the required number of patches, as seen i Figure 5. Each worker will receive a set of patches that is continual on the space filling curve. By keeping the patches in order, the spacial locality provided by the space filling curve is preserved. After each simulation time step load is rebalanced. In order to rebalance, a definition of load is required. PCS uses the following measure to describe the load per patch, L_p:

$$L_p = N_{particles}(p) + C \times N_{cells}(p)$$

The constant C is used to scale the difference in work per particle or per cell of the field. In the paper $C = 1$ is used; the value was selected based on empirical testing. The load per worker is simply the sum of the load of all patches assigned to that worker. To account for heterogeneous systems, each worker can be given a different amount of load, based on how much more or less computational power is present in that worker.

To determine the performance of patch-based load balancing, Germaschewski et al. [10] compares the results against a static uniform decomposition and a static and dynamic rectilinear decomposition. The rectilinear decomposition decomposes the domain by creating chunks of approximately even load, while the dynamic version updates this distribution every 2000 time steps throughout the simulation. Comparing results on 16 workers, patch-based load balancing performs 1.9× better then the dynamic rectilinear method,

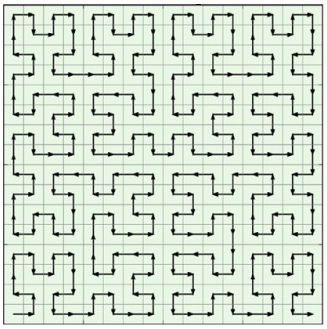

Figure 5: Example of 2d space filling curve, taken from [10]

and 3.64× better than the static approach. Uniform decomposition performs significantly worse than both the other approaches.

The patch-based load balancing proposed by Germaschewski et al. [10] even scales well with a high number of workers. The paper includes results to show that, in weak scaling experiments running with 5120 workers, with 16-core CPUs and 1 GPU each, parallel efficiency was constantly above 90%.

3.5.2 OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations [19]. OhHelp is a domain-decomposed particle-in-cell simulation method proposed by Nakashima et al. [19]. The goal of OhHelp is to use a combination of both particle and spatial decomposition in order to create a scalable method of load balancing particle-in-cell simulations. The simulation space is decomposed into evenly sized subdomains, one for each worker. Because this gives no guarantee of particle balance, every worker, except one, is also given a secondary subdomain. Workers can be given particles from their secondary subdomain in order to help other workers, as seen in Figure 6. When a worker is given particles from its secondary domain it is considered the helper

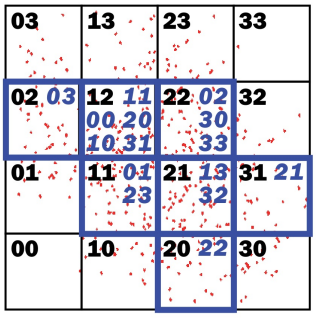

Figure 6: Example of primary (black) and secondary subdomains (blue). Image from [19].

To determine whether the load is balanced, two inequalities are used:

$$P_n \leq \frac{P}{N}(1+\alpha) \tag{1}$$

$$Q_n \leq \frac{P}{N}(1+\alpha) \tag{2}$$

Here, P is the total number of particles, P_n is the number of particles in the primary subdomain of worker n, Q_n is the total number of particles assigned to worker n, N is the total number of workers, and α is a tolerance factor greater then 1.

If Inequality (1) is satisfied for every worker, the system is in primary mode. This means that the workload is balanced with no workers requiring help. If Equation (2) is satisfied, or can be satisfied by shifting particles between helpers and primary workers, the system is in secondary mode. In secondary mode, workload is balanced because workers are helping each other. If Equation (2) cannot be satisfied by redistributing particles between helpers and primary workers, the secondary subdomains are reassigned.

Nakashima et al. [19] show the results of experiments exploring both the strong and weak scaling of OhHelp. Experiments are performed using up to 16 workers; for strong scaling, 2^{27} particles are used, while for weak scaling, the average number of particles in a subdomain is 2^{23}. Finally, each worker has 16 cores, and thus 16 subdomains.

The results show that under weak scaling OhHelp achieves a parallel efficiency of up to 75% when using 16 workers for uniform particle distribution, and 66% for non-uniform particle distribution. Strong scaling shows parallel efficiency of 70% for uniform and 62% for non-uniform particle distribution.

4 DISCUSSION

We have analysed 10 different load balancing solutions across domains. In this section we compare these 10 solutions to find their similarities and differences. This discussion is split in two parts. First, we compare all load balancing solutions by focusing on different features and how each solution implements these features. Next, we list the features we think are missing from the load balancing discussion, and argue why they should be added to the conversation.

4.1 A comparative analysis

In this section we present a feature-driven comparison of the the load balancing solutions we found in the papers in this study. The features are extracted from the selected literature, using both the description and the implementation details that are present in (most of) the papers. Our comparison is synthesised in Table 3, where we emphasise the how each load balancing approach implements each of these features.

4.1.1 Dynamic or static load balancing. As discussed in Section 2 a load balancer can either be static or dynamic. A static load balancer does all the balancing as a preprocessing step, dynamic load balancers can redistribute load during program execution. For some domains it is clear which method is better. In fog computing [25] the jobs can arrive at any moment, balancing the jobs upfront is not possible, this means the load has to be dynamically balanced. The Particle-in-Cell domain papers [10, 19] both use dynamic load balancing to avoid imbalance overtime due to particle movement. In other domains we see do not see a consensus on whether load should be balanced statically or dynamically. In Graph processing [14, 29] and molecular dynamics [12, 3, 5] we see the use of both methods.

In Section 2 we have discussed that dynamic load balancing does rely on some form of static balancing for the initial load distribution. Looking at the summarised papers we see that some dynamic balancer rely more on the static part than others. If we compare [17] to [19] we see this difference with the same domain. The static balancer in [17] has no concept of load balance, it is only designed to create localised clusters. Balancing the load is solely done dynamically, when the scaling factor is introduced. In contrast to this the method described in [19] attempts to balance the load statically, the dynamic part is there to rebalance if imbalance occurs over time.

Two main reasons for choosing dynamic load balancing are mentioned in the discussed papers. Firstly if the load is hard to model based on data distributing. The authors of [14] state that dynamic load balancing enables load balancing for all kinds of graphs and algorithms, without doing prior computation to determine the problem characteristics. The second reason is changes in load over time, both particle-in-cell papers [10, 19] use dynamic load balancing for this reason. Static load balancing [29, 5] use static load balancing because of the reduced complexity and performance overhead it provides compared to dynamic load balancing.

Although the type of application and data is an important factor in choosing between static and dynamic load balancing, it is clearly not the only one. Implementation complexity and performance overhead also have to be taken into account.

4.1.2 Load metric. To balance the load in a system, load needs to be defined first. The data that is used to determine the load per worker we will call the load metric. For example, in [10, 19] load is model based on the number of particles and field points assigned to each worker. The load balancer in [12] uses response time for each computation step as the load per worker. In these examples both, the number of particles and field points and response time are a load metric.

On first look the different load metrics seem to be very problem specific. However we see a clear distinction between two different types of load metrics. The first method is to model the load per worker based on way data or work is distributed, e.g., in [14] load expressed based on the number of vertices and edges assigned to each worker. The second method is based on runtime measurements, e.g., the method in [17] uses deviation from mean time per time step as load per worker. In Table 4 we classified each load metric as either based on data distribution or runtime measurements. Please note that methods that define load based on work, such as [3], are classified as using data distribution.

The first thing to notice from Table 4 is that both static load balancers [29, 5] use a load metric based on data distribution. This is because static load balancer are not able to use runtime measurements, because balancing is done as a prepossessing step. A dynamic load balancer can use either type of load metric, and we see that both are used. Just as with static and dynamic load balancing applications in the same domain are not always in agreement regarding which type of load metric to use, e.g., both N-body simulations use a different type of load metric [17, 20]. We do see that the chosen load metric is very dependent on what kind of data is used. Due to the nature of Both particle-in-cell applications [10, 19] load can be modelled accurately using the number of particles

Table 3: Features and their implementation by each paper. The listed features are, F1: (S)tatic/(D)ynamic, F2: Load metric, F3: Balance Unit, F4: Heterogeneous Support, and F5: (P)roactive/(R)eactive

Domain	Ref	F1	F2	F3	F4	F5
Fog Computing	[25]	D	Worker Utilisation	Job	x	P
Graph Processing	[14]	D	Runtime measurements	Vertices	x	R
	[29]	S	# Vertices & Edges	Vertices & Edges		P
N-body Simulations	[17]	D	Response time	Particles	x	R
	[20]	D	# Interactions	Interaction groups		P*
Molecular Dynamics	[12]	D	Response time	Charge groups	x	R
	[3]	D	Response time	Interactions	x	R
	[5]	S	Simulation space volume	Particles		P
Particle-in-Cell	[10]	D	# Particles & grid points	Patches	x	P*
	[19]	D	# Particles & grid points	Particles & grid		P*

Table 4: Load metrics categorised as either based on (D)ata distribution or measurements done at (R)untime .

Domain	Ref	Load Metric	D	R
Fog Computing	[25]	Worker Utilisation	x	
Graph processing	[14]	Runtime measurements		x
	[29]	# Vertices & Edges	x	
N-body simulations	[17]	Response time		x
	[20]	# Interactions	x	
Molecular Dynamics	[12]	Response time		x
	[3]	Response time		x
	[5]	# Particles	x	
Particle-in-Cell	[10]	# Particles & grid points	x	
	[19]	# Particles & grid points	x	

and per worker. The reason for both these applications to use a dynamic scheduler is to address the growing imbalance over time. In [14] the goal of the application is to be able to solve graph processing problems without any prior knowledge of the problem. To better handle unknown problems a load metric based on runtime measurements is used.

It is clear that each for each application or domain the detail of the load metric will differ. However even tough the details vary the there are clear distinctions between the two types of load balancing.

4.1.3 Balance unit. With the load metric a load balancer can estimate the load per worker. If it turns out that the load is not balanced the load balancer can solve this by (re)distribute load. The bit of work that is distributed between workers we call the balancing unit. For example, in [17] clusters are resized and particles are moved between workers to reflect this, a particle is the balance unit. The methods described in [3] moves compute objects between workers to achieve load balance, the compute objects are the balance units. In Table 3 we can see that if the load metric is based on data distribution, the balance unit is based on the same data distribution. For example, in both methods using static load balancing [29, 5] the load metric and the balance unit are the same. In [20] and [10] the balance unit and load metric do not seem to match on a first glance; however, both interaction groups and patches are groups of the data that is used as a load metric.

Just as with the load metrics, the different balance units seem to be specialised for domains or applications. However, just as with the load metrics we can classify the different balance units showing that conceptual we only encountered a small set of balance units. The classification we use is based on two characteristics of the balance unit. Firstly, is the balance unit based on moving data or is the balance unit based on moving work. A balance unit based on data moves load because is work associated with the data, e.g., the particles in [17]. If the balance unit is based on work the work is moved around without moving the data that it is connected to, e.g., the charge groups in [3]. The second classification is whether the balance unit consists of a single unit of work or data, e.g., a single particle or interaction, or if the data or work is grouped together, e.g., patches or interaction groups. In Table 5 we use these two characteristics to classify each balance unit.

Table 5: Classification of balance units. (D)ata, (W)ork, (U)nit and (G)roup

Domain	Ref	Balance Unit	D	W	U	G
Fog Computing	[25]	Job			x	
Graph processing	[14]	Vertices	x		x	
	[29]	Vertices & Edges	x		x	
N-body simulations	[17]	Particles	x		x	
	[20]	Interactions		x		x
Molecular Dynamics	[12]	Charge groups	x			x
	[3]	Interactions		x	x	
	[5]	Particles	x		x	
Particle-in-Cell	[10]	Patches	x			x
	[19]	Particles & grid	x		x	

In Table 5 we see that most balance units are based on single units of data.

4.1.4 Heterogeneous systems support. As mentioned in Section 2, distributed systems are not necessarily comprised of homogeneous workers. In these heterogeneous systems a fair workload distribution does not necessarily equal one where every worker is given the

same amount of work. In [25] the platform is comprised of different workers where there is no guarantee of heterogeneity. The heterogeneity is addressed using worker utilisation as a load metric. This load metric makes no assumptions about how much work a worker can handle, instead it is measured at runtime. The heterogeneous support comes form the use of runtime measurements as a load metric. The other load balancers that use a runtime measurement based load metric [14, 17, 12, 3] also get this heterogeneous support for 'free'. The support is there because the load balancers look at the difference in performance of all workers, and don't assume homogeneity.

When a load balancer uses a data distribution based load metric, heterogeneous support needs to added explicitly. In [10] this is done by giving every worker a relative performance number. If the performance number of one worker is twice as high as another it gets twice as much load. In contrast to using runtime measurements, this method requires upfront knowledge of the platform and information on the relative performance of all workers. Non of the other papers that use data distribution as a load metric have heterogeneous support. And [25] and [10] are the only paper that make an explicit mention of heterogeneous support.

4.1.5 Proactive / Reactive. The load in a system can be balanced proactive or reactive, i.e., balanced before of after the load is imbalanced. There is a type of load balancer where we can clearly see it is proactive, static load balancing. Static load balancing is always proactive, because static balancing is done before execution. We can also see that most dynamic load balancer that use a runtime measurements based load metric are reactive. This is because most of these load balancer detect load balance through runtime measurements and than act on it. The single exception is [25]. This is because the workload of fog computing does not create imbalance overtime, and the arriving jobs can be schedule such that the load remains balanced. Classifying the rest of the methods, i.e., the dynamic load balancer that use a data distributing based load metric, becomes a little more complex. Technically these load balancer look a the data distribution and if they detect imbalance proceed to redistribute the load. This implies that these systems are reactive, because there is imbalance before the load is rebalanced. However if we look at the remaining applications we see that all use some form of bulk parallel synchronisation model, i.e., there is a compute step followed by a synchronisation step. During these computation steps data can move around and create imbalance. This imbalance will not affect performance until the next computation step; however, during the synchronisation step the load balancer kicks in and rebalances load. This means that imbalance is present, but it is addressed before it affects performance. We classify these types of load balancers as performance proactive.

Theoretical it is possible to be proactive trough observation of the changing load; however, non of the featured paper use any such method. The performance proactive system do however achieve the same results, i.e., balance load before there is a performance impact. This statement does assume that all load balancer are able to perfectly balance the load, which is not always true. We fount that non of the featured papers have any mention of proactive versus reactive scheduling. This indicates that this is not a conscious design decision but that it is determined by other features.

4.2 Missing Features

The features from the previous section are all well represented in most or all of the featured literate. However there are some features that we find are missing from or underrepresented in the literature. In this section each of these missing features is explored. We discuss what these features entail, where in the literature they might be referenced and why they are important for improving the discussion around load balancing.

4.2.1 Elasticity. Most authors assume to have access to a stable distributed system where they have full to there assigned workers. This assumption might not always be true. A growing number of distributed platforms are becoming dynamic, e.g., cloud computing platforms. These dynamic platforms can redistribute resources as demand changes [4]. To accommodate changes in the available resources a load balancer needs to be elastic. Elasticity refers to the ability of the load balancer to react to these changes in available resources. The load balancer proposed in [25] has some form of elasticity, it can update the available workers with every new job. However in the featured literature there is no real discussion of redistribution work on a changing platform.

4.2.2 Heterogeneous system architecture support. We discussed the importance of heterogeneous support in load balancing. This discussion focused solely on different workers providing different amounts of computational power; however, platforms can also provide heterogeneity through different compute architectures. A good example is workers with GPU. Some tasks in an application might be better suited for a GPU worker, and a load balancer should accommodate this. The two reference to load balancing for heterogeneous architectures in the literature are, (1) [10] through relative performance values and (2) [25] by using a tag system. The relative performance numbers used in [10] enable the load balancer to utilise GPUs. However no real distinction is made between the workers apart from this relative performance. In [25] a tag system is proposed, this is used to tag the security level of each worker. These tags are used to schedule sensitive jobs on more secure workers. Although this system is not used for distinguishing worker architecture necessarily, it shows that load balancing over different types of workers is possible. Not considering how to fully utilise heterogeneous systems can result in underutilised the available hardware.

4.2.3 Load balancing specific performance measurements. During the literature comparison we have not looked much at the performance of each load balancer. The main reason for this is that the difference in domains, applications, and experiments make a one-to-one comparison non-trivial. Most importantly, many papers do not directly show the performance of there load balancers. In the featured literature we do see performance expressed through, comparing different load balancing methods [25, 10], load balancing overhead [12], or load balancing performance impact [12]. However most of the featured papers focus on how close their parallel scaling is the optimal scaling. It is clear why authors would choose to advertise scaling, as the goal of implementing a load balancer is often to create a well scaling application. Improving scaling is often one of the goals of load balancing; however, own its own it is not a good performance indicator for a load balancer.

An application with a very high communication overhead will scale poorly even if the load is balanced perfectly. And when we move to a heterogeneous platform optimal scaling becomes harder to define, providing an even greater source of inaccuracy. This all means that using scaling as the only performance indicator can not accurately capture the performance of a load balancer. Based on the characteristics of different load balancers we think that a comprehensive performance evaluation of load balancing should contain at least a measure of added overhead, as well as some accurate representation of the load per worker over time.

5 CONCLUSION

In our preliminary research we found that there is little literature available that compares load balancing methods. This study aims to address this by providing an insight into the similarities and difference between load balancing solution in different applications and domains. On first glance it seems like the different load balancing solutions are very specialised, only applicable to their indented applications. The specialised nature of load balancing solutions seems extra highlighted by the fact that even within the same domain there is often no consensus on how to implement a feature. However this first impression is deceiving.

We have shown that abstracting away implementation details reveal many similarities between load balancing solutions. The features which on first glance seem to be specialised can in fact be categorised to reveal common concepts behind the details. Not only that but we find that there are correlations between the use of the different features. This means that we can conclude that certain feature combinations can't exist, e.g., runtime measurement based load metric in a static load balancing. Besides ruling out combinations we see certain feature combinations being used more for different types of applications. These observations are found across the featured domains and show that even very specialised load balancers share similar concepts. The problem is that these concepts are often hidden behind details or domain specific terminology. By looking at the load balancers in this more generalised way we can see that many parts of the solutions are not necessarily bound to a specific application. What this means is that it can be very beneficial to look outside of your own field for load balancing inspiration. Based on the work we have done here we believe that it is important for the continuous development of the load balancing field to stimulate these inter-domain load balancing discussions.

5.1 Future work

This work focused on literature that directly references the term *load balancing*. There are however practices which are not called load balancing but are performing the same task, e.g., scheduling [22], workload partitioning [24] and mapping [28]. To develop a complete picture of load balancing in different domains these kinds of methods should be collected and analysed along side eachother.

We have shown that many similarities between load balancing solutions are either hidden behind domain specific terminology or are hard to find because implementation details are not be named explicitly. This creates a barrier for someone who want so learn from load balancing solutions outside of their own domain. In this study we already made an attempt at generalising implantation details for some features. These generalisations where very effective in highlighting differences and similarities between solutions. To encourage inter-domain load balancing discussion this we need a common language with which load balancing concepts and ideas can be communicated.

Creating this common language will be useful for comparing load balancing solutions. However to truly compare different solution will require a common method for performance gathering and reporting. This work should include a comprehensive set of metrics that are shown to fully encompass the performance characteristics of load balancer. Aswell as a method for testing and comparing different load balancing designs and implementations. The performance metrics will give improve the easy with which we can compare load balancing solutions. Coupling this with the framework or platform that can make it easy to implement and test load balancing can greatly improve load balancing comparison and discussion.

REFERENCES

[1] Ali M. Alakeel. "A Guide to dynamic Load balancing in Distributed Computer Systems". In: *International Journal of Computer Science and Network Security (IJCSNS* (2010), pp. 153–160.

[2] Josh Barnes and Piet Hut. "A hierarchical O(N log N) force-calculation algorithm". In: *Nature* 324.6096 (1986), pp. 446–449. ISSN: 1476-4687. DOI: 10.1038/324446a0.

[3] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. "Dynamic topology aware load balancing algorithms for molecular dynamics applications". In: *Proceedings of the 23rd international conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: Association for Computing Machinery, June 8, 2009, pp. 110–116. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542295.

[4] Flavio Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. MCC '12. New York, NY, USA: Association for Computing Machinery, Aug. 17, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[5] Kevin J. Bowers et al. "Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters". In: *ACM/IEEE SC 2006 Conference (SC'06)*. SC 2006 Proceedings Supercomputing 2006. Tampa, FL: IEEE, Nov. 2006, pp. 43–43. DOI: 10.1109/SC.2006.54.

[6] T.L. Casavant and J.G. Kuhl. "A taxonomy of scheduling in general-purpose distributed computing systems". In: *IEEE Transactions on Software Engineering* 14.2 (Feb. 1988), pp. 141–154. ISSN: 00985589. DOI: 10.1109/32.4634.

[7] Rishan Chen et al. "Improving large graph processing on partitioned graphs in the cloud". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2012, pp. 1–13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391232.

[8] Biagio Cosenza et al. "Distributed Load Balancing for Parallel Agent-Based Simulations". In: *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing. ISSN: 2377-5750. Feb. 2011, pp. 62–69. DOI: 10.1109/PDP.2011.22.

[9] John M. Dawson. "Particle simulation of plasmas". In: *Reviews of Modern Physics* 55.2 (Apr. 1, 1983). Publisher: American Physical Society, pp. 403–447. DOI: 10.1103/RevModPhys.55.403.

[10] Kai Germaschewski et al. "The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing". In: *Journal of Computational Physics* 318 (Aug. 1, 2016), pp. 305–326. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2016.05.013.

[11] Minyang Han and Khuzaima Daudjee. "Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems". In: *Proceedings of the VLDB Endowment* 8.9 (May 1, 2015), pp. 950–961. ISSN: 2150-8097. DOI: 10.14778/2777598.2777604.

[12] Berk Hess et al. "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation". In: *Journal of Chemical Theory and Computation* 4.3 (Mar. 2008), pp. 435–447. ISSN: 1549-9618, 1549-9626. DOI: 10.1021/ct700301q.

[13] Pritish Jetley et al. "Massively parallel cosmological simulations with ChaNGa". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008 IEEE International Symposium on Parallel and Distributed Processing. ISSN: 1530-2075. Apr. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536319.

[14] Zuhair Khayyat et al. "Mizan: a system for dynamic load balancing in large-scale graph processing". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: Association for Computing Machinery, Apr. 15, 2013, pp. 169–182. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465369.

[15] Mingwei Li et al. "Distributed machine learning load balancing strategy in cloud computing services". In: *Wireless Networks* (July 6, 2019). ISSN: 1022-0038, 1572-8196. DOI: 10.1007/s11276-019-02042-2.

[16] Andrew Lumsdaine et al. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17.1 (Mar. 1, 2007). Publisher: World Scientific Publishing Co., pp. 5–20. ISSN: 0129-6264. DOI: 10.1142/S0129626407002843.

[17] Youssef M. Marzouk and Ahmed F. Ghoniem. "K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical N-body simulations". In: *Journal of Computational Physics* 207.2 (Aug. 10, 2005), pp. 493–528. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.01.021.

[18] Henning Meyerhenke. "Dynamic Load Balancing for Parallel Numerical Simulations Based on Repartitioning with Disturbed Diffusion". In: *2009 15th International Conference on Parallel and Distributed Systems*. 2009 15th International Conference on Parallel and Distributed Systems. ISSN: 1521-9097. Dec. 2009, pp. 150–157. DOI: 10.1109/ICPADS.2009.114.

[19] Hiroshi Nakashima et al. "OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations". In: *Proceedings of the 23rd international conference on Supercomputing*. ICS '09. New York, NY, USA: Association for Computing Machinery, June 8, 2009, pp. 90–99. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542293.

[20] Olga Pearce et al. "Load balancing n-body simulations with highly non-uniform density". In: *Proceedings of the 28th ACM international conference on Supercomputing*. ICS '14. New York, NY, USA: Association for Computing Machinery, June 10, 2014, pp. 113–122. ISBN: 978-1-4503-2642-1. DOI: 10.1145/2597652.2597659.

[21] Behrouz Pourghebleh and Vahideh Hayyolalam. "A comprehensive and systematic review of the load balancing mechanisms in the Internet of Things". In: *Cluster Computing* 23.2 (June 2020), pp. 641–661. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-019-02950-0.

[22] K Ramamritham and J A Stankovic. "Dynamic Task Scheduling in Hard Real-Time Distributed systems". In: *IEEE Software* 1.3 (1984).

[23] S. Seo et al. "HAMA: An Efficient Matrix Computation with the MapReduce Framework". In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. 2010 IEEE Second International Conference on Cloud Computing Technology and Science. Nov. 2010, pp. 721–726. DOI: 10.1109/CloudCom.2010.17.

[24] J. Shen et al. "Workload Partitioning for Accelerating Applications on Heterogeneous Platforms". In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (Sept. 2016). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 2766–2780. ISSN: 1558-2183. DOI: 10.1109/TPDS.2015.2509972.

[25] Anil Singh and Nitin Auluck. "Load balancing aware scheduling algorithms for fog networks". In: *Software: Practice and Experience* n/a (n/a). ISSN: 1097-024X. DOI: 10.1002/spe.2722.

[26] Volker Springel. "The cosmological simulation code gadget-2". en. In: *Monthly Notices of the Royal Astronomical Society* 364.4 (Dec. 2005). Publisher: Oxford Academic, pp. 1105–1134. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2005.09655.x.

[27] Yong Meng Teo and Rassul Ayani. "Comparison of Load Balancing Strategies on Cluster-based Web Servers". In: *SIMULATION* 77.5 (Nov. 1, 2001). Publisher: SAGE Publications Ltd STM, pp. 185–195. ISSN: 0037-5497. DOI: 10.1177/003754970107700504.

[28] L. Thiele et al. "Mapping Applications to Tiled Multiprocessor Embedded Systems". In: *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*. Seventh International Conference on Application of Concurrency to System Design (ACSD 2007). ISSN: 1550-4808. July 2007, pp. 29–40. DOI: 10.1109/ACSD.2007.53.

[29] Xiaowei Zhu et al. "Gemini: A Computation-Centric Distributed Graph Processing System". In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016, pp. 301–316. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu (visited on 07/20/2020).