Elastic Resource Allocation for Distributed Graph Processing Platforms

Ravikant Dindokar and Yogesh Simmhan
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
ravikant7@ssl.serc.iisc.in, simmhan@serc.iisc.in

ABSTRACT

Distributed graph platforms like Pregel have used vertex-centric programming models to process the growing corpus of graph datasets using commodity clusters. The irregular structure of graphs cause load imbalances across machines operating on graph partitions, and this is exacerbated for non-stationary graph algorithms such as traversals, where not all parts of the graph are active at the same time. As a result, such graph platforms, even as they scale, do not make efficient use of distributed resources. Clouds offer elastic virtual machines (VMs) that can be leveraged to improve the resource utilization for such platforms and hence reduce the monetary cost for their execution. In this paper, we propose strategies for *elastic placement of graph partitions on Cloud VMs* for a subgraph-centric programming model to reduce the cost of execution compared to a static placement, even as we minimize the increase in makespan. These strategies are innovative in modeling the graph algorithm's behavior *a priori* using a metagraph sketch for the large graph. We validate our strategies for several graphs, using runtime traces for their distributed execution of a Breadth First Search (BFS) algorithms on our subgraph-centric GoFFish graph platform. Our strategies are able to reduce the cost of execution by up to 42%, compared to a static placement, while achieving a makespan that is within 29% of the optimal.

Categories and Subject Descriptors

[**Networks**]: Cloud computing; [**Computing methodologies**]: Distributed computing methodologies; [**Computing methodologies**]: Planning and scheduling

General Terms

Design, Performance, Experimentation

Keywords

Graph processing, Elastic Scheduling, Big Data, Cloud Computing, Distributed System

1. INTRODUCTION

Graph algorithms are challenging to design, program and execute in parallel due to their irregular nature. There has been a rapid growth of large graph datasets, ranging from social networks [3] and knowledge graphs [2], to power and road infrastructure graphs [1] and connectivity of the Internet of Things. At the same time, there has been a push toward developing Big Data platforms for graph processing on commodity clusters and Clouds. Distributed graph programming models such as Google's Pregel [13] and GraphLab [11] leverage an intuitive vertex-centric approach to specifying graph algorithms, where users specify the logic for a single vertex and this is executed in parallel across all vertices, with message passing or state transfer between them. These have been extended to other component-centric variants [20, 21] that execute iteratively using a Bulk Synchronous Parallel (BSP) model. However, despite their use of commodity clusters, there has not been work on effectively leveraging *elastic* Cloud resources for such irregular graph platforms.

Many of the newer graph platforms have been able to scale with the size of the graph, but they do not necessarily achieve a high efficiency of execution. For e.g., a vertex-centric programming model like Pregel is nominally able to achieve a balanced computation across different partitions (machines) due to similar number of vertices on each partition [1, 13], and edge-balanced partitioning has been attempted too [16]. But a balancing of the topology across machines translates to a balanced CPU utilization of all the machines only for *stationary graph algorithms* [10], where all vertices or edges are actively computing during the entire algorithm. Such algorithms, like PageRank or Bi-partite connectivity, can achieve good load balancing across machines.

However, when the algorithm itself moves to different parts of the graphs over different supersteps (iterations), the load across different machines gets out of balance. Such *non-stationary algorithms* [10] include traversals (e.g., breadth-first search, single source shortest path) and centrality (e.g. between-centrality) algorithms. For e.g., in BFS, the partition containing the source vertex is active in the first superstep and as the traversal progresses, its neighboring partitions get active in subsequent supersteps, and so on. As a result, only a subset of the partitions are active at a time with their host machine's CPU being used, even as the other machines holding inactive vertices are under-/un-used. For example, the average utilization for BFS, using a subgraph-

---

[1] http://www.dis.uniroma1.it/challenge9/download.shtml

centric model, over the USA Road Network (23M vertices, 85M edges) running on 8 machines is 35% (Fig. 2).

Cloud computing has been actively used by Big Data platforms due to it easy access to commodity infrastructure. One of its key benefits is the elastic access to resources, that allows virtual machines (VMs) to be acquired and released on-demand with a pay-as-you-go pricing. Infrastructure as a Cloud (IaaS) rent out VMs by the hour (Amazon AWS) or even by the minute (Google Compute and Microsoft Azure). Irregular distributed graph algorithms, operating on large graphs, can take 100's of core-minutes to run on commodity hardware. *Hence, this offers the opportunity to control the VM elasticity such that only partitions that are active take up VM resources*, thereby reducing the monetary cost of execution. There has been limited work on actively using Cloud elasticity for graph platforms. We address this gap in this paper by proposing partition activation and placement strategies on Clouds for graph platforms.

Two key intuitions drive our approach: (1) We decouple partitioning of the graph from their placement on VMs for executing a particular algorithm; and (2) We utilize a metagraph [6] sketch of the whole graph to *a priori* model the progress of the algorithm onto various partitions, which guides our placement strategy at each superstep. As a result, for each superstep, we are able to activate VMs and place relevant partitions on them, and conversely, deactivate VMs without active partitions. In the process, we reduce the monetary cost of execution on the Cloud with minimal impact on the runtime of the algorithm.

The rest of this paper is organized as follows. In § 2 we discuss related work on distributed graph processing and partitioning strategies. In § 3, we review our prior work on the GoFFish subgraph-centric distributed platform and the concept of metagraphs, which are used in our approach and evaluation. We define the system model considered and formalize the problem of partition placement onto VMs in § 4. § 5 introduces our proposed partition placement and VM activation strategies. We validate these strategies in § 6 using execution traces collected from real graphs for the Breadth First Search (BFS) algorithm and evaluate the potential benefits in reducing the VM cost. Lastly, in § 7 we present our conclusions and discuss future work.

2. RELATED WORK

MapReduce has been a staple platform for Big Data processing but graphs present challenges to its tuple-centric model. Its shortcomings for graph algorithms, which tend to be iterative, are due to the repetitive costs for disk I/O, both to reload the graph and to pass state, for each iteration [5]. To address this, Google's Pregel [13] offers a vertex-centric programming model that keeps the graph in memory, and uses an iterative Bulk Synchronous Parallel (BSP) execution model where messages are passed between vertices for state transfer at superstep boundaries. Within a superstep, active vertices execute their `compute()` method once, in parallel, and this method has access to that vertex's prior state and any incoming messages from its neighbors from the previous superstep.

All vertices are *active* when the Pregel application begins, and a vertex can locally *vote to halt* as part of its compute logic, when it "appears" to have finished. This makes it *inactive*, and the compute method of an inactive vertex is not invoked in a superstep. Inactive vertices can be revived and active again when a new message is received by them at a superstep. When all vertices are inactive in a superstep, a global *vote to halt* is reached and the application terminates.

Pregel has spawned Apache Giraph [1] as an open source implementation, and other optimizations to its programming and execution models. Giraph++ [21], Blogel [22] and our own work on GoFFish [20] coarsen the programming model to operate on partitions or subgraphs, with Giraph++ using partitions, GoFFish on subgraphs (weakly connected components, WCC) and Blogel on either vertices or blocks (WCC). This gives users more flexible access to graph components that can lead to faster convergence, and also reduces fine-grained vertex-level communication. This paper aims to use elastic Cloud VMs for such component-centric systems.

Distributed graph processing systems divide the graph into a number of partitions which are placed across machines for execution. The quality of partitioning impacts the load on a machine, cost of communication between vertices, and the iterations required to converge. A variety of partitioning techniques have been tried. Giraph's default partitioner hashes vertex IDs to machines, to balance the number vertices per machine. Other approaches that balance the number edges per partition, for algorithms that are edge-bound, have been tried [9]. GoFFish tries to balance the number of vertices per partition while also minimizing the edge cuts between partitions. This gives partitions with well-connected components that suits its subgraph-centric model. Multi-level partitioning schemes have also been identified to improve the CPU utilization [4]. Blogel further uses special 2D partitoners for spatial graphs to improve the convergence time for reachability algorithms.

However, unlike stationary algorithms [10] like PageRank where all vertices are active on all supersteps, non-stationary traversal algorithms like BFS have a varying frontier set of vertices that are active in each iteration. This results in an uneven workload across different machines. *Hence, a single partitioning scheme, however good its quality may be in achieving some topological balancing, cannot offer compute balancing across hosts, minimize communication and ensure fast convergence, for all types of graphs and algorithms.* We address the lack of compute balancing for non-stationary algorithms here, and the associated suboptimal Cloud costs.

Platforms like Mizan [10] partially address this imbalance by performing vertex migration based on the number of outgoing and incoming messages to vertices and the execution time of a vertex in a superstep. This identifies overloaded machines at the end of each superstep, and the vertices to migrate to less loaded machines. However, this runtime decision causes additional coordination costs to decide and move vertices, and re-synchronize before the next superstep's compute can be started. This can result in an increased makespan for non-stationary algorithms, which their results do not document, and also affect its correctness [12].

GPS [16] too adopts dynamic re-partitioning to reduce communication by co-locating vertices that communicate beyond threshold onto the same machine while also balancing the number of vertices per machine. It only takes into account the outgoing messages from a vertex for this decision, which is in-sufficient for load balancing in non-stationary algorithms. Similarly, [18] tries to balance the workload across the machines by an experimental study of the graph algorithm and a prediction of the number of active vertices in the next superstep to perform vertex migrations.

Such analytical and experimental predictions are difficult and costly for new graphs or algorithms.

Two key distinctions between these works on vertex migration and ours is, (1) rather than balance the workload across a *static set of machines*, we scale-out or -in the number of *elastic VMs* to match the workload on a superstep, and (2) we use an *a priori* analysis of the graph algorithm on a coarse metagraph to model the execution time of partitions *rapidly*, and use this to decide both the number of VMs required and the placement of partitions on VMs. This static planning of migration and scaling for each superstep helps hide the migration costs by interleaving data movement with compute, and can avoid increasing the makespan.

To our knowledge, there is no detailed, existing work on the effective use of elastic VMs to execute component-centric graph frameworks. Our prior work [15] briefly examines dynamic scaling of BSP workers on elastic VMs for a vertex-centric model to reduce the cost of execution of the Betweenness Centrality (BC) algorithm. This uses an intuition that BC has a sinusoidal number of active vertices when launching traversals from multiple source vertices, and this can be used to control the number of VMs. We generalize this model in the current paper using the notion of metagraphs than can be used for many graph algorithms, including BC, and without needing to empirically observe the algorithm's execution on the whole graph.

There has also been some work on algorithmic analysis of component-centric graph algorithms to guide their efficient execution [17, 23]. These identify desirable properties for algorithms designed for distributed graph processing systems like Pregel. Our work is complementary to such algorithmic innovations, and such techniques can be applied to analyzing the metagraph too.

Besides Pregel-like systems there are several other distributed graph processing systems. GraphLab [11] uses an asynchronous, though vertex-centric, programming model where a vertex can directly access its neighboring vertex and edge values, without the need for messaging. GraphLab also adopts a fast repartitioning scheme that is user-driven. Powergraph [8] follows a vertex based partitioning and replication to distribute load based on edges. Sedge [24], is a distributed graph querying system over a set of non overlapping partitions, and it can reduce the communication costs at runtime by creating new partitions or replicating them. Trinity [19] is a distributed in-memory key-value store to perform graph analytics such as path traversal on RDF data, which is treated as a graph. However, none of these distributed graph processing systems are optimized for elastic Cloud resources, which is the emphasis of this paper.

3. BACKGROUND

We use the GoFFish subgraph-centric graph programming model to translate and validate our approach of using elastic placement strategies for non-stationary graph algorithms. As background, we give details on GoFFish's graph data distribution and execution model, and the ability to construct a metagraph on top of large graphs to analyze the execution of graph algorithms.

3.1 GoFFish Subgraph-centric Model

As described before, GoFFish [20] is a distributed graph processing framework that follows a BSP model to ex-
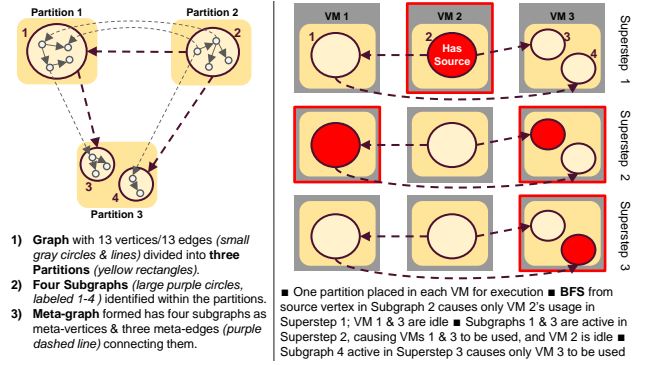

Figure 1: Metagraph formation from partitioning graph (left), and active subgraphs, along with VMs used, during BFS execution in GoFFish (right).

ecute graph application written using a subgraph-centric programming model.

Consider a directed graph $G = \langle V, E \rangle$ is partitioned into m partitions using some relevant partitioning scheme that reduces the edge cuts across partitions and balances vertices in each, and these are distributed over n machines, such that $m \geq n$. A single machine can hold one or more partitions. Partitions are vertex disjoint, meaning that a vertex belongs to exactly one partition. Edges connecting vertices in different partitions are termed as *remote edges*, whereas those having both source and sink vertices in the same partition are called *local edges*. Within each partition, *subgraphs* are formed by identifying weakly connected components, meaning that a partition can have one or more subgraphs, and no two vertices belonging to two different subgraphs are connected by a local edge. For e.g., Fig. 1 (left) shows a graph with 13 vertices and edges partitioned into three, with 4 subgraphs in all identified.

The `compute()` method defined by the user is applied to each subgraph in a superstep, and the method has access to all vertices, and local and remote edges in the subgraph. The method can update the state of local and remote edges, and pass messages to neighboring vertices or subgraphs connected through remote edges that are delivered at synchronized superstep boundaries. For e.g., Fig. 1 (right) shows a BFS that starts at a vertex present in subgraph 2, whose vertices are traversed in superstep 1, followed by traversal of vertices in neighboring subgraphs 1 and 3 in superstep 2, and their neighbor, subgraph 4, in superstep 3. A subgraph may be revisited too, or only a subset of its vertices be traversed, depending on the algorithm. The subgraphs can vote to halt and the execution stops when all subgraphs have voted to halt and no messages are in-flight. This model can also trivially implement a vertex-centric program.

3.2 Metagraphs for Algorithm Modeling

In our previous work [6], we have introduced the concept of *metagraph* which is a coarse-grained sketch over large graphs, and is naturally suited to analyze subgraph-oriented graph applications. In a metagraph, each *meta-vertex* is a disjoint *subgraph* (connected component) in the original graph, and *meta-edges* indicate remote edges that connect these subgraphs. The meta-vertices can have attributes like the number of local vertices and local edges that the subgraph

has, and the weight of the meta-edge can indicate the number of remote edges between the subgraphs. Fig. 1 (left) shows a metagraph with 4 meta-vertices and 3 meta-edges.

Depending on the type of graph and the partitioning scheme, metagraphs have meta-vertices and meta-edges that number several orders of magnitude smaller than the original graph. For e.g., we see metagraphs with 10's or 100's of meta-vertices for graphs with millions of vertices [6]. As a result, this coarse-grained approximation of the large graph helps us rapidly analyze the behavior of several traversal algorithms, and can guide runtime operations.

For e.g., when performing a BFS using a subgraph-centric model, our prior work [6] showed that the order in which subgraph are potentially visited in each superstep can be determined. This is done by performing a BFS from the meta-vertex (subgraph) that holds the source vertex of the BFS, and traversing to neighboring meta-vertices. Each traversal is a superstep, and the meta-vertices visited corresponds to the subgraph(s) that will be active in that particular superstep. This is illustrated in Fig. 1 (right).

This information, combined with an analytical model of the cost of a local BFS on a single subgraph, helped us estimate the number of supersteps required for the algorithm to converge. The metagraph is simple to construct, either at graph partitioning time or by running a simple traversal algorithm using GoFFish that take a few seconds for even large graphs with millions of vertices. Also, the size of the metagraph itself is small enough that it can be analyzed on a single machine using a sequential algorithm.

In this paper, we reuse two important results from our previous work [6] that supports our assumption of *a priori* knowledge of the algorithm behavior. First, given the subgraph holding the source vertex and the metagraph, we can accurately determine the superstep at which a subgraph will be visited for the first time by a BFS and estimate the cost for local BFS to be performed on it. Second, we can predict the supersteps at which a subgraph may be revisited and the BFS potentially repeated on it.

4. PROBLEM

We discuss the high level problem and the system model as context, before giving a formal definition of our problem.

4.1 Decoupling Partitioning from Placement

Vertex and block-centric distributed graph platforms partition a large graph into many partitions, and workers in each host operate upon one or more partitions. These three steps – that of *partitioning* a graph, *placement* of partitions onto a host, and *assigning* workers (threads/processes) on a machine to operate on partitions present on it – are loosely coupled decisions. For e.g., Giraph by default hashes a graph into as many partitions as requested by an application at runtime. There are one or more workers on each machine (equal to the number of mapper slots) and one or more partitions are assigned to a worker, and pushed to its machine. In GoFFish, we allow the user to specify the number of partitions to create at graph load time, allow multiple partitions per machine – typically one per core, and allocate two threads per core to operate on the partition, and each thread works on one subgraph at a time within the partition. However, in both cases, once these bindings are made, they are retained for all supersteps.
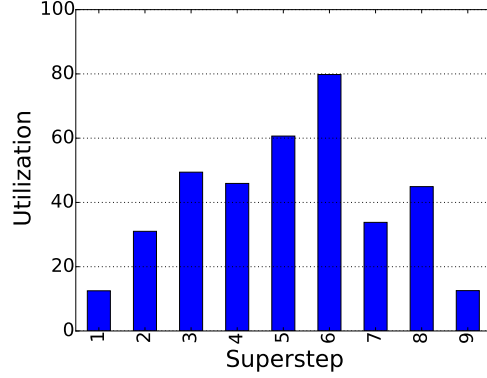

Figure 2: CPU utilization % over different supersteps, for US Road Network graph with 8 partitions executing BFS across 8 cores using GoFFish.

In each BSP superstep, vertices/subgraphs in all or some partitions may be actively performing computation. For non-stationary algorithms, such as BFS or SSSP, the inactive partitions, while taking up disk (and possibly memory) space, do not contribute to the computational usage on these machines. Fig. 2 shows the average utilization of the hosts in each superstep when performing a BFS on a USRN graph (Table 1) with 8 partitions, each using one core, on GoFFish. We see the CPU usage grow from 10% to 80% and drop again, depending on the number of active subgraphs.

Our focus here will be on such non-stationary graph algorithms. Since the compute load on a machine varies with each superstep as a result of which partitions on them are active or not, it can be efficient to consolidate the computational load of active partitions at each superstep on fewer (virtual) machines in order to increase the utilization of these machines. In the context of Cloud Virtual Machines (VMs), this also allows the VMs without any active partitions to be shutdown, and stop incurring monetary costs. This is the thesis of this paper, to explore *how we can reduce the overall monetary cost for running the graph algorithm with minimal impact on the makespan of the algorithm, using partition placement strategies on elastic VMs based on their activation schedule across supersteps, as compared to a traditional hashing of partitions onto a static set of VMs.*

4.2 System Model and Assumptions

There are several reasonable assumptions that have to be satisfied in order for the above goal from being achieved.

- *VM costing should be at a fine time granularity.* Since the graph algorithms may run from 10's of seconds to 10's of minutes, elastic management of VMs will work only when VMs themselves can be acquired, released and billed at similar time scales. Both Google Compute and Microsoft Azure VMs allow billing at 1 VM-min increments, with the former requiring a minimum billing of 10 VM-mins. We use 1 core-min as the smallest billing increment and unit of acquisition/release of small (1-core) VMs. We also assume that the startup and shutdown times are sufficiently small to not impact the costing or usage time of VMs. In future, with light-weight containerization of IaaS Clouds, even per-second billing may be possible.

- *Framework support for elasticity.* The distributed graph platform should allow the persistence of state for a partition and its migration to different VMs at runtime, at superstep boundaries. Giraph support check-pointing of messages and state at supersteps. Other vertex-centric frameworks have also demonstrated this [10, 16]. GoFFish does not yet support this, though there is no design limitation that prevents this. We also investigate placement strategies that pin a partition to a single VM to avoid the need for migration.
- *Modeling graph algorithms.* The (non-stationary) graph algorithm itself should exhibit variable activation of partitions at different supersteps. Stationary algorithms, where all partitions are active at all supersteps, are unlikely to benefit. Also, we should be able to predict the activations of partitions for *a priori* planning at launch time rather than runtime to avoid VM startup and data movement time from increasing the makespan, i.e. through prior planning, we can hide these by interleaving them with the compute time of a superstep. The meta-graph approach [6] discussed above can enable this prediction.

4.3 Problem Definition

Let a directed graph $\mathbb{G} = \langle V, E \rangle$ be partitioned into n partitions, $P_1, P_2, \ldots, P_n$. Say we are given a graph application composed in a partition-centric manner, that takes m supersteps to execute. Let there be a *time function*,

$$\mathcal{A} : P_i \times s \to \tau_i^s$$

that gives the computing time τ_i^s taken to execute a partition P_i in a superstep $s \leq m$ by a single exclusive virtual machine (VM). Here, if the function maps to a time value of 0, that partition is not active in that superstep.

The *minimum makespan* for this application is given by,

$$T_{Min} = \sum_{s=1}^{m} \tau_{Max}^s \text{ , where } \quad \tau_{Max}^s = \max_{i=1}^{n} \left(\tau_i^s \right)$$

i.e., the minimum time to execute this application is the sum across all supersteps of the maximum computing taken among all partitions in each superstep.

Say there is a *partition placement function*,

$$\mathcal{M} : P_i \times s \to v_j$$

that maps each partition P_i for a superstep s uniquely onto a virtual machine $v_j \in v_1, v_2, \ldots, v_l$ that can be used to execute that partition in a superstep. Similarly, let a partition presence function $\widehat{\mathcal{M}}(i, s, j)$ return 1 if a partition i is present in a VM j in superstep s, and 0 otherwise.

The *actual makespan* for the application is given by summing up for each superstep, the time taken by the VM holding partitions that together take the longest time to execute sequentially in that superstep:

$$T = \sum_{s=1}^{m} \left(\max_{j=1}^{l} \left(\sum_{i=1}^{n} \widehat{\mathcal{M}}(i, s, j) \cdot \tau_i^s \right) \right)$$

If the cost for using a VM for a minimum time quanta δ is γ, the cost for this execution lies between Γ_{Min} and Γ_{Max},

$$\Gamma_{Min} = \sum_{j=1}^{l} \left\lceil \frac{\sum_{s=1}^{m} \widehat{\mathcal{M}}(i, s, j) \cdot \tau_i^s}{\delta} \right\rceil \cdot \gamma$$

$$\Gamma_{Max} = \sum_{s=1}^{m} \left(\left\lceil \frac{\tau_{Max}^s}{\delta} \right\rceil \cdot \mid \Upsilon_s \mid \cdot \gamma \right)$$

where Υ_s is the set of active VMs in superstep s, and $\mid \Upsilon_s \mid$ is the number of active VMs in that set,

$$\Upsilon_s = \{v_j \mid \widehat{\mathcal{M}}(i, s, j) = 1 \text{ for any } 1 \leq i \leq n\}$$

Γ_{Min} gives the minimum cost for all the VMs if each of their active times are summed over all supersteps and rounded up to the nearest δ. This does not account for the cost of turning off and restarting VMs that are inactive in intermediate supersteps, which would causes their billing to be rounded up each time. Whereas, Γ_{Max} gives the maximum cost required to run the application when each VM is billed in each superstep, independently, based on only the runtime of that superstep rounded up to the nearest δ. The *actual billing cost* for the application Γ falls between Γ_{Max} and Γ_{Min}. In practice, as we show, the actual value of Γ depends on the strategy used to select and activate VMs when making the placement decision.

Problem: Given a graph $\mathbb{G}$ and its partitions P_i, and the time function $\mathcal{A}$ for a graph application, the problem is to find a partition placement function $\mathcal{M}$, that maps the active partitions to VMs in each superstep, such that the actual cost Γ to execute the application on the graph is minimized while also minimizing the increase in the actual makespan T above the minimum makespan T_{Min}, i.e., *find $\mathcal{M}$ that minimizes Γ and $(T - T_{Min})$*.

As we shall discuss next, this can be modeled as an optimization problem, similar to bin packing, with the primary goal of reducing the cost of executing the application on elastic VMs through intelligent partition placement, and the secondary goal of reducing the increase in makespan above the theoretical minimum makespan.

5. PARTITION PLACEMENT STRATEGIES

5.1 Default Strategy

The default "flat" partition placement strategy used in GoFFish is to allocate as many cores (VMs) as the number of partitions, with each 1-core VM exclusively operating on a single partition for all supersteps of the application [4]. This placement function is given as:

$$\mathcal{M} : P_i \times s \to v_i$$

where each partition P_i is placed in its own VM v_i. The advantage of this is that a partition is processed in as fast as manner as possible on its own VM, given the partition-centric programming model, and its makespan matches T_{Min}. It is also trivial to solve and implement this strategy in practice, taking $\mathcal{O}(n)$ in time complexity for n partitions.

The actual billing cost for this strategy corresponds to the Γ_{Max}, and since all n VMs are kept active for the entire duration of the application, the cost can be simplified as:

$$\Gamma = n \cdot \left\lceil \frac{T_{Min}}{\delta} \right\rceil \cdot \gamma$$

5.2 Optimal and First Fit Decreasing Heuristics (OPT, FFD)

We reduce the given problem to a linear programming problem that gives the optimal number of VMs and the mapping of partitions to those VMs for each superstep, while

guaranteeing that the makespan does not increase beyond the T_{Min}. We set the necessary condition to retain the makespan at T_{Min} by requiring that each superstep takes only τ_{Max}^s, which is the most time taken by any single partition in a superstep if it has an exclusive VM. This is given as:

$$\max_{j=1}^{l} \left(\sum_{i=1}^{n} \widehat{\mathcal{M}}(i,s,j) \cdot \tau_i^s \right) = \tau_{Max}^s, \quad \forall 1 \le s \le m$$

If we treat each VM as a bin that has a time capacity of τ_{Max}^s in superstep s, we need to find a mapping $\mathcal{M}$ for all partitions (given by their timing τ_i^s) onto the *smallest number of bins in each superstep*. We can use linear programming to solve this problem.

Two simplifying assumptions are made here to assure optimality. One is that there is no cost to reassign partitions to VM between supersteps. This is unlikely in practice as data movement of partitions between VM does incur time cost, which affects both the makespan and the billing cost. But it is useful to consider this optimal solution as a baseline for comparison.

The linear programming problem can be solved using standard techniques [14], but it can be computationally costly to get the optimal solution. We term this strategy as Optimal (OPT). In addition, we use the heuristic First Fit Decreasing (FFD) algorithm to approximately solve this optimization problem with a lower complexity.

The pseudocode for FFD is given in Alg. 1, and it follows a greedy approach. At each superstep s, we start with no existing VMs and the partitions P_i are sorted in decreasing order of their execution times for that superstep, τ_i^s. In this sorted order, we test if each partition's execution time can fit in an existing VM. If so, the partition is mapped to that VM for this superstep, and the VM's capacity decreased by that partition's execution time. If no VMs can hold this partition, we create a new VM with capacity τ_{Max}^s and map the partition to this new VM, and decrement its capacity.

Time Complexity. OPT guarantees that the makespan does not increase more than T_{Min} while minimizing the number of active VMs per superstep, which is expected to reduce the billing cost. The tight theoretical bound for FFD relative to OPT on the number of active VMs is $(11/9 \cdot OPT + 6/9)$ [7], but is faster to calculate. Sorting the partitions requires $\mathcal{O}(n \log n)$ time and mapping each to a VM requires a linear scan of all VMs. For the ith partition, a maximum of i VMs can exist. So VM mapping takes:

$$\sum_{i=1}^{n} \log i = \mathcal{O}(\sum_{i=1}^{n} \log i) = \mathcal{O}(\log n!) = \mathcal{O}(n \log n)$$

and each superstep takes $\mathcal{O}(n \log n)$. The total complexity of FFD for m supersteps is $\mathcal{O}(m \times n \log n)$

Activation Strategy. Given the mapping function from the placement strategy, we propose a VM activation strategy to minimize the actual billing cost. This decides whether each VM, at the end of a superstep, can be left running or terminated for the next superstep. This is important since billing rounds up to the nearest δ, and stopping a VM for less than δ time and restarting it is costlier than retaining that VM idly for that duration, i.e., if l VMs are used in a superstep s, $l-1$ in superstep $s+1$, and again l VMs in $s+2$. If the duration of superstep $s+1 \le \delta$, it is cheaper to retain l VMs for all 3 supersteps.

Algorithm 1 First Fit Decreasing algorithm

1: **procedure** FIRSTFITDECREASING($P, \mathcal{A}, n, m$) ▷
   P is the set of n partitions. $\mathcal{A}$ is the time function that gives τ values. m is the number of supersteps.
2:     **for** $s \le m$ **do** ▷ *iterate over supersteps*
3:         $v[\ ] \leftarrow \varnothing \;; l = 0$ ▷ *init list of VM capacities*
4:         $p[\ ] \leftarrow$ SORTDESCENDING(P) ▷ *Sort by τ_i^s*
5:         **for** $i \le n$ **do** ▷ *iterate over each partition*
6:             assigned $\leftarrow false$
7:             **for each** $j \le l$ and assigned $= false$ **do**
   ▷ *does VM j have capacity for partition i?*
8:                 **if** $v[j] \ge \tau_{p[i]}^s$ **then**
9:                     $\mathcal{M}(p[i], s) \leftarrow j$ ▷ *map P_i to v_j*
10:                   $v[j] = v[j] - \tau_{p[i]}^s$ ▷ $\downarrow$ *capacity*
11:                   assigned $\leftarrow true$
12:                 **end if**
13:             **end for**
14:             **if** assigned $= false$ **then**
15:                 $v[++l] \leftarrow \tau_{Max}^s$ ▷ *create new VM*
16:                 $\mathcal{M}(p[i], s) \leftarrow l$ ▷ *do mapping*
17:                 $v[l] = v[l] - \tau_{p[i]}^s$ ▷ *reduce capacity*
18:             **end if**
19:         **end for** ▷ *mapping done for superstep s*
20:     **end for** ▷ *mapping done for all supersteps*
21:     **return** $\mathcal{M}$
22: **end procedure**

In our strategy, for each VM we do such a test, to see if the time remaining in a VM before the next δ increment is less than the time taken by the next superstep. If so, we retain that VM and reuse it for that superstep rather than create a new VM. Otherwise, if keeping that VM will cause it to go past the δ boundary, we terminate it.

Data Movement Cost. As a variation of OPT, we also evaluate these mapping if the data movement cost for moving a partition from one VM to another is considered. Called OPT-DM, this is more realistic when implementing OPT (or FFD) on the Cloud without special means to rapidly move or mount partitions between VMs. Here, the placement algorithm itself remains the same, but when calculating the billing cost, we include the time for data movement that causes the VM's billing to increase.

Specifically, we assume a shared persistent storage (like AWS S3 or Azure BLOB store) where partitions are moved to from VMs at the end of each superstep, and then before the next superstep starts, they are copied to the set of VMs that they are mapped to in that superstep. So each VM pays the time cost for moving data in and out of it at the start and end of a superstep, and is added to the billing cost.

5.3 Max Fit packing with Pinning (MF/P)

In this strategy, we avoid the data movement costs of OPT-DM by "pinning" a partition to a particular VM, and not changing the mapping after that. In other words, for a partition P_i whose $\tau_i^s > 0$:

$$\mathcal{M} : P_i \times s \to v_j \implies \mathcal{M} : P_i \times s' \to v_j, \;\; \forall s' > s$$

and hence, $\widehat{\mathcal{M}}(i,s,j) = 1 \implies \widehat{\mathcal{M}}(i,s',j) = 1, \;\; \forall s' > s$

Here, we use a strategy similar to FFD to greedily place an unpinned partition onto an existing VM with the *maximum* available capacity in a superstep, if possible, and if not possible, start a new VM. Once pinned, the partition

remains in that VM for the rest of the application. As a result, there are no data transfer costs, which additionally makes this simpler to implement.We term this as *Max Fit with Pinning (MF/P)*.

In FFD, the capacity of a VM in a superstep s was τ_{Max}^s. However, some of the VMs at the start of a superstep may already have partitions pinned on them, some or all of which may be active in this superstep. As a result, the makespan of this superstep depends both on the largest (unpinned) partition time, as well as the VM holding the largest cumulative pinned partition times.

Let λ_j^s be the load on a VM j in superstep s, defined as: $\lambda_j^s = \sum_{i=1}^n \widehat{\mathcal{M}}(i, s, j) \cdot \tau_i^s$, i.e., the cumulative time of all partitions mapped to that VM. We redefine τ_{Max}^s for MF/P at the start of superstep s as:

$$\tau_{Max}^s = \max\left(\ \max_{i=1}^n\left(\tau_i^s\right),\ \ \max_{j=1}^l\left(\lambda_j^s\right)\ \right)$$

Here, the first term within the outer *max* function gives the time taken by the largest partition in this superstep, while the second term gives the largest of the total times taken by all pinned partitions in a VM.

We skip the pseudocode for MF/P for brevity. Its key distinctions from FFD are that the partitions do not migrate between VM across supersteps, the initial capacity of a VM on a superstep is based on the largest partition in that superstep as well as the pinned partitions, and we pick the VM with the largest capacity for partition placement, rather than the first VM that has adequate capacity. This results in the following changes to Alg. 1. In Line 5, we only iterate through partitions that are not already pinned in a previous superstep. Those that are pinned retain their mapping. In Line 7, rather than iterate through each VM's capacity, we only test the VM with the largest available capacity. And lastly, in Line 15, we compute the value of τ_{Max}^s based on the updated function given above.

Note that this strategy does not guarantee a makespan that matches T_{Min}, as is obvious from the higher value of τ_{Max}^s. We retain the same VM activation strategy as used in OPT to decide whether to keep a VM active or terminate it at the end of every superstep. We retain the same VM activation strategy as used in OPT to decide whether to keep a VM active or terminate it at the end of every superstep.

Computational Complexity. The MF/P strategy maps each partition exactly once and keeps this mapping throught the runtime of the application. For mapping a partition, it finds the VM with the maximum capacity. If the partition fits in that VM, it maps the partition to that VM and otherwise spins a new VM. Finding the VM with the maximum capacity requires linear time. So the asymptotic time complexity of this algorithm across all supersteps is $\mathcal{O}(n^2)$.

5.4 First Fit Lookahead, with Pinning (LA/P)

One of the potential downsides of MF/P is that it decides to pinning a partition to a VM based only on the time taken by the partition in the current superstep. Since partitions once pinned do not migrate, we may end up with a placement that may be well-suited in the current superstep but lead to under-performance in future supersteps. This can result in several VM with pinned active partitions that are unbalanced in a superstep, which can increase the makespan. Since the *a priori* prediction model can provide partition timings for all supersteps, we can leverage this for

a more global planning across supersteps. To keep the problem tractable, we propose a variation of MF/P where the partition information for the current superstep and the next superstep are used to decide placement. Once decided, we continue to pin a partition to a VM. We term this strategy as *Lookahead with Pinning (LA/P)*.

The intuition here is that we first map unpinned partitions in a superstep, going from partitions the largest to the smallest execution times. Then, when considering VMs to map them to, we prefer VMs that have a higher capacity in the *next* superstep, rather than consider the first VM with adequate capacity (FFD) or the VM with the largest capacity (MF/P), in the current superstep.

We use two *rank* values to decide this placement:

Current Rank for each *active* partition P_i (i.e. $\tau_i^s > 0$) in superstep s is the index of that partition when all the active partitions are sorted in descending order of the execution times, τ_i^s.

Forward Rank for each *active* VM v_j in superstep s (i.e., v_j has some active partition in s pinned to it) is the index of that VM when all the VMs are sorted in ascending order of their load in the *next* superstep, λ_j^{s+1}.

Here again, we describe LA/P in terms of its distinctions from FFD's Alg. 1. In Line 5, we first sort the partitions based on their current rank and only iterate unpinned partitions. Those that are pinned retain their mapping. In Line 7, we sort the VMs based on their forward rank and then iterate through them. The VM mapping, however, is done only if the VM has adequate capacity in the current superstep. The forward rank is also recalculated after each new mapping in this superstep. And lastly, in Line 15, similar to MF/P, we compute the value of τ_{Max}^s based on the updated function that considers pinning.

In LA/P too, we do not guarantee that the makespan does not grow beyond T_{Min}, and we use the same VM activation strategy as FFD.

Computational Complexity. For each superstep, the algorithm sorts n partitions based on their execution time to calculate their current rank, taking $\mathcal{O}(n \log n)$ time. For each of the n partitions to be mapped, the VMs are sorted based on their forward rank, that takes $\mathcal{O}(n \log n)$ time. This gives a total complexity of $\mathcal{O}(n \log n + n^2 \log n))$, which is dominated by the latter term.

6. EVALUATION

We evaluate the different placement strategies for performing SSSP (BFS) over undirected graphs on elastic VMs. We initially run the graph algorithm over the different graphs on a commodity cluster to get their partition timings ($\mathcal{A}$), and use this as input for simulating the application execution using the strategies.

6.1 Setup and Datasets

We use 4 graphs that are partitioned into 8 or 40 partitions, as noted: LIVJ/8P [2], USRN/8P [3], and ORKT/40P [4]. For the default strategy, we run the subgraph-centric SSSP/-BFS application using GoFFish on a 24-node commodity cluster connected by Gigabit Ethernet, and each partition is allocated one AMD 3380 2.3 GHz core and 4 GB RAM, and

---

[2]http://snap.stanford.edu/data/soc-LiveJournal1.html
[3]http://www.dis.uniroma1.it/challenge9/download.shtml
[4]http://snap.stanford.edu/data/com-Orkut.html

Table 1: Datasets used

Graph (Name/Part.s)	$|V|$	$|E|$	Dia.
LiveJournal (LIVJ/8P)	4.847M	68.993M	16
USA Road (USRN/8P)	23.947M	58.333M	$6,262$
Orkut (ORKT/40P)	3.072M	234.370M	9

shares a 256 GB SSD on that node. We use CentOS 7 and JDK v7. The graphs are partitioned using METIS with a default load factor of 1.03 for vertex-balanced partitioning.

We use GoFFish's logging framework to get the compute time for each subgraph in a superstep, and calculate the time for a partition as the sum all its subgraph times in that superstep. This is used as the partition execution time ($\mathcal{A}$) passed to the strategies.

The placement strategies are scripted in Python v2.7, and take as input $\mathcal{A} : P_i \times s \to \tau_i^s$, along with the number of partitions n and the supersteps m. The output generated by the algorithms is the partition placement function $\mathcal{M} : P_i \times s \to v_j$. From this mapping, we calculate the actual makespan, billing cost, and other metrics described next.

6.2 Plots and Metrics

Makespan: Makespan is the total time taken by the graph application, as calculated for the mapping returned by each strategy. It is calculated as the sum over all supersteps of the time taken by the slowest VM in that superstep. This is plotted for each graph in Figs. 3a–3c.

Cost in Core-Minutes: While makespan gives the actual runtime of the application, the actual billing cost depends on how long each VM was active, and when they were turned on and off, as decided both by the mapping and the VM activation strategy. We use the mapping and activation information to calculate the total core-mins for which the VMs will be billed, using a 1-minute billing cycle, and rounding up the VM cost to the nearest minute each time it is turned off. This duplicates the actual billing logic used by IaaS Cloud providers like Azure. The actual monetary cost is just a multiplication of these core-mins by the per-minute rate, which depends on the data center, VM size, etc. This is plotted for each graph in Figs. 3d–3f.

Under-Utilization: In the BSP exection model used by component-centric frameworks, VMs remain active for the entire superstep even if some of them have completed processing their partitions. We capture the wasted time due to the slowest VM across all supersteps as under-utilization. It is given as the difference between the core-minutes for which VM were provisioned and the core-minutes for which they actually processed partitions, i.e.,

$$\sum_{s=1}^{m} \left(\tau_{Max}^s \cdot | \Upsilon_s | \right) - \sum_{s=1}^{m} \sum_{i=1}^{n} \left(\tau_i^s \right)$$

This is plotted for each graph in Figs. 3g–3i.

Core-Seconds: The core-mins cost uses a VM-minute granularity billing model provided by cloud providers. We attempt to highlight the relative benefits of the strategies if such costing granularity constraints were not present, and calculate the core-seconds for which VMs were provisioned by each strategy. This is given by,

$$\sum_{s=1}^{m} \left(\tau_{Max}^s \cdot | \Upsilon_s | \right)$$

This is plotted for each graph in Figs. 3j–3l.

6.3 Analysis

The time, cost and utilization values for OPT and FFD are identical in all cases. So the *FFD algo is a good enough approximation* for OPT while only taking 1 sec to run the placement strategy for the largest graph ORKT/40P, compared to 13 secs taken to calculate using OPT. Hence, FFD can be chosen over OPT when performing online scheduling.

Further, the makespan for OPT and FFD are the same as the default strategy in all cases and equals T_{Min}, the smallest possible makespan. Both these algorithms are successfully able to provide *adequate VMs for the required computation on the active partitions* to allow them to complete without delay. Since they do not consider data movement costs in a superstep, the only time spent is on the computation of the active partitions by an exclusive VM. Hence, the secondary objective of not increasing the makespan above T_{Min} is also achieved. However, in practice, the partitions will need to be moved between VMs across supersteps depending on the placement mapping generated, so OPT-DM is more plausible.

For **LIVJ/8P**, the makespan for MF/P is modestly higher than the default, at 27 secs against 21 secs, and is the same as LA/P (Fig. 3a). We observed here that the VM with maximum capacity for MF/P and the VM with highest forward rank for LA/P turned out to be same, causing the mapping performed by both the algorithms to be identical.

We also see that the core-mins for OPT, FFD, MF/P and LA/P are all comparable, taking 6-8 core-mins (Fig. 3d). This cost for MF/P and LA/P is smaller than the 8 core-mins taken by the default strategy, saving 25% in cost.

Interestingly, MF/P and LA/P cost lesser than OPT and FFD. This is because OPT and FFD guarantee that the makespan will not rise beyond T_{Min}, and as a result allocate adequate VMs to meet this goal. But MF/P and LA/P do not allow partition movement once they are pinned, and in this case, all partitions are pinned at the end of the second superstep. As a result, the number of active VMs at the second superstep (6 VMs) is retained for the rest of the supersteps, and this value is smaller than the peak number of VMs used by OPT (8 VMs) which results in a higher cost.

However, we also see from the core-secs used (Fig. 3j) that OPT and FFD use fewer VM cycles than MF/P and LA/P (112 core-secs vs. 142 core-secs), even though this does not reflect in a reduced cost for the VMs used due to the core-min billing granularity.

When we consider the data movement costs for OPT-DM, it takes a much longer time to complete compared to the default and other strategies, taking almost 7× longer and also costing more due to this added time. A whole 11 core-mins of this is under-utilized (Fig. 3g), due to data movement when the CPU is mostly idle. It can be seen that in the absence of this wastage, the cost would be comparable to OPT and FFD.

For **USRN/8P**, we see that the makespan for MF/P and LA/P are 6% slower than OPT (Fig. 3b). In MF/P and LA/P strategies, all the partitions are pinned to VMs by the third superstep itself. This causes the makespan to increase due to multiple active partitions being on the same VM in future supersteps, and hence increasing the sequentially processing time of those partitions in a superstep. We also see that the cost for LA/P is same as that of MF/P,
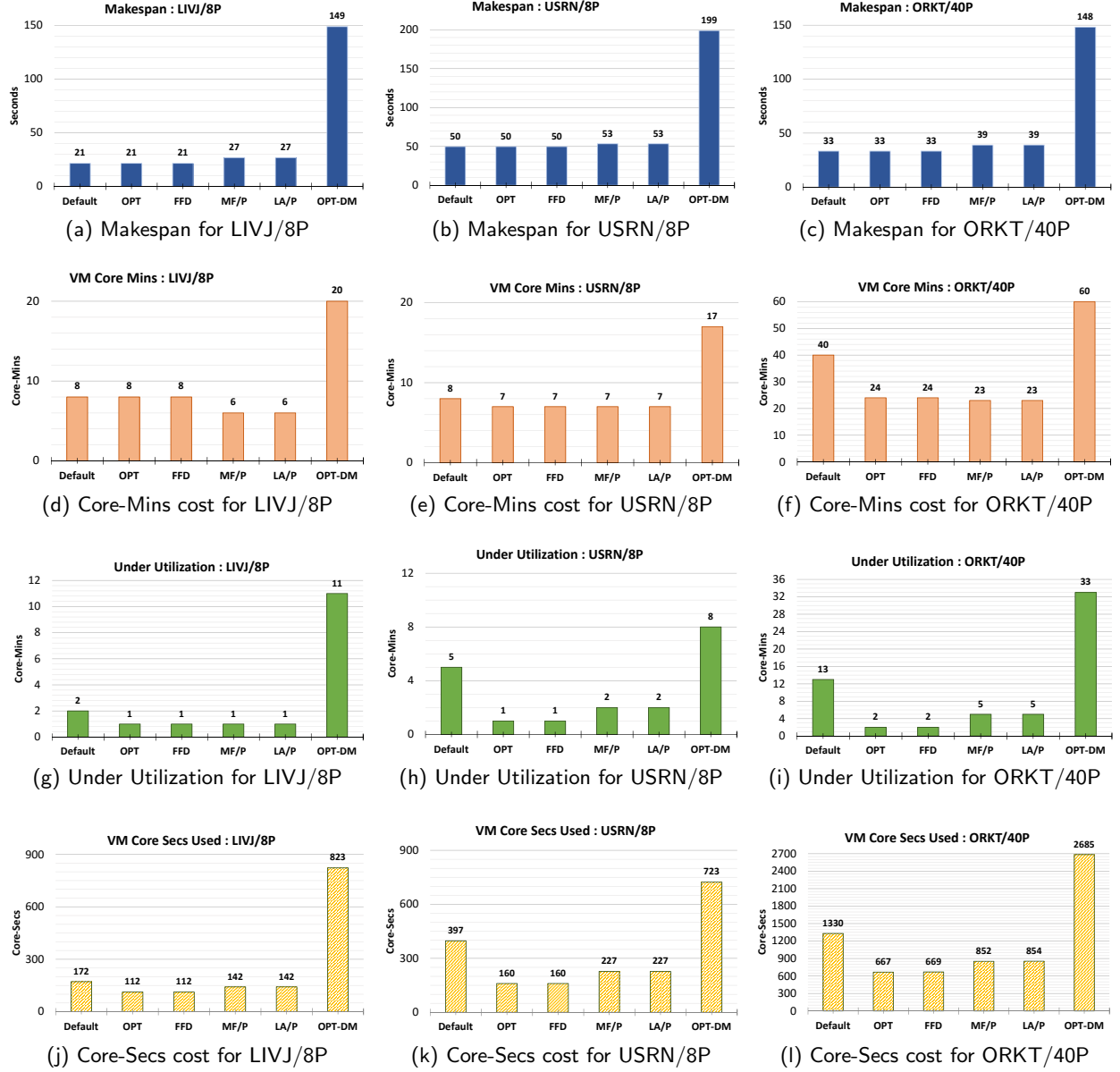
Figure 3: Performance metrics of the three graphs for different placement strategies.

and those of OPT and FFD due to granualarity of billing.

The core-secs used by MF/P and LA/P at 227 core-secs are much higher than 160 core-secs used by OPT and FFD. But this is not reflected in the core-mins metrics as the MF/P and LA/P algorithms retain a fixed number of VMs after all partitions get pinned, while the OPT and FFD algorithms end up using more VMs to meet the T_{Min} makespan constraint. For the same reason, the under-utilization, for LA/P and MF/P is more compared to the OPT and FFD.

ORKT/40P is a large graph and takes 33 secs to run using the default strategy (Fig. 3c), and with the cost being 40 core-mins (Fig. 3f). We see that OPT and FFD are able to complete at a 40% cheaper cost than the default, saving 16 core-mins. Here again, LA/P and MF/P are modestly cheaper than the default by 42% while having a small increa-

se in the makespan relative to T_{Min}. The core-secs used by LA/P of 854 core-secs approaches OPT's 667 core-secs, and is much smaller than the default's 1,330 core-secs (Fig. 3l). As a result, the under-utilization too is low (Fig. 3f), with just 5 core-min wasted as compared to the default's 13 wasted core-mins.

OPT-DM is worse than the default, both in makespan as well as in cost. Given that ORKT is a large graph and each partition has a size of ~ 100 MB, the data movement cost for each superstep adds up and causes an increase in both makespan and cost.

In summary, we see that both MF/P and LA/P are comparable in terms of results, and offer practical strategies to leverage elasticity of VM for partition placement. They are 6-29% slower than the default strategy, but 12-42% cheaper

and consistently use much fewer core-secs to execute than the default. MF/P may be preferable for its simplicity.

The *OPT* and *FFD* strategies guarantee a makespan that matches T_{Min} and are the same or cheaper than the default on cost. They also out-perform all other strategies on core-secs and under-utilization. However, they may not be practical to implement because the data movement costs in unlikely to be ignored. For graphs that are compactly stored, or when if VMs mount the same shared network drive (e.g., AWS's Elastic Block Store (EBS) volumes), these algorithms can be feasible in practice. But if naïvely transferring data betweens VMs at the end of each superstep, *OPT-DM* is much worse than all other strategies.

7. CONCLUSIONS & FUTURE WORK

In this paper, we have proposed to decouple the partitioning and placement strategies for component-centric graph frameworks to allow flexibility in scheduling them onto elastic VMs that can prevent over-allocation of resources. We have designed several partition placement strategies for graph applications whose runtime behavior can be modeled *a priori*, motivated by our earlier work on meta-graph that can help with coarse-grained static analysis.

These strategies that include OPT and FFD that are optimal and heuristic formalizations that guarantee the theoretical minimum makespan, as achieved by the default model of over-allocating one VM per partition, while reducing the number of VM used. They also include greedy strategies, MF/P and LA/P, that pin partitions to VMs to avoid data movement, using information on partition timings and VM loads at the current or subsequent supersteps.

The results show that the billing costs, based on realistic IaaS Cloud models, are reduced with marginal increase in makespan for the proposed strategies when evaluated on three different real-world graphs. We also see a more significant improvement in under-utilization and core-secs used, which are orthogonal to the billing granularity.

Leveraging elasticity for distributed graph processing is poorly explored in literature and there are several promising avenues for future work. We propose to examine other non-stationary graph algorithms such as betweenness-centrality and independent-set. This includes the ability to model them using meta-graphs and their benefits from elastic placement. There is scope for using look-ahead heuristics *without pinning* to better leverage the additional knowledge. The optimization goal itself can be modulated to explore the trade-off between makespan and cost.

The minute-granularity is still too coarse for the graphs and application considered, and the strategies may show improved behavior on larger graphs where the makespan is longer, and the core-mins cost much higher. Considering containers such as Docker is also an alternative light-weight "virtualization" that can be started and shutdown rapidly, instead of heavy-weight VMs.

We also plan to examine the impact of the *a priori* predictions being inaccurate – in our evaluation, we assume perfect knowledge of partition timings but that is unlikely even using the meta-graph model. Here, the static placement strategies proposed here will need to be complemented with dynamic runtime information to update the placements.

8. REFERENCES

[1] Apache giraph. `http://giraph.apache.org`.

[2] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[3] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[4] N. Choudhury, R. Dindokar, A. Dixit, and Y. Simmhan. Partitioning strategies for load balancing subgraph-centric distributed graph processing. *arXiv preprint arXiv:1508.04265*, 2015.

[5] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[6] R. Dindokar, N. Choudhury, and Y. Simmhan. Analysis of subgraph-centric distributed shortest path algorithm. In *The 4th International Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics*. IEEE, 2015.

[7] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd (i)âĽ'd' 11/9opt (i)+ 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer, 2007.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[9] A. Guerrieri and A. Montresor. Distributed edge partitioning for graph processing. *arXiv preprint arXiv:1403.6270*, 2014.

[10] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[12] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[14] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[15] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and analysis of bsp graph processing models on public clouds. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 203–214. IEEE, 2013.

[16] S. Salihoglu and J. Widom. Gps: A graph processing

system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[17] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. In *40th International Conference on Very Large Data Bases*. Stanford InfoLab, September 2014.

[18] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 553–564. IEEE, 2013.

[19] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

[20] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par 2014 Parallel Processing*, pages 451–462. Springer, 2014.

[21] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[22] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14), 2014.

[23] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[24] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2012.