pMACH : Power and Migration Aware Container scHeduling

Sourav Panda, K. K. Ramakrishnan, Laxmi N. Bhuyan
University of California, Riverside, CA

***Abstract*—Data center workload fluctuations need periodic, but careful scheduling to minimize power consumption while meeting the task completion time requirements. Existing data center scheduling systems tightly pack containers to save power. However, with the growth of multi-tiered applications, there is a significant need to account for the affinity between application components, to minimize communication overheads and latency. Centralized container scheduling systems using graph partitioning algorithms cause a significant number of task migrations, with associated downtime.**

We design pMACH, a novel distributed container scheduling scheme for optimizing both power and task completion time in data centers. It minimizes task migrations and packs frequently communicating containers together without overloading servers. pMACH operates at peak energy efficiency, thus reducing energy consumption while also providing greater headroom for unpredictable workload spikes. We also propose in-network monitoring using smartNICs (sNIC) to measure the communications and then perform scheduling in a hierarchical, parallelized framework to achieve high performance and scalability. pMACH is based on incremental partitioning and it leverages the previous scheduling decision to significantly reduce the number of containers moved between servers, avoiding application downtime.

Both testbed measurements and large-scale trace-driven simulations show that pMACH saves at least 13.44% more power compared to previous scheduling systems. It speeds task completion, reducing the 95th percentile by a factor of 1.76-2.11 compared to existing container scheduling schemes. Compared to other static graph-based approaches, our incremental partitioning technique reduces migrations per epoch by 82%.

***Index Terms*—Data center scheduling, power saving, cost saving, meeting SLA**

I. Introduction

Striking the right balance between conflicting scheduling requirements such as overprovisioning to satisfy an application's service level agreements (SLA) vs. tightly packing servers to save power in a data center (DC) can be challenging. Tightly packing containers is necessary to achieve high server utilization and power saving [1]–[4] by turning off idle servers. In general, DCs operate at $\sim 20\%$ server utilization [5]–[7] and 10% network utilization [2], [8] in order to meet application SLAs. However, this results in high overall DC power consumption as more servers remain powered on.

While there exists some prior work to minimize both power and task completion time [9], they are not incremental, leading to a significant number of container migrations. They ignore the cost of container migrations when adapting to workload changes or when the workload is consolidated to a smaller number of servers to reduce power consumption. Container migration (e.g., CRIU [10]) also results in downtime [11], and frequent migrations can adversely impact task completion times and are likely to result in SLA violations [12]. Thus, it is desirable to have a DC scheduler that simultaneously reduces power, task completion time, and container migrations and is also scalable to DC scales. The challenges are several - the need to operate servers efficiently [13], support fluctuating workloads [8], account for application container affinity [14], and account for migration overheads [11].

Today's DCs typically employ some form of heuristic-driven bin packing such as RC-Informed [15], Borg [16], pMapper [17] and others [18]–[20]. These solutions do not consider container affinity, potentially resulting in hosted cloud applications having higher latency [9] due to large inter-container communications. State-of-the-art task placement frameworks such as Borg [16] and RC-Informed [15] pack containers in highly utilized servers. Borg aims to reduce stranded resources while RC-Informed over-subscribes CPU resources at 125% [15], as a way of minimizing the number of servers deployed. To minimize power consumption, pMapper [17] determines the target utilization for each server based on the power model for the server. It then places VMs on servers using a bin-packing algorithm, trying to meet the target utilization on each server. E-PVM [21] places containers on the server with the lowest utilization, so as to leave large headroom for load spikes and achieve low task completion time.

Goldilocks [9] is another approach for scheduling latency sensitive tasks in a DC. It balances task completion time and energy, benefiting from placing frequently communicating containers together. However, it uses a centralized, periodic graph partitioning and scheduling policy using Metis [22], which does not scale to large DCs consisting of tens of thousands of servers. The change in container graph going from one epoch to the next may be incremental, but re-partitioning the entire graph, as in [9], results in a lot of container migrations. Vertices can be moved from one partition to another due to repartitioning. As vertex migrations correspond to container migrations, they are expensive and must be minimized. Furthermore, their work does not consider the overhead associated with transmitting the traffic matrix.

A DC cluster of several thousand servers, switches and links is typically broken up into smaller identical units. These units are called pods, comprising of several hundred servers along with the top-of-the-rack and aggregation switches. The DC network provides high-performance connectivity between all pods in the DC. We propose pMACH a Two-Tier distributed scheduling framework to adaptively 'right size' the DC by first considering a pod-level partitioning of containers, and then repartitioning the container sub-graph within a pod. pMACH schedules groups of containers (pMACH is generic, and may be used for scheduling VMs as well) of a partition on a server. It minimizes container migrations by adopting an incremental partitioning technique. pMACH's main focus is on achieving scalability using a Two-Tier partitioning algorithm, and executing the algorithm in an entirely distributed manner, unlike the centralized approach that has been the state-of-the-art. pMACH's strengths are:

- **Scalability:** pMACH can schedule a large number of containers over a cluster of ten thousands of servers in a relatively short time.
- **Multi-objective optimization:** pMACH balances between power consumption, task completion time, and task migrations.
- **Efficient:** pMACH only requires a small amount of processing resources (few cores on a select server in each pod of the DC), and uses network offload to relieve CPU cores of scheduler related activity.
- **Practical:** rather than assuming the container communication graph, pMACH collects the needed information in real-time on a Smart network interface card (sNIC).

pMACH significantly reduces task completion time as containers that frequently communicate with each other are placed together in the DC topology. Power saving is achieved by having a minimal number of servers, so that unused servers can be turned off. Container migrations are reduced by accounting for dirty vertices (vertices that are moved from their original group to another group in the graph), thereby minimizing downtime. We consider three mechanisms to perform hierarchical partitioning of the container graph, namely, ParMetis Base partitioning, ParMetis Adaptive partitioning [23], and Tabu Search. Both ParMetis offerings (e.g. Base and Adaptive) are highly parallelized. The difference between them is that Adaptive partitioning reduces container migrations and is faster to deal with workload variation. Tabu Search is a widely used meta-heuristic for graph partitioning as shown in [24]–[26] and allows us to provide a multi-objective cost formulation, accounting for container migration costs. Tabu Search however has poor scaling properties for larger graphs. Hence, we propose a hierarchical Two-Tier partitioning architecture that combines the advantages of both ParMetis Adaptive partitioning and Tabu search.

To obtain the container graph, we use a sNIC to collect the communication graph and provide it to the appropriate ParMetis graph partitioning worker nodes. This helps us save crucial CPU cycles. We use an efficient data stream summarization [27] to derive the edge weights with reasonable accuracy to allow frequently communicating container pairs to be placed together, to minimize task completion time.

Both testbed measurements and large-scale trace-driven simulations show that pMACH saves 13.44% more power compared to other scheduling systems. It speeds task completion, reducing the 95th percentile by a factor of 1.76-2.11 compared to existing container scheduling schemes. Compared to the static graph-based approach [9], our incremental partitioning technique reduces the migrations per epoch by 82%. Our major contributions include:

- A distributed scheduling system to scalably schedule containers across tens of thousands of servers.
- A Two-Tier scheduler composed of ParMetis Adaptive partitioning and Tabu Search to help reduce the partitioning time and container migrations.
- An efficient telemetry data structure on the sNIC to obtain the container graph in real-time.
- We implemented pMACH in a DC testbed (Cloudlab [28]) using 16 servers. We also implemented a large-scale flow-level simulation to demonstrate the scalability of pMACH.

II. Background and Related Work

Problem Statement: A DC scheduler runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines [16]. This work focuses on scheduling for light-weight container instances. The broad goals for a power and migration aware DC scheduling approach are:

- **Task Completion Time:** There is a need to honor container resource requirements and place frequently communicating container pairs together so that the task completion time (latency) is reduced.
- **Power Consumption:** It is desirable to consolidate containers to fewer servers and operate them at peak energy efficiency.
- **Downtime:** It is important to minimize the downtime impact of container migrations.

Related Work: E-PVM and RC-Informed are instances of the vector bin packing that model static resource allocation problems, where there is a set of servers with known capacities and a set of services with known demands [19]. Firstly, E-PVM distributes containers to the least occupied servers, leaving sufficient head room for spikes, but resulting in undesirably higher power consumption [21]. Alternatively, RC-Informed [15] predicts the workload for the scheduler to safely oversubscribe resources and tightly pack containers, thereby consuming less energy compared to E-PVM [9]. The problem with E-PVM and RC-Informed is that they do not consider container pair affinities and nor do they take advantage of peak energy efficiency. Compared to E-PVM, 6.6% to 18.8% power can be saved by alternatives that pack containers more tightly [9]. Furthermore, workload prediction can be imperfect and RC-Informed is shown to predict a new VM's CPU utilization with only 81% accuracy [15]. Under-prediction will cause the target peak utilization to be exceeded, and with oversubscribing of resources at 125%, it can result in violating latency requirements. Thus, it is desirable to have a lower utilization level for each processor and still save energy.

Another approach is to represent containers and the communication between them as a graph and use partitioning to allocate containers to different nodes [9]. The approach considers a container graph with resource demands as vertex weights and inter-container communication as edge weights. By running the graph partitioning algorithm accounting for edge cut and partition aggregate utilization, containers with high communications are grouped together and the load of the container group gets balanced. Goldilocks [9] is based on periodic partitioning of the container graph by Metis [22] and mapping it to DC resources. According to the formulation in [9], Metis K-Way partitioning places frequently communicating containers together by minimizing edge cut (e.g., communication between servers). It also respects server capacities by balancing container resource demands across servers.

Tabu Search is a widely used meta-heuristic for graph partitioning as shown in [24]–[26] and allows us to provide a custom cost formulation that can account for the cost of migrations. Local search methods have the tendency to be stuck in suboptimal regions. Tabu Search enhances the performance of these techniques by prohibiting already visited solutions or others through user-provided rules [29]. As shown in the next section, Tabu search reduces the number of migrations considerably, which is an important criterion.

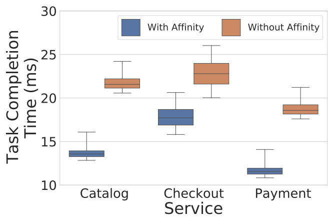
Outstanding Challenges: The shortfall of Goldilocks is that the partitioning at every epoch is not incremental, causing a lot of container migrations, and it is not parallelizable, making it slow. In reality, there are only small changes in the workload between epochs. Incremental partitioning [23] reduces vertex migrations while reducing the edge cut and load imbalance. ParMetis is a Message Passing Interface (MPI [30]) based graph partitioning technique that distributes the graph's vertices across processing cores, to reduce the partitioning time. Instead of the centralized partitioning and scheduling in Goldilocks, we envision a distributed architecture to do both functions. Thanks to advances made in the graph partitioning algorithms, edge-cut minimization and load balance can also be carried out in parallel (e.g., multi-core or multi-server) by using ParMetis [23]. Alternatively, we could use Tabu Search also, but it is not scalable. The next section shows how our Two-Tier approach combines the benefits of Adaptive partitioning and Tabu search for a scalable design.

III. Motivating Experiments

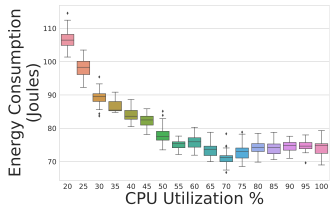
In this section, we carry out several experiments to measure the impact of container affinity, energy consumption and migrations on the performance of a DC. We also test how all these factors can be improved through graph partitioning.

A. Metrics

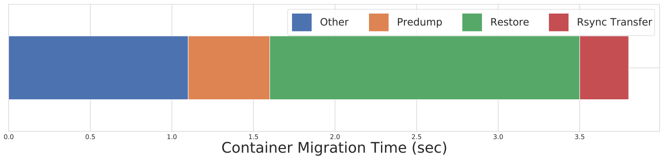
Container Affinity: Our previous work [9] shows that it can achieve 2.6 times better task completion time compared to alternatives such as E-PVM, RC-Informed, and p-Mapper by grouping frequently communicating containers together. To understand this in our context, we utilize a 10-tier Kubernetes microservice application provided in [31] on a testbed with four servers connected by an intermediate switch. First we let the Kubernetes scheduler decide the container placement by itself and in the second scenario we place the high affinity container pairs together (e.g. CheckoutService with



((a)) Affinity vs Response Time. The wicks represent 5th and 95th%tile.



((b)) Peak Energy Efficiency at 65%-75% CPU utilization



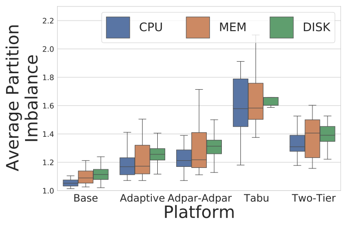
((c)) Migration Time Breakdown

Fig. 1: Modern DC Applications and Affinity

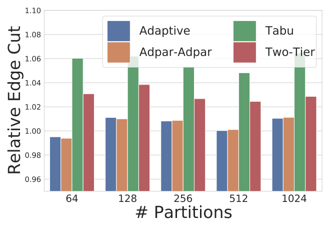
PaymentService) by setting the nodeName [32] configuration. In Figure 1(a) we show that by exploiting affinities across three services (Catalog, Checkout, and Payment) the 95th percentile response time sees a speedup of 1.5, 1.2, and 1.5 times, respectively. This is a different workload compared to [9], resulting in a lower speedup. Clearly, container affinity must be considered while placing the containers, unlike other bin packing approaches such as E-PVM and RC-Informed.

Energy Consumption: Figure 1(b) shows a boxplot of the energy consumption in Joules, measured using RAPL [33] with respect to the load on the CPU. In this experiment, we use a 16 core Intel(R) Xeon(R) CPU D-1548 2.00GHz x86_64 architecture CloudLab [28] instance. We toggle the CPU utilization on all CPU cores of the instance (x-axis in Figure 1(b)) and study the total energy consumed by the CPU package (y-axis in Figure 1(b)) to complete a buffer I/O workload [34]. We observe that the energy consumed is lowest around 65% to 75% CPU utilization, displaying a 'U' curve for energy consumption. Similar observations have been made in the past [9], [35], generally referring to it as Peak Energy Efficiency, which is defined as the point achieving the maximum number of operations completed per watt. Such a strategy saves more total server power, while leaving a larger headroom to deal with instantaneous load fluctuations. The non-linear relationship between CPU load and power curve may be attributed to the cubic reduction in processing power with a linear reduction in performance for DVFS and dynamic overclocking, such as with Intel's TurboBoost [35]–[38].

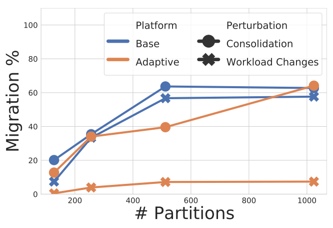
Migrations: Consolidation can contribute to considerable power savings by turning off both servers and network switches and links. Maintaining affinity among containers is important to reduce communication overhead and reduce task completion time. The byproduct of consolidation and enforcing affinity are migrations. Containers will have to be moved at scheduling epochs, resulting in container migration overheads (both additional processing and communication) and undesirable downtime. Figure 1(c) shows that using CRIU [10] a Memcached container instance from CloudSuite [39] takes upwards of 3.5 seconds to migrate. Furthermore, the image predump, image transfer using rsync, and image restore require stopping application execution resulting in application downtime. Overall, it is important to minimize migrations, which was not considered earlier (*e.g.,* [9]). The typical sources of migrations delays are 1) checkpoint/restore 2) writes to remote storage, and 3) scheduling delays [40].



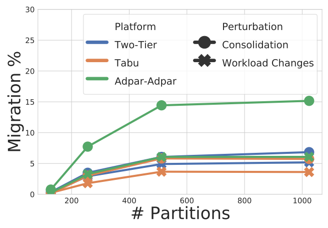
((a)) Partition Balance



((b)) Relative Edge Cut



((c)) ParMetis Migrations



((d)) Migrations (non-ParMetis)

Fig. 2: Partition goodness for different approaches.

B. *Effective Graph Partitioning*

Since we transform the container placement problem to a graph partitioning algorithm, we must consider how effective the partitioning is. ParMetis distributes vertices among cores (e.g. MPI workers) to parallelize the algorithm while balancing the load on the MPI workers. The technique can operate in two modes, namely, COUPLED or UNCOUPLED. These two approaches vary in partitioning time. In the COUPLED approach all vertices that belong to the same original partition are placed within the same CPU core before the next partitioning phase starts. The advantage of the COUPLED approach is that partitioning time is much lower because of increased local computation and reduced communication between the cores [41]. One constraint while running in the COUPLED mode is that the number of MPI workers must equal the original and target number of partitions. As we explore the partitioning time in this section by varying the number of partitions (e.g. 1024), we cannot physically have those many MPI workers, forcing us to operate in the UNCOUPLED mode. Later in the paper, we explain how we can scalably respect this constraint imposed by the COUPLED approach and leverage its speedup.

Graph partitioning imposes significant processing requirements with large graphs. Even though it is possible to have a 256-node MPI cluster [42] that can quickly partition such graphs, it would be impractical and expensive to have such a dedicated cluster in a DC. We perform the **graph partitioning in a distributed manner** in the DC by carefully designating CPU cores in selected servers in each pod of the DC. Taking advantage of the high-bandwidth links in the DC, and intelligently splitting it into a hierarchical solution, we are able to partition the large graph rapidly. Figure 2 shows the clustering goodness for various platforms such as ParMetis base partitioning (**Base**), ParMetis Adaptive partitioning (**Adaptive**), Tabu Search, and a combination of these.

To manage the scale of the problem, we envisage partitioning the graph first at a coarser granularity (Tier-1). Then, we partition each subgraph (Tier-2). Adpar-Adpar refers to the partitioning result, where the first tier of partitioning employs the ParMetis Adaptive partitioning, generating X partitions. The second tier partitioning then internally partitions the graph, using ParMetis Adaptive partitioning, into $\frac{Target\ \#Partitions}{X}$ partitions. We also define Two-Tier, where the first tier carried out by ParMetis Adaptive partitioning and the second tier is carried out using Tabu Search. We carry out several experiments to compare these partitioning techniques. The partitioning was done using a single CPU core.

Container Graph: We utilize a trace derived from the CDF of DC traffic, using a NS3-based DC simulation [43]. We obtain the communication matrix time series for different workloads. This yields both the container-pair connectivity and the edge-weights that represent the amount of communication per epoch. The container graph vertex weights are the percentage utilization, measured when running CloudSuite [39] container instances on CloudLab [28] servers. Using the docker stat API [44], we measure the CPU, memory and disk utilizations. We use three different workloads (Memcached, Hadoop , and web-search using Apache Solr).

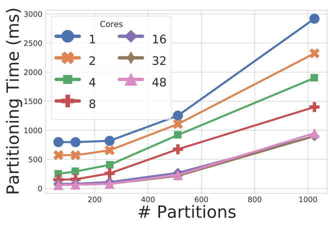
The simulation generates the connectivity graph and the communication (i.e., edge weights), and the testbed measurements provide the utilization (i.e., vertex weights). We combine the two based on the application type (e.g. Memcached, Hadoop, Websearch). For example, if an edge connects a Memcached client to a server, then the vertex weight is that of the Memcached client and server. This graph has 4 million vertices. We consider three metrics for partition goodness and relate them to the container placement problem.

Imbalance: Vertices represent the container resource requirements in a multi-dimensional form (e.g. CPU, MEM, DISK). Our goal is that each partition, depicting a server resource in a DC, should see close to the same, balanced, load. Therefore partition imbalance (e.g., deviation from mean partition weight) must be minimized. Figure 2(a) shows the imbalance, for every dimension, computed as the average absolute difference between each partition's weight and the mean partition weight. In this experiment we set the number of partitions to 1024. Figure 2(a) shows that ParMetis Base partitioning has the lowest partition imbalance. Adaptive partitioning is more imbalanced as it tries to minimize migrations (e.g. it prioritizes vertex moves that place the vertices back to their original partition). The hierarchical approach, Adpar-Adpar, has a similar imbalance as Adaptive partitioning as it relies on the same mechanism. Tabu Search results in higher imbalance as it penalizes vertex migrations more. Finally, Two-Tier (i.e., Adaptive at the coarse level and then Tabu Search) depicts slightly higher imbalance compared to adaptive partitioning as the second tier Tabu Search heavily penalizes container migration within the sub-graph.

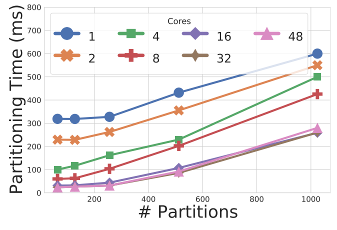
Edge Cut: Edge weights represent the communication between the containers, and the edge cut denotes the communication intensity between partitions. As partitions represent the placement of containers on a physical server in our case, the more we reduce the edge cut, the more we take advantage of container pair affinity with frequently communicating containers being placed closer together. Figure 2(b) shows the edge cut relative to base partitioning, $\frac{e}{e_{BP}}$, where e denotes the edge cut provided by the partitioning algorithm and e_{BP} denotes the edge cut afforded by Base partitioning. Adaptive partitioning and Adpar-Adpar depict very similar edge cut and sometimes even lower than Base partitioning. As Tabu Search heavily penalizes vertex migrations that are required to minimize edge cut, standalone Tabu search has a higher edge cut. Finally, Two-Tier improves over Tabu search by leveraging the lower edge cut across coarse-grained partitions generated by Tier-1.

Migrations: The graph is repartitioned periodically to take into account the change in workload that may result in different assignment. We express migrations as a percentage of the total number of containers running during a given time interval. Figure 2(c) and 2(d) shows the percentage of containers migrated as a consequence of two different types of perturbations: namely, **workload changes** caused by partitioning every 10 minutes considering the graph snapshots at that time; and **consolidation** where the number of target partitions are reduced by one to save energy. For hierarchical partitioning schemes, as the Tier-2 sub-graph partitions will correspond to servers, consolidation only reduces the target number of partitions in the Tier-2 step. The target number of partitions for the Tier-1 partitioning remains unchanged as the coarsened partitions correspond to pods as described below. Figure 2(c) shows that Base partitioning is ill-suited, similar to Metis used in [9], for DC container placement, since the number of migrations is very high. This is because Base partitioning does not take the previous partitioning result into account and only tries to aggressively minimize edge cut and imbalance. However, Adaptive partitioning, shows very few migrations for workload changes as it takes the previous partitioning solution into account. But, it fails to yield the same low number of migrations when there is consolidation as compared to the original partitioning (e.g., previous epoch's partitioning result). It uses migrations to mitigate either the poor balance or edge cut when one partition is taken away.

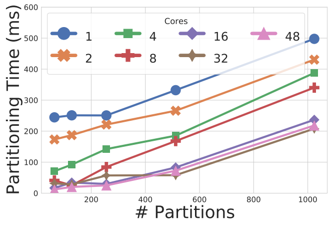
Figure 2(d) shows that none of the approaches using ParMetis (e.g., Base or Adaptive) performs as well as Tabu Search when considering consolidation in terms of vertex migrations. Tabu Search's custom cost function allows us to provide a higher penalty for migration. With hierarchical approaches, the first tier is not impacted by consolidation because the target number of partitions is fixed, but the the number of partitions for the second tier may reduce. Adpar-Adpar suffers because the second tier Adaptive partitioning has too many migrations. The Two-Tier scheduler drastically outperforms other approaches, except Tabu-Search, when considering consolidation. This is because the target number of partitions for Tier-1 is unchanged and Tier-2 heavily penalizes migrations by assigning a higher weight to migrations in its formulation, while also including edge cut and imbalance.



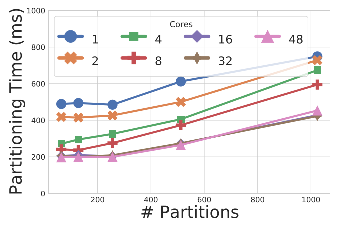
((a)) ParMetis Base



((b)) ParMetis Adaptive



((c)) Adpar-Adpar



((d)) Two-Tier

Fig. 3: Partitioning time for different platforms.

Partitioning Time: Figure 3 shows the partitioning time on a 48-core machine by varying the number of CPU cores devoted to computing the partitioning algorithm. The parallelism (i.e., more CPU cores) helps reduce the partitioning time for all the approaches. It shows the benefit of using ParMetis instead of Metis, used in Goldilocks [9]. On the other hand, the partitioning time increases with the increase in the target number of partitions. Comparing Figure 3(a) and 3(b), Adaptive partitioning is more scalable than base partitioning, taking much less time to partition the graph. Latency with Adpar-Adpar in Figure 3(c) goes down slightly relative to Adaptive partitioning. This is because the Tier-1 no longer has to generate a partitioning result with significant number of partitions, thereby reducing the total partitioning time. Lastly, Two-Tier takes slightly longer time than Adaptive partitioning as it has to spend the second tier in the slower Tabu Search.

IV. pMACH: Distributed Container Scheduling

We need to rapidly partition container graphs with minimal communication overhead. End-hosts are responsible for transmitting sub-graphs to designated partitioning workers **(pWorker)** in the pod, avoiding communication across pods (e.g., which is unavoidable in the centralized approach). We do this in the background, with the partitioning task in epoch t operating over data gathered in epoch $t-1$. Next, the entire container graph is partitioned using Adaptive Partitioning. It's highly parallel and focuses on container-pairs that communicate across neighbouring pods. Adaptive partitioning also minimizes edge cut in the output partitioning. Lastly, to factor the cost of migrations, since Adaptive Partitioning is poor at consolidation, we run Tabu Search on the same pWorker, which is slower but operates on a smaller graph. Altogether, the distributed architecture for graph partitioning is a key innovation that makes wide-scale, real-time scheduling that is adaptive to workload changes, practical.

A. Two-Tier Hierarchical Scheduler Overview

A scheduler in a DC is typically responsible for the placement of tasks among a large number, typically of the order of 10,000 servers (e.g., as in Google's DCs [16]). Furthermore, as in a fat-tree DC network (DCN) architecture (e.g., [45]), we consider a k-ary fat tree network with $\frac{k^3}{4}$ end hosts. In our context that would roughly translate to 11,664 servers distributed among 36 pods handled by one scheduler. To help manage this scale, factoring in the complexity of the graph partitioning algorithm, pMACH uses a Two-Tier hierarchical scheduler, as shown in Figure 4. pMACH develops a two-level graph partitioning algorithm, namely the ParMetis Adaptive partitioning as the first tier and Tabu Search as the second tier. Tier-1 is responsible for partitioning the container graph over pods using the large-scale, scheduler-wide communication graph as input. Tier-2 is responsible for intra-pod scheduling using a smaller pod-wide communication graph as input. This design is inspired by DC such as [16], [46] that use pods as a logical and physical clustering of DC resources, creating a modular solution that can adapt to different-size DCs.

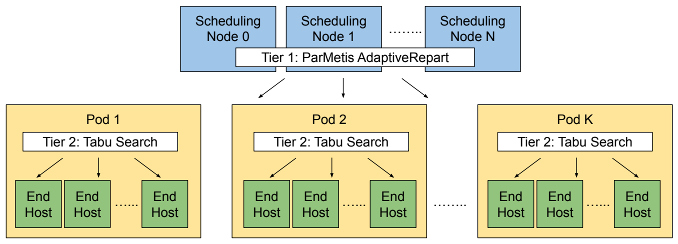


Fig. 4: Two-Tier Hierarchical Scheduler

Adaptive partitioning of a graph with 4 million vertices, with a target of 36 partitions, takes 101ms compared to 5619ms with Tabu Search. This speedup holds even as the number of partitions increase. As the Tier-2 input container sub-graph is small, partitioning takes at most 71ms. Therefore, intra-pod scheduling is managed by Tabu Search for scheduling the containers over fewer, $\frac{k^2}{4}$ (324) servers. Since pMACH employs an epoch based scheduler, we place new containers according to the Best-Fit algorithm [47] with a 70% cap on utilization (i.e., to operate at Peak Energy Efficiency).

In Figure 5, we show the three stages in our distributed container scheduling technique to schedule containers on to servers. The end hosts transmit the communication graph to the designated pWorker within the pod for the purposes of graph partitioning. Next, the Tier-1 ParMetis Adaptive partitioning program is invoked. We designate one pWorker (one core on one select server) per pod to partition the graph. If a scheduler's domain consists of k pods, then Adaptive partitioning will take the container graph as input and generate k partitions. In the next stage, the same pWorker in each pod concurrently run independent instances of the Tier-2 Tabu Search optimization problem where the input graph consists only the containers running in this pod plus the graph changes made by Tier-1 scheduler (i.e., inter-pod migration). Once the serialized Tabu Search procedure for the pod terminates, we migrate containers corresponding to dirty vertices from the original partition to the new partition, where the partition ids correspond to server ids. Some containers may have to be moved to another pod as per the Tier-1 partitioning output.

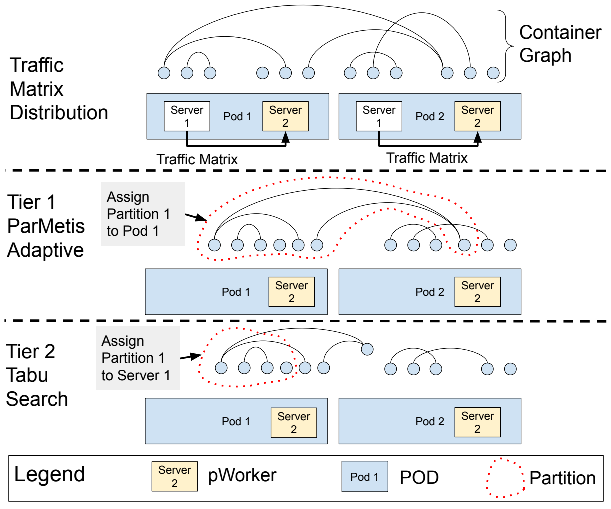


Fig. 5: Distributed Scheduling

B. Tier-1 Scheduler: Container graph scheduling

$$\min \sum_{1\le i<j\le n} |E^c_{i,j}| \tag{1}$$

$$W^1_d \approx W^2_d .. \approx W^n_d,\ where\ d \in D \tag{2}$$

$$\forall P_i, \sum_{j\in P_i} A^c_j \le B^c_i,\ where\ 1 \le i \le n \tag{3}$$

The Tier-1 scheduler runs ParMetis adaptive partitioning using the objective function to minimize edge cut (Eq. 1). It seeks to ensure the partition weights are almost balanced (Eq. 2). Eq. 3 guarantees that the partition resource demands do not exceed the pod capacities. Here, n represents the number of partitions, $E^c_{i,j}$ is the sum of edge weights between partition i and j, W^i_d represents the d^{th} dimension weight of partition i, where $d \in D$. The dimensions D include CPU, memory, and disk. B^c_i represents the capacity of pod i. P_i is the container group assigned to pod i and A_j represents the resource demands of container j. This cost formulation does not explicitly factor the cost of migrations, but it tries to minimize migrations by leveraging the previous partitioning result along with the assumption that graph changes are small.

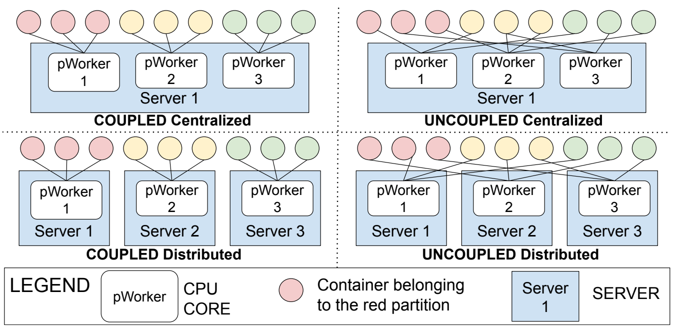
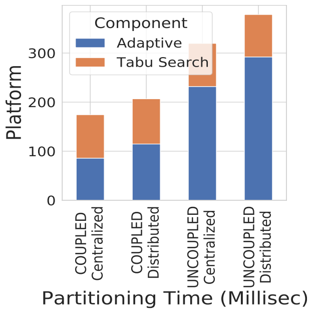
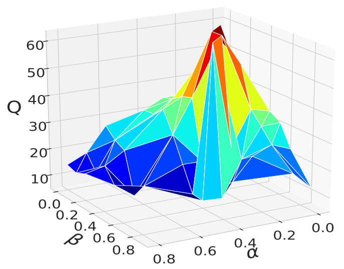


Fig. 6: Vertex distribution over pWorker

Our MPI program can run on a centralized server, where the computation is distributed over CPU cores or in a distributed manner. According to Figure 6, pMACH can distribute vertices (e.g., containers) over pWorkers in four different ways (e.g., centralized/distributed, each being COUPLED/UNCOUPLED). We measure pMACH's partition time overhead for different implementations of ”Two-Tier” as shown in Fig. 6. Partitioning time is the dominant time for scheduling, followed by the migration related down-times, whose impact on the task completion time is studied in section V-A. In Figure 7(a), we see the partitioning time breakdown for Two-Tier, composed of the latency to run ParMetis Adaptive repartitioning (Adaptive) and Tabu Search on a container graph that contains 6 million vertices. We experiment with a distributed approach (16 servers each with 1 core) and a centralized server based approach (1 server with 16 cores). For each, we explore the COUPLED and UNCOUPLED approach.



((a)) pMACH Partitioning Time



((b)) Coeff selection

Fig. 7: Partitioning time and coefficient selection

The COUPLED approach is on average 1.8 times faster than the UNCOUPLED approach. By switching from the UNCOUPLED to COUPLED Tier-1 partitioning scheme, we see a significant speedup because Tier-1 is responsible for processing a large-scale container graph, dominating the total partitioning time. We choose the COUPLED over UNCOUPLED approach for its lower partitioning time. In a centralized approach, the data collected across end-hosts in all pods must be transmitted over the network to a single server (or a group of servers). Typically, communication spanning pods is more expensive since more links are traversed. The distributed approach, reserves one pWorker per pod, resulting primarily in intra-pod communication (from end hosts in the pod to the pod's designated worker). Therefore, the COUPLED distributed approach reduces the communication overhead across pods. Hence, we opted for the COUPLED distributed approach over the COUPLED centralized approach. The COUPLED-centralized scheduler is 1.18 times faster compared to the COUPLED-distributed approach. This is because workers communicate over shared memory in the COUPLED-centralized scheduler as opposed to network links. We tolerate this partitioning time penalty to minimize cross-pod communication during the traffic matrix distribution phase. *In summary, we select the* ***COUPLED distributed*** *scheduling mechanism as it is scalable and faster.* The complexity analysis for Tier 1 and 2 can be found [48] and [49], respectively.

C. Tier-2 Scheduler: Intra-pod scheduling

We partition the intra-pod container sub-graph at the Tier-2 scheduler once the Tier-1 scheduler has computed and transmits the changes to the container graph to each of the pods. The intra-pod container graph is the graph involving the containers (i.e., the vertices) within that pod. The Tier-2 scheduler runs a sequential Tabu Search algorithm using an objective function as shown in Eq. 4.

$$\begin{aligned}
&\min \quad \alpha \times EC + \beta \times IB + \gamma \times DV \\
&EC \quad = \sum_{1\le i<j\le n} |E^c_{i,j}|/E \\
&IB \quad = \frac{1}{n|D|}\sum_{d\in D}\sum_{1\le i\le n} |W^i_d - \overline{W_d} \quad |/\overline{W_d} \\
&DV \quad = \sum_{v\in V} I_{P_{t-1}(v)\neq P_t(v)}/|V|
\end{aligned} \tag{4}$$

$$\forall Q_i, \sum_{j\in Q_i} A^c_j \le S^c_i, \ where\ 1\le i\le n \tag{5}$$

The objective function consists of three parameters (e.g α, β, and γ) that act as weights to the edge cut (EC), imbalance (IB), and dirty vertices (DV). The indicator variable $I_{P_{t-1}(v)\neq P_t(v)}$ equals 1 when the vertex is assigned to a different partition as compared to the original partition, otherwise 0. To ensure edge cut, imbalance, and dirty vertices are dimensionless, we normalize the multi-objective function using the following: E represents the total edge weight and $|V|$ represents the total number of vertices. Equation 5 ensures that the resource demands do not exceed server capacity. S^c_i represents the capacity of server i. Q_i is the container group assigned to server i. By setting a large value for γ relative to α and β, we penalize vertex migrations more. Therefore, even in the event of consolidation, container migrations are low.

We employ Tabu Search for iterative refinement of our graph partitioning solution, following [26]. Each solution (e.g., candidate) provides a cut, which assigns containers to servers. We start with an original partition. At each iteration, we compute the neighborhood solutions (e.g., current solution + a candidate vertex migration). From this, we filter away Tabu moves and then select the vertex move that satisfies Eq. 4. Next, the current solution is updated and the selected move is stored in a Tabu list for a predefined tenure (e.g., next t iterations). The Tabu list ensures that a local minimum is not returned by discouraging the search from coming back to previously-visited solutions. Under certain circumstances a move that is in the Tabu list can be selected, which is referred to as the aspiration criterion. In our technique, a Tabu move will be selected if it yields a solution that is better than the best solution so far. The Tabu Search program stops if a fixed Max Iterations value is reached or if there was no improvement in b iterations. Next we describe how we select the coefficient for Tabu Search. First, we compute different partitioning outputs of the same graph by varying α and β (e.g., $\alpha+\beta+\gamma=1$). We then summarize the partition quality Q as $\frac{1}{N(EC)}+\frac{1}{N(IB)}+\frac{1}{N(DV)}$ (see Eq. 4) where N scales the individual quantity to the range $[0, 1]$ across all partitioning outputs of the same graph. Fig 7(b) shows how Q changes with α and β. This plot summarizes multiple partitioning outcomes over several input graphs, using the median Q. We observe (α, β, γ) equal to (0.234, 0.512, 0.254) maximizes the median value of Q and use it for subsequent experiments.

D. Distributed Monitoring using sNIC

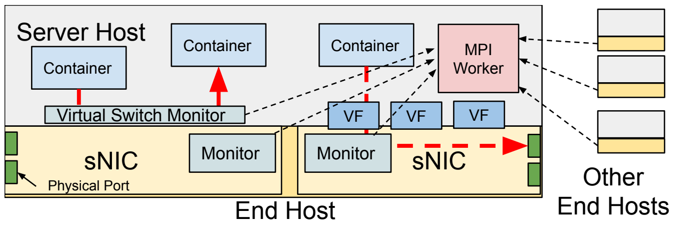


Fig. 8: Pod Architecture: Monitoring and Graph Partitioning

Unlike previous approaches for scheduling and power management in DCs where the communication graph is assumed to be known (or ignored), pMACH explicitly accounts for it and utilizes a sNIC to collect the container-level communication graph. The intra-host container communication graph is collected by the software switch running on the host [50] while the inter-host container communication graph is collected by the sNIC, which are a often used in today's DC networks [51]. We assume every node contains a sNIC to offload the traffic matrix collection, reducing the overhead on the host while collecting the information at link speed [51], [52].

Figure 8 shows the pod-architecture. All servers in the pod are responsible for generating the communication graph. The container resource utilization weights along with intra-host and inter-host container communication weights are transmitted to the designated server within that pod for graph-partitioning. The vertex weights are derived using the docker stat API [44]. The edge weights are measured in the sNIC and software switch. As communication weights help characterize affinity, estimating them helps us trade-off between space (memory and bandwidth) vs. partitioning quality. Reducing bandwidth consumption helps reduce interference for latency sensitive user traffic. Reducing the partitioning quality degrades application task completion time. It is possible to seek a balance between space and application performance. We describe four data plane algorithms below that we evaluated to collect the communication graph edge weights. Based on the analysis in section V-B, we choose Elastic Sketch.

Confluo: Uses a data structure called Atomic MultiLog that supports highly-concurrent read-write operations [53]. Since all the edge weights are accurately recorded, Confluo occupies a lot of space, but has no error estimating edge weights.

CountMIN Sketch: A compact space data structure for summarizing data streams. It uses hash functions to map container-pair communication events to frequencies (e.g., edge weight) at the expense of overcounting due to collisions [54].

Elastic Sketch: It consists of two parts: a "heavy" part recording high-affinity container pair communication weights and a "light" part recording low-affinity container pair communication weights. The heavy part is a hash table while the light part is CountMIN Sketch [27].

Nitro Sketch: It combines a Count Sketch [55] with a sampling strategy to reduce the number of communication edge weight update operations. It is the fastest, but results in higher error estimating communication weights [50].

E. Mapping Partitions to Servers

The Tier-1 ParMetis scheduler equates pods to partitions while the Tier-2 Tabu Search scheduler equates servers to partitions. This is possible because both ParMetis and Tabu Search support heterogeneous partitions. In ParMetis Adaptive partitioning, the user can supplement the $tpwgts$ argument to regulate the fraction of vertex weight that should be distributed to each partition (e.g. pod) for each dimension [41]. Likewise in Tabu Search, vertex moves that violate server capacity can be deemed as illegal moves (e.g., non-neighboring moves). Therefore, both ParMetis and Tabu Search can capture heterogeneous pod and server capacities. Different CPU speeds (e.g., GHz) can also be captured by adjusting vertex weights.

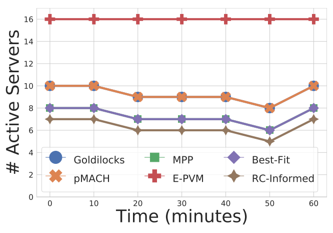
V. Evaluation

We evaluate pMACH on a testbed implementation and compare it with a number of alternatives published in the literature, viz., E-PVM [21], MPP [17], Goldilocks [9], Best-Fit [47] (e.g. Borg stand-in) and RC-Informed [15]. We also do limited measurements with a sNIC. Finally, we carry out large scale simulations to predict the performance of DCs.

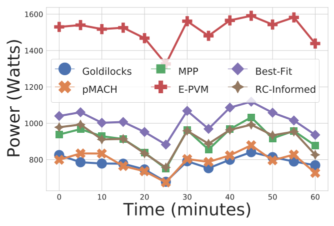
Testbed: We run the Cloudsuite benchmark, which has 432 containers on a testbed containing 16 servers (each with 20 cores) on Cloudlab [28]. The container graph is obtained from running the CloudSuite [39] workload components Memcached, Hadoop MapReduce, and Apache Solr (an equal number of instances of each). The vertex weights, depicting server resource consumption is measured using the docker stat API [44]. CloudSuite benchmarks comprise multiple containers with communication between each other, and the graph's edges characterizes this connectivity. We consider the provisioning of new services where containers are created and killed in the trace at different points in time based on real production datacenter traces [43]. We used the IPTraf monitoring tool [56] to measure the communication rate between pairs of containers, which is used as the edge weight. IPTraf monitors the virtual port for each container. Similar to [2], [9], servers with no active containers are turned off to reduce power consumption. The epoch length is 10 mins. We empirically determined that scheduling decisions with epoch lengths $<$ 10 mins, made it prone to transient changes, unsuitable for partitioning.

A. Testbed Results

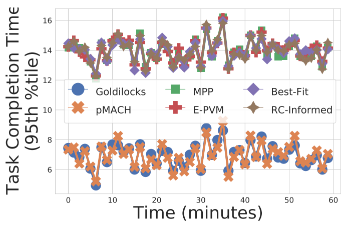
Partitioning Quality: In Figure 9(a) we plot the number of active serves over time for different scheduling techniques. It is observed that E-PVM occupies the largest number of servers, as it places containers in the least utilized server. The bucket based RC-Informed technique yields the lowest number of active servers due to CPU over subscription. Best-Fit packs containers at 95% utilization, yielding fewer servers compared to Goldilocks and pMACH that pack at 70% utilization to operate at peak energy efficiency. MPP tries to minimize power consumption by greedily increasing the target utilization on the server. Its curve overlaps with that of Best-Fit in Fig. 9(a). In Figure 9(b) we see that the power consumption with Goldilocks and pMACH is the lowest, a reduction of 13.44% compared to RC-Informed. RC-Informed consumes less power than E-PVM, Best-Fit, and MPP because it occupies fewer servers. Goldilocks and pMACH consume less power by running at utilizations that result in peak energy efficiency in terms of tasks completed for the energy consumed. Packing at 70% utilization also provides more head room to sustain CPU utilization spikes. In these experiments we have three different container workloads running (e.g. Memcached, Hadoop, and Web-Search). In Figure 9(c) we show the 95th percentile task completion time for the Twitter Memcached Workload by varying the RPS between 44K to 440K across the entire testbed, which effectively varies the resource utilization for the containers. The task completion time is measured at the Memcached-client as it issues get and set requests to the Memcached-servers. As expected, Goldilocks and pMACH show a substantial improvement, with the 95th percentile task completion time speedup of 2.011 as it takes account of the container pair affinity.



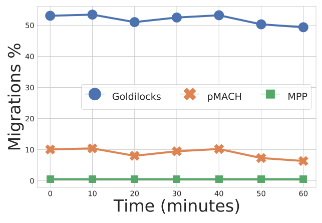
((a)) Active Servers



((b)) Power Consumption



((c)) Task Compl. Time (Memcached)



((d)) Container Migrations

Fig. 9: Scheduling quality with a testbed containing 16 servers.

Goldilocks did not do incremental graph partitioning. At every epoch, Goldilocks re-partitioned from scratch and assumed the containers migrate to their new location, while ignoring the overhead of migrations. Here, we incorporate the impact of container migrations of pMACH in the application-level metric of task completion time. In this experiment, containers are migrated as per the scheduler's decision, using CRIU. Much of the scheduling activity is performed concurrently with task execution. But, the downtime due to migrations directly impacts the task completion time. The average downtime is 2.39s. However, the parallelism in performing 44 migrations with all 16 servers results in 6.06s downtime. While this downtime could be reduced by live migration optimizations, e.g., [57], it is crucial to reduce the amount of migrations, as we strive with pMACH. pMACH still achieves close to Goldilocks' completion time by minimizing dirty vertices. Figure 9(d) shows container migration events as a percentage of total number of containers. Unlike Goldilocks, MPP and pMACH, the other scheduling mechanisms distribute containers only when they arrive, and have no migrations. Goldilocks has average 51.8% migrations per epoch. But, pMACH only has 8.83% migrations, benefiting from its incremental partitioning approach. MPP has the least migrations as it migrates containers only when the server's utilization deviates from the target utilization. This results in poorer task completion time and higher power consumption than pMACH.

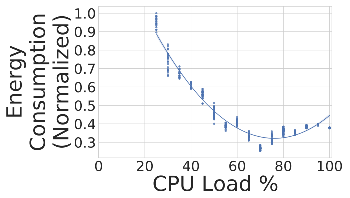


Fig. 10: CPU Load vs. Normalized Energy

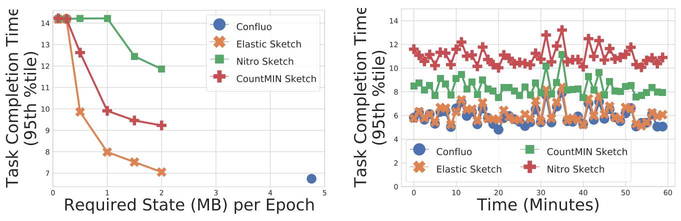
Energy Consumption: We now verify if indeed running at 70% utilization is efficient for overall DC energy consumption. We replicate experiments for Figure 9, but with different maximum caps on CPU utilization, to examine sensitivity to that parameter. Figure 10 shows the energy consumption, measured using RAPL [33], for 16 servers against different levels of packing capacities using pMACH's scheduler. Consistent with previous work on Peak Energy Efficiency [13], [35] we observe that the total power consumption is the lowest when the utilization per server is capped at 70%.

B. Distributed Data Collection

We study different ways to collect the communication graph on the sNIC using the trace to derive the partitioning quality in section V-A. This section emphasizes the reduction in network bandwidth for transmitting the graph and the resulting tradeoff in application performance. We replay the packet trace over a sNIC, transfer the data collected from the sNIC to the host, and partition the communication graph using the data collected from the sNIC as well as the intra-host container communication obtained by the cluster metrics collection framework. All the approaches other than Confluo [53] use some sort of probabilistic approximation, allowing them to have lower packet processing and data transmission overhead, but are more prone to estimation error. In all the data structures the tuple key is the source and destination container IP. We also use a local testbed consisting of server with 20 Intel Xeon 2.20GHz CPU cores and 256GB memory running Linux (4.4.0-142). It has Netronome Agilio LX 2×40 GbE sNICs which have 8GB DDR3 memory and 96 flow processing cores.

Figure 11(a) shows the task completion time vs the overall memory usage at all the servers in a 16 server implementation. We run the pMACH graph partitioning technique to determine the container placement, similar to the testbed experiments of section V-A. We compute the 95th percentile of task completion time for the Memcached workload. Memory usage reflects the amount of data that must be transferred over the wire. Confluo consumes the most amount of memory compared to all other platforms as it must track all edges, but also yields the lowest task completion time. Nitro Sketch and CountMIN Sketch result in high task completion times because of the edge weight overestimation that results in many of the low-affinity containers pairs being scheduled together. These low-affinity containers compete with container-pairs that actually have high affinity and thus result in overall poor placement. We observe Elastic Sketch has a negligible increase in task completion time compared to Confluo but with a substantial memory usage reduction of 2.38 times. This is because Elastic Sketch prioritizes the retention of heavy flows, in turn preserving container-container affinity.

Figure 11(b) shows the result of the experiment in the form of a time series, where the memory state of the data summarization approach is transferred every epoch. Confluo transfers 4.76MB of data per epoch, while the other dataplane algorithms transfer only about 2MB data per epoch. Clearly, Elastic Sketch, despite using 57.98% less bandwidth, has low application task completion time, matching Confluo.



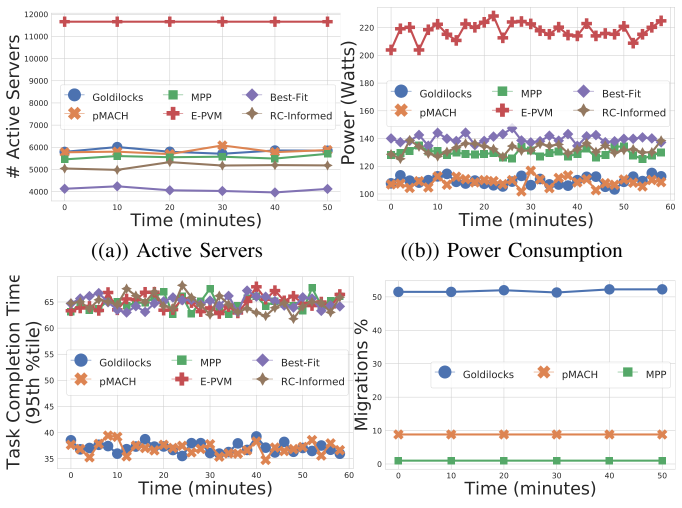
((a)) Sensitivity to amount of state ((b)) Time Series

Fig. 11: Task Completion Time w/other monitoring approaches

C. *Large Scale Simulation Results*

We also performed a flow-level, large scale simulation with a 36-ary fat tree topology, with $11,664$ servers and a total of $104,958$ containers (e.g., targeting 20-30% utilization for baseline E-PVM [5]–[7]). We utilize a trace from a NS3-based DC simulation [43]. We obtain the communication matrix time series for different workloads (i.e., Memcached, Hadoop, and Microsoft Web Search). This represents the container graph edge weights and connectivity. We then merge the graph edges with container graph vertex weights per application (e.g., Memcached). We ran actual instances of containers from the Cloudsuite [39] benchmark on CloudLab [28] and measured resource demands to get the vertex weights.

In Figure 12(a), we see E-PVM always chooses the least utilized server, but all 11664 servers are active. RC-Informed requires the least number of servers because it permits over-subscription of server resources. Best-Fit requires a slightly higher number of servers, because it sets target of 95% utilization. Similarly, MPP does not oversubscribe server resources, but greedily selects the servers to provision, based on the power model. Goldilocks and pMACH use more of servers as they operate at a lower target of 70% utilization. But, as we see in Figure 12(b) for the power consumption, using the power model we measured on CloudLab [28], Goldilocks and pMACH consume the least energy as they operate at peak energy efficiency. In Figure 12(c) we compute the 95th percentile



((a)) Active Servers ((b)) Power Consumption

((c)) Task Completion Time ((d)) Container Migrations

Fig. 12: Scheduling quality using trace driven simulation.

task completion time for Apache Solr search engine in the testbed and vary the search request rate. We use the processing time distribution for search queries based on a benchmark measurements made in CloudLab [28]. We use packet latency obtained from measuring the container pair communication latency in our testbed using the Arista 7050SX-72Q switch. We ignore the queuing delays based on typically low link utilizations in DCs ($< 25\%$) [58]. By accounting for container-pair affinity, the 95th percentile task completion speeds up by 1.76x. Lastly, in Figure 12(d) we see that the migrations reduce from 51.70% with Goldilocks to 8.7% with pMACH. MPP's migrations remain below 1.3%, but consumes 19% more power and 76% longer task completion time. This is because MPP does not operate at peak energy efficiency and ignores container affinity.

VI. Conclusion

pMACH is a framework that solves the complex provisioning problem in containerized data centers. pMACH includes a novel graph-based locality aware container placement scheme that significantly reduces power consumption, task completion time, and migrations. We show that by operating server resources at Peak Energy Efficiency, we both save power and provide greater headroom for traffic spikes. Our Two-Tier distributed graph partitioning architecture can scale to tens of thousands of servers and compute the partitioning result quickly. By carefully CPU cores in selected servers in each pod of the data center, and taking advantage of the high-bandwidth data center links, we split the graph partitioning into a hierarchical solution. pMACH tracks container-to-container communication and uses data stream summarization techniques to communicate the traffic matrices efficiently to designated servers in each pod for partitioning the graph.

VII. Acknowledgement

We sincerely thank the US NSF for their generous support through grants CCF-1815643 and CNS-1763929. We thank our shepherd Dr. Ying Zhang and the anonymous reviewers for their suggestions and comments.

References

[1] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić, "Identifying and using energy-critical paths," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2079296.2079314

[2] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. USA: USENIX Association, 2010, p. 17.

[3] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," in *2011 Proceedings IEEE INFOCOM*, 2011, pp. 1098–1106.

[4] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy proportionality and workload consolidation for latency-critical applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 342–355. [Online]. Available: https://doi.org/10.1145/2806777.2806848

[5] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, p. 205–216, Mar. 2009. [Online]. Available: https://doi.org/10.1145/2528521.1508269

[6] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 598–610.

[7] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, p. 301–312.

[8] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 123–137, Aug. 2015. [Online]. Available: https://doi.org/10.1145/2829988.2787472

[9] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Goldilocks: Adaptive resource provisioning in containerized data centers," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 666–677.

[10] "Live migration using criu," https://criu.org/Docker.

[11] "Gcp live miogration," https://cloud.google.com/compute/docs/instances/live-migration.

[12] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "Timetrader: Exploiting latency tail to save datacenter energy for online search," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 585–597.

[13] D. Wong, "Peak efficiency aware scheduling for highly energy proportional servers," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 481–492. [Online]. Available: https://doi.org/10.1109/ISCA.2016.49

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[15] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, October 2017.

[16] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[17] A. Verma, P. Ahuja, and A. Neogi, "pmapper: Power and migration cost aware application placement in virtualized systems," in *Middleware 2008*, V. Issarny and R. Schantz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 243–264.

[18] "Kubernetes resource bin packing." [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/resource-bin-packing/

[19] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," January 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/

[20] "Aws ecs bin packing." [Online]. Available: https://aws.amazon.com/blogs/compute/amazon-ecs-task-placement/

[21] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "An opportunity cost approach for job assignment in a scalable computing cluster," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 7, pp. 760–768, 2000.

[22] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0," http://www.cs.umn.edu/~metis, University of Minnesota, Minneapolis, MN, 2009.

[23] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, vol. 47, pp. 109–124, 1997.

[24] A. Lim and Y.-M. Chee, "Graph partitioning using tabu search," in *1991., IEEE International Sympoisum on Circuits and Systems*, 1991, pp. 1164–1167 vol.2.

[25] E. Rolland, H. Pirkul, and F. Glover, "Tabu search for graph partitioning," *Annals of Operations Research*, vol. 63, pp. 209–232, 04 1996.

[26] M. Jahanian, J. Chen, and K. K. Ramakrishnan, "Graph-based namespaces and load sharing for efficient information dissemination in disasters," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, 2019, pp. 1–12.

[27] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 561–575. [Online]. Available: https://doi.org/10.1145/3230543.3230544

[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[29] K. Zhou, W. Wan, X. Chen, Z. Shao, and L. T. Biegler, "A parallel method with hybrid algorithms for mixed integer nonlinear programming," in *23rd European Symposium on Computer Aided Process Engineering*, ser. Computer Aided Chemical Engineering, A. Kraslawski and I. Turunen, Eds. Elsevier, 2013, vol. 32, pp. 271–276. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444632340500464

[30] M. P. Forum, "Mpi: A message-passing interface standard," USA, Tech. Rep., 1994.

[31] "Google cloud platform: Micro services demo." [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[32] "Kubernetes nodename." [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[33] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3177754

[34] "Linux manual page: dd." [Online]. Available: https://man7.org/linux/man-pages/man1/dd.1.html

[35] D. Wong, "Peak efficiency aware scheduling for highly energy proportional servers," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 481–492. [Online]. Available: https://doi.org/10.1109/ISCA.2016.49

[36] "Intel turbo boost." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[37] C.-H. Hsu and S. W. Poole, "Revisiting server energy proportionality," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 834–840.

[38] D. Wong, J. Chen, and M. Annavaram, "A retrospective look back on the road towards energy proportionality," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 110–111.

[39] "Cloudsuite." [Online]. Available: https://github.com/parsa-epfl/cloudsuite

[40] "Task migration at scale using criu." [Online]. Available: https://linuxplumbersconf.org/event/2/contributions/69/

[41] "Parmetis manual." [Online]. Available: http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf

[42] N. Desai, R. Bradshaw, A. Lusk, and E. Lusk, "Mpi cluster system software," in *Recent Advances in Parallel Virtual Machine and Message*

Passing Interface, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 277–286.

[43] "D-salt." [Online]. Available: https://aistein.github.io/d-salt/

[44] "Docker stats." [Online]. Available: https://docs.docker.com/engine/reference/commandline/stats/

[45] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 39–50. [Online]. Available: https://doi.org/10.1145/1592568.1592575

[46] F. Engineering, "Introducing data center fabric, the next-generation Facebook data center network," https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[47] "Bin packing." [Online]. Available: https://www.ics.uci.edu/~goodrich/teach/cs165/notes/BinPacking.pdf

[48] G. Karypis and V. Kumar, "Kumar, v.: A fast and high quality multilevel scheme for partitioning irregular graphs. siam journal on scientific computing 20(1), 359-392," *Siam Journal on Scientific Computing*, vol. 20, 01 1999.

[49] H. Pirim, E. Bayraktar, and B. Eksioglu, *Tabu Search: A Comparative Study*, 09 2008.

[50] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 334–350. [Online]. Available: https://doi.org/10.1145/3341302.3342076

[51] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[52] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 318–333. [Online]. Available: https://doi.org/10.1145/3341302.3342079

[53] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 421–436. [Online]. Available: https://www.usenix.org/system/files/nsdi19-khandelwal.pdf

[54] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, p. 58–75, Apr. 2005. [Online]. Available: https://doi.org/10.1016/j.jalgor.2003.12.001

[55] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ser. ICALP '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 693–703.

[56] "Iptraf." [Online]. Available: http://iptraf.seul.org/

[57] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, "Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 121–132. [Online]. Available: https://doi.org/10.1145/1952682.1952699

[58] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 267–280. [Online]. Available: https://doi.org/10.1145/1879141.1879175