Stratified Sampling for Even Workload Partitioning Applied to Single Source Shortest Path Algorithm

Jeeva Paudel
Dept. of Computing Science
University of Alberta
jeeva@cs.ualberta.ca

Levi H. S. Lelis
Departamento de Informática
Univ. Federal de Viçosa
levi.lelis@ufv.br

José Nelson Amaral
Dept. of Computing Science
University of Alberta
amaral@cs.ualberta.ca

ABSTRACT

An efficient implementation of large graph processing algorithms on distributed-memory machines requires a balanced partitioning of the graph across the machines. In a previous paper we presented an algorithm, named Workload Partitioning and Scheduling (`WPS`), that uses domain-specific knowledge to guide a sampling procedure in large implicitly-defined graphs. `WPS`'s sampling procedure is used for partitioning the workload into parts of similar size which is then distributed amongst different machines. This article extends that earlier study and presents an investigation of the parallel and distributed implementation of Meyer's Δ-Stepping algorithm for solving the Single-Source Shortest Path (SSSP) problem for directed graphs. Our implementation leverages the `WPS` algorithm for evenly distributing the workload involved in processing the vertices of the input graph across distributed-memory machines. In contrast with the previous study, which focussed on implicitly-defined graphs, this work demonstrates that `WPS` is also equally applicable on explicitly-defined graphs.

Empirical evidence shows that applying `WPS` to Meyer's SSSP algorithm yields significant performance benefits.

1. INTRODUCTION

Applications in various domains such as combinatorial optimization, graph theory, and state-space exploration exhibit irregular data and task parallelism. These applications tend to generate substantial amounts of work at runtime. A large body of recent work has focussed on exploiting irregular and latent parallelism inherent in such applications with considerable success. Kulkarni *et al.* show that many applications exhibit a generalized form of data-parallelism called amorphous data-parallelism [9], despite the application's irregular nature. Similarly, another related problem of evenly distributing the work in irregular applications across distributed-memory nodes has also been studied extensively. Domain experts have implemented highly optimized parallel programs and libraries, such as the parallel BGL[4], Pregel [14] and its open-source counterpart Giraph, distributed GraphLab [12], Green-Marl [5], Δ-Stepping [15, 13], for processing graphs on large distributed-memory machines. These efforts are highly tailored towards domain-specific needs. As such, it is difficult to extract broadly applicable mechanisms from these implementations.

In a previous article we proposed a structured approach, called Workload Partitioning and Scheduling (`WPS`), for predicting the workload in irregular applications [17]. Our previous work was primarily aimed at algorithms where the work items, such as vertices in graphs, may be generated implicitly and may not be available *a priori*. An important contribution from that study is the following observation: Many work items in applications tend to show a large degree of similarity. The similarity depends on the application domain. For example, two states in the 4×4 sliding tile puzzle problem may be considered similar if they both entail the same number of moves to reach a goal state. As such, domain-specific knowledge offers good heuristics to define the similarity of work items in applications. Such heuristics can be used to selectively sample only a small number of work items. Workload analysis of the sampled work items can then be used as a good predictor of the entire workload generated by an application. This estimate of the total workload generated by an application, either implicitly or explicitly, can then be leveraged by a runtime system to evenly distribute the workload. Our previous work illustrates this observation largely in terms of tree search algorithms.

This work extends that earlier study and presents an investigation of the parallel and distributed implementation of Meyer's Δ-Stepping algorithm for solving the Single Source Shortest Path (SSSP) problem for directed graphs. Our implementation leverages the `WPS` algorithm for evenly distributing the workload involved in processing the vertices of the input graph across distributed-memory machines. In contrast with Paudel et al.'s study which focused on implicitly-defined trees, this work demonstrates that `WPS` is also effective on explicitly-defined graphs. As we show later in this paper, the distinction between trees and graphs is important in the context of workload distribution due to `StraSa`, the sampling algorithm employed by `WPS`. Although `StraSa` is able to produce near-perfect estimates of the workload when the work items are represented as trees [11, 17], it often fails to produce good estimates when the work items are represented as graphs [10]. Our empirical results show that despite its inability of producing good estimates in graph structures, `WPS` is effective in partitioning the workload of SSSP problems.

2. PROBLEM DEFINITION

In this paper $G = (V, E)$ is a explicitly-defined directed graph representing an application problem. Let $\Gamma(n^*)$ with $n^* \in V$ be a *Work-Item Set* (*WIS*) representing the set of items processed by an algorithm while finding the shortest paths from n^* to all other vertices in the graph. In this paper we will write *WIS* instead of $\Gamma(n^*)$ whenever n^* is clear from context. For each $n \in V$, *child*(n) is the set of items generated when n is processed: *child*$(n) = \{n_i | (n, n_i) \in E\}$.

Given M processing nodes in a computer cluster and a *WIS*, the workload partitioning problem consists in partitioning the items in *WIS* into M parts $W_1, W_2, \cdots, W_M$ of similar size. Our goal is to minimize $\sum_{i,j \in \{1,\cdots,M\}} |W_i| - |W_j|$, where $|W_i|$ is the size of W_i. We assume that all items in *WIS* take approximately the same amount of time to be processed.

Our current work is similar to our original work in that we also aim at partitioning the workload of graph-search algorithms. However, the current approach differs from the earlier work in two important ways. First, we apply WPS to explicitly-defined graphs, which is in contrast with the implicitly-defined graphs used in our earlier work. Second, the work items of the problems, where WPS was originally applied, could be represented by a tree. The tree representation allowed WPS to sample the work items directly. By contrast, as we explain later in the paper, we do not have access to the *WIS* during WPS's sampling. We approximate the *WIS* by assuming it has a tree structure. Although our assumption does not hold in practice, our empirical results show that the WPS approach can also be effective when applied to SSSP.

3. STRATIFIED SAMPLING (StraSa)

A key part of the WPS algorithm is the sampling it performs for partitioning the workload amongst different processing nodes. In this section we explain Stratified Sampling (StraSa), the sampling algorithm WPS employs.

Knuth [8] presents a technique to estimate the size of the tree expanded by a search algorithm such as chronological backtracking. His technique repeatedly performs a random walk from the root of the tree. When all branches have the same structure, a random walk down one branch is enough to estimate the size of the entire tree. Knuth observed that his technique was not effective when the tree is imbalanced. Chen [1] addressed this problem by stratifying the search tree to reduce the variance of the sampling process. This paper refers to Chen's technique as *Stratified Sampling* (StraSa). WPS uses StraSa to estimate the *WIS* size and, based on such an estimation, it finds a partition of the items in the *WIS*.

Definition 1 (Source's Tree). *Let $S(n^*) = (V', E')$ be a tree rooted at source vertex n^*. $S(n^*)$ is constructed following G such that $V \subseteq V'$ and for any $n_1, n_2 \in V'$ we have that $(n_1, n_2) \in E'$ iff $(n_1, n_2) \in E$. We write S instead of $S(n^*)$ whenever n^* is clear from context.*

The tree $S(n^*)$ can be constructed from G by traversing G in a depth-first search ordering starting from n^* and bounded by the largest shortest distance between n^* and any other vertex in G.

Definition 2 (Stratification). *$T = \{t_1, \ldots, t_n\}$ is a stratification for S if it is a disjoint partitioning of V'. If $n \in V'$, $t_i \in T$ and $n \in t_i$, then $T(n) = t_i$ states that the stratum of n is t_i.*

Algorithm 1 StraSa, a single probe

```
Require: root $n^*$ of a tree and a stratification $T$
Ensure: a sampled tree $ST$ represented by an array of sets
    $A$, where $A[i]$ is the set of pairs $\langle n, w\rangle$ for the vertices $n$
    at level $i$, and an array of sets $C$, where $C[i]$ is the set
    of vertices $n$ at level $i$ but not expanded.
 1: $A[0] \leftarrow \{\langle n^*, 1\rangle\}$
 2: $i \leftarrow 0$
 3: while stopping condition is false do
 4:   for each element $\langle n, w\rangle$ in $A[i]$ do
 5:     for each child $\hat{n}$ of $n$ do
 6:       if $A[i+1]$ contains an element $\langle n', w'\rangle$ with
          $T(n') = T(\hat{n})$ then
 7:         $w' \leftarrow w' + w$
 8:         with probability $w/w'$, replace $\langle n', w'\rangle$ in $A[i+1]$ by $\langle \hat{n}, w'\rangle$ and insert $n'$ in $C[i+1]$; insert $\hat{n}$
            in $C[i+1]$ otherwise
 9:       else
10:         insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$
11:       end if
12:     end for
13:   end for
14:   $i \leftarrow i + 1$
15: end while
```

StraSa is a general approach for approximating any function of the form

$$\varphi(n^*) = \sum_{n \in S(n^*)} z(n) \tag{1}$$

where z is any function assigning a numerical value to a vertex. $\varphi(n^*)$ represents a numerical property of the search tree rooted at n^*. For instance, if $z(n) = 1$ for all $n \in S(n^*)$, then $\varphi(n^*)$ is the size of the tree. As we explain later, in the paper we are interested in estimating the number of outgoing edges in a given tree. Thus, in our case $z(n)$ returns the number of outgoing edges from vertex n.

Instead of traversing the entire tree and summing all z-values, StraSa assumes that subtrees rooted at vertices of the same stratum have equal values of φ and thus only one vertex of each stratum, chosen randomly, is processed. This selective expansion is the key to StraSa's efficiency as it explores only a fraction of the tree when estimating $\varphi(n^*)$.

Given a vertex n^* and a stratification T, StraSa estimates $\varphi(n^*)$ as follows. First, it samples the tree rooted at n^* and returns a set A of *representative-weight* pairs, with one such pair for every unique stratum seen during sampling. Given a pair $\langle n, w\rangle \in A$ for stratum $t \in T$, n is the unique vertex of stratum t that was expanded during sampling and w is an estimate of the number of vertices of stratum t in $S(n^*)$. $\varphi(n^*)$ is then approximated by $\hat{\varphi}(n^*)$, defined as

$$\hat{\varphi}(n^*) = \sum_{\langle n, w\rangle \in A} w \cdot z(n) \,. \tag{2}$$

Algorithm 1 shows StraSa in detail. The set A is divided into subsets, one for every layer in the search tree; $A[i]$ is the set of representative-weight pairs for the strata encountered at level i. In StraSa, the strata must be partially ordered such that an vertex's stratum is strictly greater than that of its parent in the tree. Chen suggests that this constraint can always be guaranteed by adding the depth of an item in the tree to the stratification and then sorting the strata lexicographically. In this implementation of `StraSa` the depth of exploration is implicitly added to the stratification: strata at each tree level are treated separately by the division of A into the $A[i]$. If the same stratum occurs on different levels, the occurrences are treated as though they were of different stratum.

$A[0]$ is initialized to contain only the root of the tree to be probed, with weight 1 (line 1). In each iteration (lines 4 – 10), all the items from $A[i]$ are expanded to get representative items for $A[i+1]$ as follows. Every item in $A[i]$ is processed and its children are considered for inclusion in $A[i+1]$. If a child $\hat{n}$ has a stratum t that is already represented in $A[i+1]$ by another vertex n', then a *merge* action on $\hat{n}$ and n' is performed. A merge action increases the weight in the corresponding representative-weight pair of stratum t by the weight $w(n)$ of $\hat{n}$'s parent n (from level i) since there were $w(n)$ items at level i that are assumed to have children of stratum t at level $i+1$. $\hat{n}$ will replace the n' according to the probability shown in line 8. Chen [1] proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, `StraSa` expands the items in $A[i+1]$. This process continues until it reaches a level i^* where $A[i^*]$ is empty.

One run of the `StraSa` algorithm is called a *probe.* $\hat{\varphi}^{(p)}(n^*)$ is the p-th probing result of `StraSa`. `StraSa` is unbiased, i.e., the average of the $\hat{\varphi}(n^*)$-values converges to $\varphi(n^*)$ in the limit as the number of probes goes to infinity. Chen [1] states the following theorem:

THEOREM 1. *Given a stratification T and a set of p independent probes $\hat{\varphi}^{(1)}(n^*), \cdots, \hat{\varphi}^{(p)}(n^*)$ from a tree $S(n^*)$, $\frac{1}{p}\sum_{j=1}^{p}\hat{\varphi}^{(j)}(n^*)$ converges to $\varphi(S)$ as p grows large.*

Each `StraSa` probe outputs a subtree of the tree called *sampled tree* (ST). In contrast with Chen's version of `StraSa`, our version of the algorithm also outputs an array of sets C containing the vertices encountered during sampling which were not processed. C is organized by levels, e.g., $C[i]$ is the set of vertices `StraSa` encountered but did not expand at level i of $S(n^*)$. `WPS` uses C to evenly divide the workload among different processing nodes, as we now explain.

4. WORKLOAD PARTITIONING AND SCHEDULING ALGORITHM (WPS)

This section provides an explanation of the Workload Partitioning and Scheduling Algorithm (`WPS`). In Section 6 we explain how `WPS` can be applied to the SSSP problem.

`WPS` operates in four phases: sampling, estimating, partitioning, and distributing. Algorithm 2 shows a high-level description of `WPS`.

4.1 Sampling

The working items of the application problems Paudel et al. [17] originally applied `WPS` to are represented by the vertices in S. Due to this property, by sampling S, `WPS` is effectively sampling the working items, and a balanced partitioning of S represents a balanced partitioning of the items.

`WPS` employs `StraSa` on S to selectively process only one among several items of the same stratum at each level of the tree. This phase produces a sampled tree ST and a set C of items that were encountered but not expanded. The subtree ST is used to estimate the size of subtree rooted at items of different strata (see Section 4.2 below), while the items in C are partitioned amongst the available processing nodes according to the size of the subtrees provided by ST (see Section 4.3 below).

Algorithm 2 Workload Partitioning and Scheduling

```
Require: starting vertex $n^*$ of $S$ and a stratification $T$
Ensure: solution for the problem represented by $n^*$
 1: $[A, C] \leftarrow StraSa(n^*, T)$
 2: $\chi \leftarrow ComputePropertySubtree(A, T)$
 3: $\{W_1, W_2, \cdots, W_M\} \leftarrow BLDM(\chi, C)$ // see [16]
 4: for $i \in \{1, \cdots, M\}$ do
 5:     asynchronously copy $W_i$ to node $i$
 6: end for
```

Algorithm 3 Compute Property Subtree

```
Require: sampled tree $ST$ represented by $A$ and stratification $T$
Ensure: a collection $\chi$ of the estimated subtree sizes $Y_t^i$
        for each level $i$ and stratum $t$ in $A$.
 1: $\chi \leftarrow \{\}$
 2: for $i \leftarrow$ tree depth to 1 do
 3:     for each item $n$ in $A[i]$ do
 4:         $Y_{T(n)}^i \leftarrow 1$
 5:         for each child $n''$ of $n$ in the tree do
 6:             $Y_{T(n)}^i \leftarrow Y_{T(n)}^i + Y_{T(n'')}^{i+1}$
 7:         end for
 8:         insert $Y_{T(n)}^i$ in $\chi$
 9:     end for
10:     $i \leftarrow i - 1$
11: end for
```

4.2 Estimating

In this phase, `WPS` traverses the ST bottom up and uses dynamic programming to compute the estimated size of the subtrees rooted at each item $n \in ST$. This procedure is shown in Algorithm 3. Such a procedure outputs a collection χ of the estimated subtree sizes Y_t^i for each tree level i and stratum t in A. χ is then used in Partitioning, the next phase of the algorithm.

4.3 Partitioning

In the Sampling phase, `WPS` processes a small subset of the items in the tree through `StraSa`. In this phase `WPS` partitions the remaining items in the tree—the items not processed by `StraSa`—into M groups, where M is the number of processing nodes available. The items not processed by `StraSa` are the items in C as well as the items reachable from the items in C.

`StraSa` ensures that for each item n' in $C[i]$ there is a unique item n in $A[i]$ with $T(n) = T(n')$. Moreover, given Chen's assumption that items of the same stratum root subtrees of the same size, $Y_{T(n)}^i$ in χ is an estimate of the number of items in the subtree rooted at n' (number of items reachable from n'). Thus, at this point, the problem of evenly partitioning the workload reduces to the NP-Hard multi-way number partitioning problem [2]: The algorithm must partition the items n' in C into M parts $W_1, W_2, \cdots, W_M$ such that the sum of the $Y_{T(n')}^i$ values in each part W_j and W_k

with $j, k \in \{1, 2, \cdots, M\}$ are as similar as possible to each other. WPS employs the Balanced Largest-First Differencing Method (BLDM) [16] to compute an approximated solution to the number partitioning problem. BLDM is a widely used and effective algorithm that performs k-way partitioning for $k \geq 2$ in $\mathcal{O}(n \log n)$ time.

WPS uses the estimated number of items rooted at each given item as a workload metric for even distribution. WPS assumes that the time required to process an item is constant throughout the tree—an assumption that also holds in all the applications studied in the current paper. WPS could be easily adapted to use other workload metrics as well. For example, in applications where work items have different processing times, StraSa could estimate the total processing time of subtrees as opposed to estimating the size of the subtrees. Then, the Balanced Largest-First Differencing Method (BLDM) [16] is used to partition the items not processed by StraSa into parts of similar processing time.

4.4 Distributing

In this phase, WPS stores one subset W_1 in local memory for processing in the current processing node and distributes the remaining $W_{j|j=2..M}$ subsets of items to the $M - 1$ remaining processing nodes.

5. SINGLE SOURCE SHORTEST PATH PROBLEM

The shortest path problem is an important combinatorial optimization problem with applications in diverse domains such as web mapping, game theory, operations research, and engineering. Consider a directed graph $G = (V, E)$, with $|E| = m$, and $|V| = n$. Let s be the *source* vertex (also called the *root* vertex) of the graph, and l be a function assigning a non-negative real-valued weight to each edge of G. The edge weights represent the distance between the adjacent vertices. The single-source shortest-path (SSSP) problem computes the distance $d^*(v)$ for every vertex v in the graph such that $d^*(v)$ equals the distance from the source vertex to v along the shortest path.

5.1 Sequential Algorithm: Dijkstra's Algorithm

A well-known sequential algorithm for computing SSSP is Dijkstra's algorithm. The algorithm maintains a *tentative distance* $d(v)$, for each vertex $v \in V$. The value $d(v)$ always represents an upper bound on the actual shortest distance $d^*(v)$. Initially, the tentative distance of the root vertex is initialized to 0 and that of all other vertices to ∞. All vertices v whose $d(v) \neq d^*(v)$ are called *unsettled* vertices.

The algorithm begins by considering all the vertices as unsettled and proceeds in multiple iterations. In each iteration, the algorithm selects the vertex u that has the minimum tentative distance. Initially, only the *source* vertex is settled because it has the minimum tentative distance in the entire graph *i.e.,* $d(u) = d^*(u) = 0$. Then, for each neighbour v of u in the edge $e = \langle u, v\rangle$, the algorithm modifies the tentative distance by performing an operation called *edge relaxation.* Reducing the tentative distance of u can possibly reduce the tentative distance of its neighbour as well. Given an edge $e = \langle u, v\rangle$, the operation $\texttt{Relax}(u, v)$ is defined as

$$d(v) \leftarrow \min\{d(v), d(u) + w(\langle u, v\rangle)\},$$

where w represents the edge weight.

For each edge (u, v) encountered during the relaxation process, the algorithm adds a pair $\langle v, distance\rangle$ to the workset, where $distance = d(u) + weight(u, v)$. The workset is ordered by the *distance* of each pair. Initially, the tree contains only the source vertex as the root, and the workset contains tuples corresponding to the neighbours of the source vertex. The algorithm repeatedly removes the pair with the minimum distance from the workset and, if the corresponding vertex has not already been visited, updates the vertex's distance and adds $\langle neighbour, distance\rangle$ pairs to the workset for each of its neighbours. As the algorithm proceeds, $d(v)$ matches the actual shortest distance $d^*(v)$. A vertex v is said to be *settled* if $d(v) = d^*(v)$. The algorithm terminates when there are no more unsettled vertices in the graph.

Dijkstra's algorithm processes the vertices in a fixed priority order. Therefore, Dijkstra's algorithm is inherently sequential and less amenable to parallelization.

5.2 Parallelizing SSSP: Bellman-Ford Algorithm

Dijkstra's algorithm processes one vertex in any iteration. The key to parallelizing SSSP lies in processing the distance of multiple vertices in parallel. Multiple vertices in the graph can be processed in parallel if the shortest distance computed for each vertex does not depend on the order of processing of the vertices. To derive the unordered algorithm, we note that shortest distance from the source to each vertex in the graph can be computed using the following fixpoint system.

Initialization:

$$d(root) = 0; d(k) = \infty, \forall k \text{ other than root}$$

Fixpoint computation:

$$d(v) \leftarrow \min\{d(v), (d(u) + w(\langle u, v\rangle))\},$$

where $\forall u \in$ incoming neighbours of v, and $w(u, v)$ is the edge weight, i.e., the distance between u and v. This observation yields an unordered algorithm for SSSP that is essentially the Bellman-Ford algorithm, shown in Algorithm 4.

A brief outline of how the Bellman-Ford algorithm operates is as follows: The algorithm proceeds in multiple iterations. In each iteration, a vertex u is declared to be active if its tentative distance $d(u)$ changed in the previous iteration. All active vertices are added to the workset. For each active vertex u, the algorithm considers all the edges $e = \langle u, v\rangle$ and performs $\texttt{Relax}(u, v)$. This fix-point iterative algorithm may update a vertex in the graph multiple times, but the vertex's distance is guaranteed to decrease monotonically to the correct value.

Initially, the only active vertex in the graph is the source vertex. In every step, the algorithm removes a vertex from the workset and tries to reduce the distance of each outgoing neighbour. If the distance of a neighbour is reduced, this neighbour is added to the workset. Hence, the workset always contains the vertices whose neighbours may need to be updated later. The algorithm terminates when the workset is empty, which implies that there are no more active vertices to be processed.

Algorithm 4 Bellman-Ford's Parallel SSSP

```
Require: Graph $G$ and WorkSet $ws$
Ensure: Graph $G$ with each node's distance from the root
 1: $ws$.add(root)
 2: for each node $n$ in $ws$ do
 3:   for each node $m$ in $neighbours(n)$ do
 4:     $dist \leftarrow n.getDist() + G.edgeWeight(n, m)$
 5:     if $dist < m.getDist()$ then
 6:       $m.setDist(dist)$
 7:       $ws.add(m)$
 8:     end if
 9:   end for
10: end for
```

5.2.1 *Available Parallelism in SSSP*

The available parallelism in SSSP initially increases exponentially. At each step, all vertices in the workset can be processed in parallel without conflicts. The available parallelism drops when the algorithm runs out of vertices to process.

5.2.2 *Distribution of Vertices in the Workset*

To distribute the active nodes in the workset, we run the Bellman-Ford algorithm to generate a worksheet with $10 \times M$ vertices, where M is the number of available processing nodes. The vertices are then block distributed across the processing nodes. Each processor then operates on its portion of the workset and updates only its portion of the workset as new vertices need to be added. The vertices are retrieved and added to the workset at different ends in order to prioritize unprocessed vertices to be processed in the next iteration.

A processing node whose workset is empty attempts to steal vertices from the workset of other processors. The algorithm terminates when the workset of all the processing nodes is empty.

5.2.3 *Inefficiency of Bellman-Ford algorithm*

The distributed implementation of Bellman-Ford algorithm suffers from two distinct source of overheads. First, note that Dijkstra's algorithm relaxes each vertex only once. From the set of vertices that have not yet been relaxed, the Dijkstra's algorithm processes the vertices in increasing order of their tentative distances. However, Bellman-Ford algorithm processes edges in the graph through random selection. As such, Dijkstra's priority-based relaxation order converges much faster and is therefore, more work-efficient than the Bellman-Ford algorithm. Second, the block distribution of vertices does not account for the branching factor of the vertices distributed. Therefore, the worksets of different processing nodes will tend to be of substantially different sizes, specially in large graphs with high degrees of vertices. The uneven work distribution leads to the overhead of dynamic load balancing.

5.3 Δ-Stepping Algorithm

Dijkstra's algorithm and the Bellman-Ford algorithms employ two contrasting strategies for selecting active vertices in each iteration. The former chooses only one vertex, which is guaranteed to be settled, from the fixed priority queue. In contrast, the latter activates any vertex whose tentative distance was reduced in the previous iteration. The Δ-Stepping algorithm [15] strikes a balance between these two extremes and offers an efficient schedule for the fixpoint algorithm by weakening the total ordering constraint on the queue.

Algorithm 5 Δ-Stepping Algorithm for SSSP

```
Require: $G(V, E)$, source vertex $s$, length function $l : E \rightarrow R$
Ensure: $\delta(v), v \in V$, the weight of the shortest path from $s$ to $v$
 1: for each $v \in V$ do
 2:   $d(v) \leftarrow \infty$
 3: end for
 4: relax$(s, 0)$
 5: $i \leftarrow 0$
 6: while B is not empty do
 7:   $S \leftarrow \phi$
 8:   while $B[i] \neq \phi$ do
 9:     $Req \leftarrow \{(w, d(v) + l(v, w)) : v \in B[i] \wedge \langle v, w \rangle \in light(v)\}$
10:     $S \leftarrow S \cup B[i]$
11:     $B[i] \leftarrow \phi$
12:     for each $(v, x) \in Req$ do
13:       relax$(v, x)$
14:     end for
15:   end while
16:   $Req \leftarrow \{(w, d(v)+l(v, w)) : v \in S \wedge \langle v, w \rangle \in heavy(v)\}$
17:   for each $(v, x) \in Req$ do
18:     relax$(v, x)$
19:   end for
20:   $i \leftarrow i + 1$
21: end while
22: for each v $\in$ V do
23:   $\delta(v) \leftarrow d(v)$
24: end for
```

Algorithm 6 The *relax* procedure in Δ-Stepping algorithm

```
Require: $v$, weight request $x$
Ensure: Assignment of $v$ to appropriate bucket
 1: if $x < d(v)$ then
 2:   $B[\lfloor d(v)/\Delta \rfloor] \leftarrow B[\lfloor d(v)/\Delta \rfloor] \backslash \{v\}$
 3:   $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup \{v\}$
 4: end if
```

The Δ-Stepping algorithm partitions the vertices in the graph and hashes the vertices into an array B of *buckets* based on their distance label. The parameter Δ is a positive real number that represents the *bucket width*. For an index $k \geq 0$, the bucket B_k stores vertices v whose tentative distance fall in the range $[k\Delta, (k+1)\Delta - 1]$. The bucket index for a vertex v is given by $\lfloor \frac{d(v)}{\Delta} \rfloor$. A high level overview of the Δ-Stepping algorithm is shown in Algorithm 5.

The algorithm can perform deletion and edge relaxation for an entire bucket in parallel. The parameter, Δ, governs the amount of parallelism in each bucket and the number of extra updates on the vertices. For example, a large value of Δ yields increased parallelism per bucket, but also more wasted work. Intuitively, setting $\Delta = \infty$ yields Bellman-Ford algorithm whereas setting $\Delta = 1$ yields Dijkstra's algorithm.

Initially, the root vertex s is placed in the bucket B_0 and all other vertices are placed in the bucket B_∞. The algorithm works in multiple phases. The goal of phase k is to settle all the vertices whose actual shortest distance falls in the range of bucket k. The phase works in multiple iterations. In each iteration, a vertex u is declared to be active if its tentative distance changed in the previous iteration and the vertex is found in the bucket B_k. Note that in the first iteration of the phase, all vertices found in the bucket are considered active. For each active vertex v and all its incident edges $e = \langle u, v \rangle$, the algorithm performs the $\texttt{Relax}(u, v)$ operation.

A vertex may require multiple `Relax` operations to become settled. During a *Relax* operation, the tentative distance of a vertex in a bucket B_k may reduce in such a manner that the new value also falls within the current bucket's Δ value. In such a case, the vertex is re-inserted into the bucket B_k. If the new tentative distance of the vertex falls in the range of a bucket of lower index, the vertex is removed from its current bucket and inserted into the new bucket. Such movement of vertices among different buckets are treated as a step within the `Relax` procedure. The operations of the `Relax` procedure are shown in Algorithm 6.

We now focus on the operations inside each bucket (the inner while loop in Algorithm 5) in detail. Inside a bucket, the vertices are processed based on their edge weights. The edges with $(l(e) < \Delta, e \in E)$ are categorized as *light* edges while the edges with $(l(e) \geq \Delta, e \in E)$ are categorized as *heavy* edges. During each phase, the algorithm removes all vertices from the current bucket, adds the vertices to the set S, and selectively relaxes only *light* outgoing edges emanating from these vertices. This phase may result in new vertices being added to the current bucket, which are then deleted in subsequent phases. It is also possible that vertices previously deleted from the current bucket may be reinserted, if their tentative distance is improved. *Heavy* edges are not relaxed in a phase because they result in tentative values outside the current bucket. *Heavy* edges are only relaxed after all their respective starting nodes have become settled. In other words, once the current bucket becomes empty after relaxations, all heavy edges out of the vertices in S are relaxed at once. When the bucket does not contain any active vertices, the phase terminates and the algorithm proceeds to the next non-empty bucket of index higher than k.

Note that the edge relaxations in each phase can proceed asynchronously in parallel provided the individual tentative distance values are updated atomically.

5.3.1 *Distributed Implementation of the Δ-Stepping algorithm*

The vertices in the graph are block distributed in the form of buckets such that each vertex is owned by a processing node and belongs to its assigned bucket. Each processing node maintains the bucket B_k and also stores the adjacency lists of all vertices whose indices hash to the value k. Each processor relaxes only the edges between those vertices that belong to its bucket. All relaxations of edges incident at vertices belonging to other processing nodes must be transferred to their owners. Hence, the relaxation requests are generated and processed only locally.

While the vertices within a bucket may be processed in any order relative to each other, there are two choices as to the order in which the buckets may be processed. Ideally, processing the non-empty buckets in increasing order of their bucket numbers yields the best work efficiency and fast convergence of the relaxation operations. However, this approach also limits the utilization of processing nodes. Therefore, popular distributed implementations of the Δ-Stepping algorithm permit processing of the buckets out of order. The price of this additional parallelism is that some nodes may be relaxed repeatedly. The proper choice of the bucket width Δ for a given graph helps to strike a balance between work efficiency and parallelism.

Relax operations require communication among the processors: to perform $\texttt{Relax}(u.v)$, the owner of the source vertex u must send $d(u)$ to the owner of the destination vertex v (if the two owners are distinct). The iterations and phases are executed in a bulk synchronous manner. A processing node whose bucket is empty attempts to steal vertices from the bucket of other processors. The algorithm terminates when the buckets of all the processing nodes are empty. Termination checks and computing the next bucket index require collective reduction operations.

6. APPLYING WORKLOAD PARTITIONING AND SCHEDULING TO SSSP

A major challenge in distributing buckets across the processing nodes is to ensure an even distribution of the workload. A simple assignment of buckets where all processing nodes own the same number of vertices in the graph does not guarantee a balanced workload distribution. For instance, a processing node that owns vertices with high out-degrees will need to perform more relaxations compared to the processing node that owns vertices with lower out-degrees. Also, when all edges $\langle u, v \rangle$ of a high-degree vertex u are to be relaxed, then the value of the *tentative distance* $(d(v))$ must be made available to all processing units that store outgoing edges of u.

One approach to try to evenly distribute the workload would be to pre-process the graph to identify the out-degree of each vertex. Then, a distribution of buckets by considering the total out-degrees of vertices in each processing node would yield a more balanced workload distribution. However, traversing the entire graph to load the out-degree of each vertex is computationally expensive and may even offset the performance gains from improved load distribution. Furthermore, the out-degree of a vertex on its own is not a reliable indicator of the work associated with the node in the graph. For example, two vertices of the same out-degree could involve substantially different workloads because the number of paths passing through the two vertices could be quite different. Consequently, the number of edge relaxations required could be vastly different.

To reduce such an overhead, this work selectively observes the out-degree of some vertices in the graph and relies on those observations to predict the out-degree of unobserved vertices. We apply `WPS` for such a prediction and for distributing the vertices. The trick is to sample the graph with `StraSa` and create a tree of visited vertices. Specifically, we use `StraSa` to estimate the total number of outgoing edges in each bucket of the Δ-Stepping algorithm. Then, we apply the BLDM algorithm for partitioning the buckets into M parts of similar total number of outgoing edges.

The problem with such an approach is that `StraSa`'s sampling procedure is not able to estimate the number of edge relaxations that will be performed during the actual search. This problem arises because `StraSa` samples a tree and is not able to capture the actual structure of the underlying graph—the number of edge relaxations performed during search is closely related to the graph structure. In this paper we assume that the information `StraSa` samples is enough to make effective partitions of the *WIS*, the set of working items processed by the Δ-Stepping algorithm. Specifically, we use the estimated number of outgoing edges in a given bucket B as a surrogate for the total number of edge relaxations that occurs in B during the execution of the Δ-Stepping algorithm. In Section 7 we show empirically that the number of outgoing edges is in fact a good surrogate for the number of edge relaxations as `WPS` is able to achieve substantial speedups while parallelizing the Δ-Stepping algorithm.

We note that the problems `WPS` was evaluated on in its original paper had a natural tree structure, which allowed `WPS` to sample the working items directly and there was no need to employ a surrogate function for `WPS`'s sampling.

6.1 Stratification

The stratified sampling technique underlying `WPS` samples all vertices with properties unique according to the stratification. Ideally, the strata should be representative of the vertices in the input graph. The goal is to yield a precise estimate of the number of outgoing edges in different buckets, even if the properties of the vertices are heterogeneous within the same stratum. A stratification will be successful if the vertices can be partitioned into different strata, each of which is internally homogeneous, by using prior information about the vertices. If each stratum is homogeneous, then the property used for stratification varies little from one vertex to another within a stratum. Consequently, a precise estimate of any stratum parameter can be obtained from a small sample in that stratum. These estimates can then be combined to obtain a precise estimate for the entire tree. However, if the vertex properties within a stratum vary considerably, many samples from within that stratum will be necessary for a precise estimate. Thus, depending on the choice of stratification, i.e., the sampling heuristic, the sampling process has the potential to under-estimate or over-estimate the overall workload that an application generates.

Our evaluation uses a stratification that labels vertices u and v in S with the same stratum if the sum of the out-degrees of the vertex u, its children and its grandchildren is equal to the sum of the out-degrees of the vertex v, its children and its grandchildren. The intuition behind this stratification is that we expect the number of reachable outgoing edges from u to be similar to the number of reachable outgoing edges from v if u and v have the same number of outgoing edges up to a distance of two edges from u and v.

6.2 Usage of Multiple Probes

Having a homogeneous stratification is one way of producing accurate predictions. Another way is by using a large number of probes. By increasing the number of probes `StraSa` tends to be more accurate at the expense of an increased running time. The estimation in this paper uses k independent probes of `StraSa`. Each subsequent probe of sampling may visit different vertices at a given level in the tree than its predecessor. Visiting a new vertex at the same level of the tree typically yields additional information about the sampled space of vertices. To accommodate such new information, on each subsequent sampling probe, the stratification updates the sampled tree from the previous probe by including the newly visited vertices. This approach guarantees that the sampled tree encompasses all the paths visited during sampling.

The estimation phase then uses the average of the vertices in the sampled tree at a given level and belonging to the same stratum while computing the out-degree of the subtrees rooted at the vertices. Section 8 discusses the combination of stratification and the value of k that yields the best performance for SSSP in our evaluation platform.

6.3 Bucket Distribution using `WPS`

Once `WPS` estimates the out-degree of the vertices in the graph, the next step involves grouping buckets into collections such that the total out-degree of vertices in each collection is as equal as possible. Finally, the collections of buckets are assigned to the available processing nodes. The empirical evaluation in this paper defines the bucket width Δ in such a way that there are a large number of buckets compared to the available processing nodes. This strategy ensures that there is a realistic opportunity for load balancing. Large bucket widths will lead to fewer buckets, which are not practical enough for load balancing purpose because vertices assigned to a bucket cannot be re-assigned to alternative buckets.

7. EXPERIMENTAL SETUP

Platform.

Performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux 6.2. All binaries were compiled with GCC 4.4.3 using the -O3 flag.

Compiler and Runtime.

The SSSP algorithm is implemented in the X10 programming language. The x10C++ compiler 2.3.1 is used for all measurements. The nodes in the cluster are connected by an InfiniBand network with a bandwidth of 10 Gbit/s and use MVAPICH2 library for communication. The experimental runs create eight worker threads per node and vary the number of nodes from 1 to 16 so that the number of threads is the same as the total number of cores.

Stratification and Number of Probes.

As explained before, our evaluation uses a stratification that labels vertices u and v in S with the same stratum if the sum of the out-degrees of the vertex u, its children and its grandchildren is equal to the sum of the out-degrees of the vertex v, its children and its grandchildren. Section 8 discusses the performance impacts of different stratifications and number of probes.

8. EXPERIMENTAL EVALUATION

The experimental evaluation in this paper uses two graphs as inputs for the SSSP problem. The small input is a graph with 2.39 Million nodes and 5.77 Million edges with the weights in the range (0, 1000], and a branching factor in the range of [1, 10]. The sequential execution time to compute the SSSP in this graph takes 287 seconds using Dijkstra's algorithm.

The large input is a graph with 268 Million nodes and 1 billion edges with weights in the range (0, 100000], and a branching factor in the range of [1, 1000]. The sequential runtime to compute the SSSP in the large graph is 462 seconds.

8.1 Speedups

The choice of an appropriate metric for proper evaluation of a workload partitioning algorithm depends both on the algorithmic properties and on the computing platform. For instance, balancing message traffic across distributed-memory machines could be more important than balancing the amount of computation executed on each processing node in the context of large-scale production data centers. On the contrary, for long-running applications, it is important to improve the overall execution time by reducing both the message traffic and by balancing the computational workload on each processing node. WPS enables programmers to easily specify such a choice of metric that belies the partitioning goal.

For the SSSP problem, this work uses speedup as the metric for evaluating the performance of WPS. The speedups of different algorithms on these inputs are shown in Figures 1 and 2. The reported speedups are relative to the sequential implementation of Dijkstra's ordered algorithm, which uses a heap-based priority queue. The relative speedup for these algorithms increases as the problem size increases. This is because smaller graphs possess insufficient parallelism to saturate the available 128 processors. Note that the figures only show the performance of Bellman-Ford and Δ-Stepping algorithms using Dist-WS [18] for distributed load balancing. Dist-WS yields better performance than using X10's intra-node work-stealing scheduler only. The best performance is achieved with the Δ-Stepping algorithm using WPS for even workload distribution across nodes, and by relying on X10's work-stealing scheduler for intra-node load balancing. For instance, the Δ-Stepping approach to SSSP using WPS outperforms the baseline Δ-Stepping by 41% in terms of execution time.

8.2 Performance Impact of Bucket Width (Δ)

The choice of the Δ value may significantly impact the performance of the Δ-Stepping algorithm. The Δ parameter impacts the total number of relaxations and the number of phases necessary to compute the SSSP. Empirical evaluation shows that Δ values of 400 and 8,000 yield best results for the small and large graph inputs, respectively.

The evaluation also shows that there is no direct correlation between the choice of stratification and the value of Δ. The same value of Δ leads to the best performance for different stratifications, as we now show.

8.3 Performance Impact of Stratification

Stratification 1 ($S1$) is the stratification described above in Section 7. Stratification 2 ($S2$) is a coarse version of $S1$, i.e., with fewer strata. Namely, $S2$ considers two items to be of different stratum if the sum of the out-degrees of the children and the grandchildren they produce differ by more than two. The numbers in parenthesis in Figure 4 represent the number of probes used. WPS performs worse in all domains when using a single probe (see WPS (1)). The performance does not improve even when coordinating WPS with Dist-WS (see WPS+). WPS yields best performance under $S1$ and $S2$ at

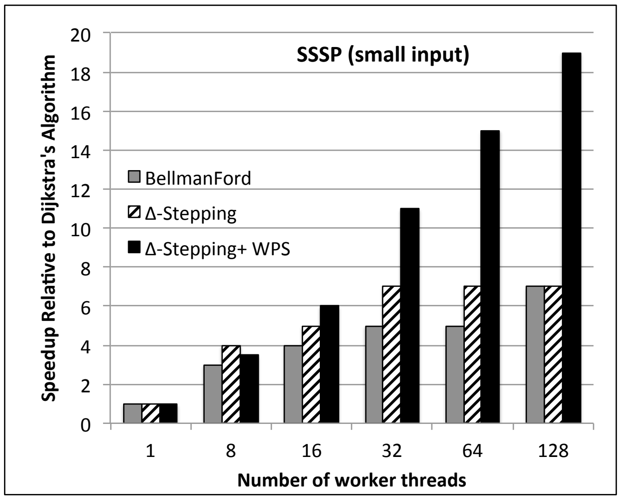

Figure 1: Performance of SSSP using small input.

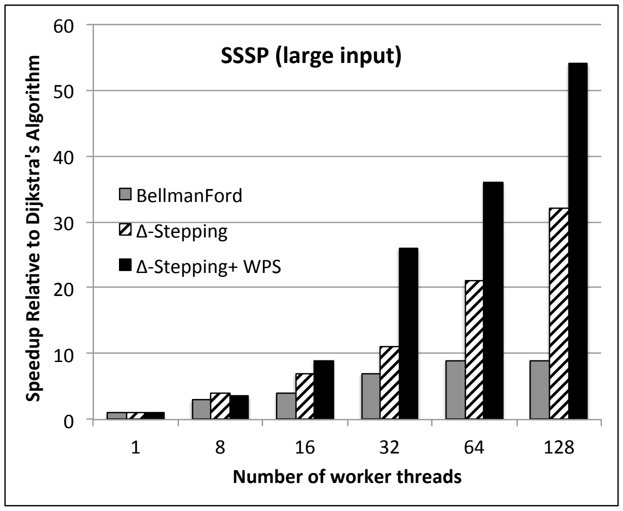

Figure 2: Performance of SSSP using large input.

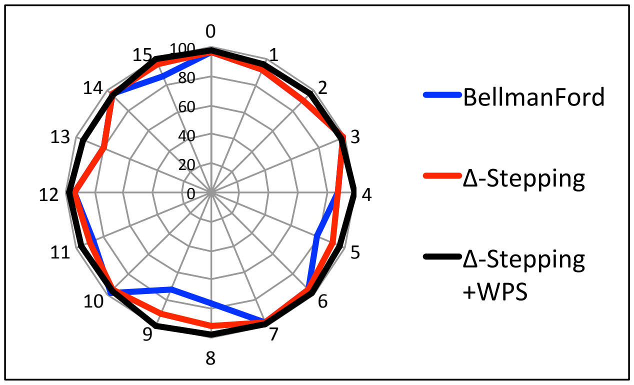

Figure 3: Average Node Utilizations for SSSP Algorithms.

five and ten probes of StraSa respectively.

Increasing the number of probes beyond five for $S1$ and ten for $S2$ is not beneficial. In fact, the performance wors-

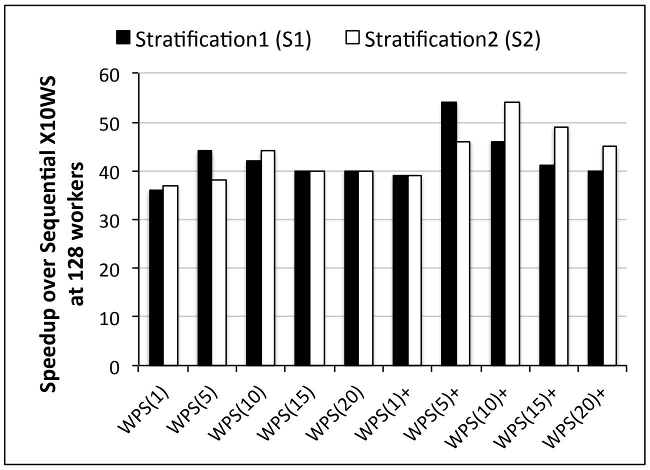

Figure 4: Performance of SSSP using `WPS` with different stratifications. The number of probes used are indicated in the parentheses, and the '+' sign in the x-axis labels indicate `WPS` + `Dist-WS`.

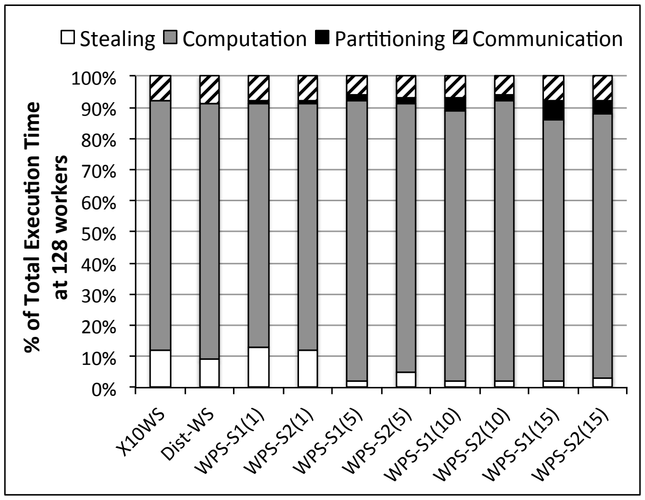

Figure 5: Breakdown of execution time of SSSP when using different algorithms and stratifications.

ened at larger number of probes because of the overhead of sampling. Figure 5 shows the increasing amount of time spent on partitioning as the number of probes increases. At lower number of probes, the times spent on partitioning is practically negligible. However, the resulting loss of precision of the estimates leads to poor load distribution. As a result, the time spent on communication tends to be larger at lower number of probes. The loss in performance is regained by coordinating `WPS` with `Dist-WS`.

Through an improved workload distribution, significantly reduced work-stealing operations, and reduced machine idle times, `WPS` achieves an increased and uniform CPU utilization compared to `X10WS` and `Dist-WS`. The radial axes and the points in the circumference of the circles in Figure 3 respectively indicate the average CPU utilization of eight cores in a node and a node in the cluster.

9. RELATED WORK

Balancing the distribution of computational workload while minimizing communication across distributed memory machines is fundamental to the efficiency of distributed graph processing algorithms.

Many graph libraries and graph processing systems developed as part of research initiatives and industry needs facilitate workload partitioning in large graphs. Libraries, such as Pregel [14] and its open-source counterpart Giraph, FENNEL [20], GoldenOrb [3], and HAMA [19] offer multiple approaches for partitioning the graph data. The three common approaches to partitioning the data are hash-based, range-based, or min-cut. Hash and range-based partitioning approaches divide a dataset based on simple heuristic: to evenly distribute vertices across compute nodes, irrespective of their edge connectivity. Min-cut based partitioning approaches do consider vertex connectivity and partition the data such that it places strongly connected vertices close to each other, such as on the same cluster.

Many popular graph processing platforms such as Pregel [14] that builds on MapReduce, and its open source cousin Apache Giraph, PEGASUS [6] and GraphLab [12] use as a default partitioner Hash Partition of vertices, which corresponds to assigning each vertex to one of the k partitions uniformly at random. This heuristic efficiently balances the number of vertices over different clusters, but ignores the differences in workload involved in processing different vertices.

These graph libraries have been shown to yield good performance for certain algorithms. Leveraging the efficiency of these libraries requires a substantial implementation effort in writing applications using provided library routines, and must be repeated for each new algorithm or graph representation. These libraries and graph-processing infrastructures inherently assume that either the entire graph is known statically or that the structure of the graph is static while performing graph partitioning.

In contrast to these existing graph libraries and infrastructures, this work builds on a graph partitioning approach that is agnostic to graph representation, and is not tied to a specific implementation or an infrastructure. This work is a demonstration that the idea of `WPS`, proposed by Paudel et al., applies well to the problem of workload partitioning in the Single Source Shortest Path problem.

Another approach to workload partitioning that is closely related to the `WPS` technique applied in this work is the approach adopted in the Mizan system [7]. Mizan uses a Bulk Synchronous Parallel-based approach which first reads and partitions the graph data across workers. The system then proceeds as a series of supersteps, each separated by a global synchronization barrier. During each superstep, each vertex processes incoming messages from the previous superstep and sends messages to neighbouring vertices (which are processed in the following superstep). Mizan balances its workload by moving only selected vertices across workers. Vertex migration is performed when all workers reach a superstep synchronization barrier to avoid violating the computation integrity, isolation and correctness of the BSP compute model. Unlike Mizan, `WPS` does not require distributed runtime monitoring of workload characteristics, and a distributed migration planner that decides which vertices to migrate.

10. SUMMARY

This work describes the use of the workload partitioning approach WPS on applications with explicitly-defined graphs. An evaluation using WPS for workload distribution in the well-known SSSP problem showed that WPS can estimate, with reasonable accuracy, the entire workload in an application with explicitly-defined graph by visiting only a small set of the vertices in the graph. The estimate offers a good basis for balanced workload distribution with minimal runtime overhead. Using WPS for workload distribution in SSSP in our evaluation platform yields 41% improvement in execution time at 128 worker threads compared to the state-of-the-art Meyer and Sanders' distributed solution to SSSP. This study also confirms the expected tradeoff between accuracy and the runtime of the sampling algorithm underlying WPS. Another important observation from this study is that WPS offers flexibility in choosing a suitable objective that accommodates specific needs of an application while performing workload partitioning.

11. REFERENCES

[1] P.-C. Chen. Heuristic Sampling: A Method for Predicting the Performance of Tree Searching Programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[3] GoldenOrb. Massive-Scale Graph Analysis. http://goldenorbos.org, 2011. [Online; accessed 9-Jun-2015].

[4] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[5] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[7] 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*, EuroSys '13, pages 169–182, Prague, Czech Republic, 2013. ACM.

[8] D. E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29:121–136, 1975.

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *Programming Language Design and Implementation*, PLDI '07, pages 211–222, San Diego, California, USA, 2007. ACM.

[10] L. H. S. Lelis, R. Stern, and N. Sturtevant. Estimating Search Tree Size with Duplicate Detection. In *Proceedings of the Symposium on Combinatorial Search*, pages 114–122. AAAI Press, 2014.

[11] L. H. S. Lelis, S. Zilles, and R. C. Holte. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence*, pages 53–76, 2013.

[12] 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, Apr. 2012.

[13] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX).* SIAM, 2007.

[14] 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*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] U. Meyer and P. Sanders. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 393–404, London, UK, UK, 1998. Springer-Verlag.

[16] W. Michiels, J. Korst, E. Aarts, and J. Leeuwen. Performance Ratios for the Differencing Method Applied to the Balanced Number Partitioning Problem. In *STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 583–595. Springer Berlin Heidelberg, 2003.

[17] J. Paudel, L. H. S. Lelis, and J. N. Amaral. Stratified Sampling for Even Workload Partitioning Applied to IDA* and Delaunay Algorithms. In *IEEE International Parallel & Distributed Processing Symposium*, Hyderabad, India, 2015.

[18] J. Paudel, O. Tardieu, and J. N. Amaral. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *International Conference on Parallel Processing*, pages 100–109, Lyon, France, 2013.

[19] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.

[20] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.