Engineering a Parallel Δ-stepping Algorithm

Erika Duriakova[1] Deepak Ajwani[2] Neil Hurley[1]

[1]Insight Centre for Data Analytics, University College Dublin
first.last@insight-centre.org
[2]School of Computer Science and Informatics, University College Dublin
first.last@ucd.ie

Abstract

Computation of the single-source shortest path (SSSP) is a fundamental primitive in many network analytics tasks. With the increasing size of networks to be analysed, there is a need for efficient tools to compute shortest paths, especially on the widely adopted shared-memory multicore architectures. The Δ-stepping algorithm, that trades-off the work efficiency of Dijkstra's algorithm with the parallelism offered by the Bellman-Ford algorithm, has been found to be among the fastest implementations on various parallel architectures. Despite its widespread popularity, the different design choices in implementing the parallel Δ-stepping algorithm are not properly understood and these design choices can have a significant impact on the final performance. In this paper, we carefully compare two different implementations of the Δ-stepping algorithm for shared-memory multicore architectures: (i) a static workload assignment where the nodes are assigned to threads at the beginning of the algorithm and only the assigned thread can relax edges leading to a node and (ii) a dynamic workload assignment where the nodes are dynamically allocated to threads at the time of bucket relaxation. Based on an extensive empirical study on a range of graph classes, edge density and weight distributions, we show that while the more intuitive and widely used approach of dynamically balanced workload suits dense power-law graphs well, the static partitioning approach outperforms this more intuitive approach on a wide range of graph classes. Our findings can guide a network analyst in selecting the best parallel implementation of the Δ-stepping algorithm for a given analytics task and a given graph class.

1 Introduction

The SSSP computation is a fundamental primitive in many network analytics tasks. For instance, the computation of betweenness [10] and closeness centrality of nodes require repeated computation of shortest paths between nodes. Similarly, studying the information spreading on networks often involves computation of shortest paths. With the increasing size of networks to be analyzed, we need efficient computation of shortest paths, motivating further research into scalable shortest path computations.

In this paper, we focus on engineering shortest path algorithms that leverage the parallelism offered by shared memory multicore architectures. Furthermore, we perform a thorough investigation of the various engineering choices that impact on the performance of these algorithms.

Consider a graph, $G(V, E)$ with n nodes and m edges, such that each edge $(v, u) \in E$ has an associated positive weight $w(v, u)$. SSSP algorithms typically initialise and update a tentative distance $d^*(v)$ associated with each node $v \in V$, that upper bounds the true shortest distance. This tentative distance is updated using the process of edge *relaxation*. Relaxing an edge (v, u) updates $d^*(u)$ as $\min\{d^*(u), d^*(v) + w(v, u)\}$. Dijkstra's well-known algorithm [6], greedily chooses the node of minimum tentative distance to visit and relaxes all its incident edges. This guarantees that any node which is visited has found its true shortest distance and therefore no edge needs to be relaxed twice, resulting in $O(m + n \log n)$ work with Fibonacci heaps [9]. However, the sequential nature of the algorithm makes its parallelisation difficult. On the other hand, the Bellman-Ford algorithm [1, 8] relaxes all edges in each round. This is more amenable to parallelisation, but has a significantly worse asymptotic complexity of $O(nm)$ for total work. The Δ-stepping algorithm [16] tries to find a balance between the low work but sequential Dijkstra's algorithm and the high work, but parallelisable Bellman-Ford algorithm. As such, this has been found to be among the fastest and most practical on numerous parallel architectures ([7, 11, 14, 17, 18, 21]).

Despite its widespread popularity, the different design choices in implementing the parallel Δ-stepping SSSP algorithm are not properly understood and these design choices can have a significant impact on the final performance. In this paper, we address this gap. Specifically, we explore two different approaches to workload assignment: (i) a static workload assignment where the nodes are assigned to threads at the beginning of the algorithm and only the assigned threads can relax edges leading to these nodes and (ii) a dynamic workload assignment where the nodes are dynamically allocated to threads at the time of bucket relaxation. We perform an extensive empirical study on a range of graph classes, edge density and weight distributions. We find an intriguing trade-off between these approaches: the popular dynamic workload assignment results in better workload balance among the processors and generates fewer requests for node relaxation, but its performance suffers from the locking mechanism to deal with racing conditions; on the other hand, the static partitioning of nodes results in more workload imbalance and generates more requests for node relaxations, but it can avoid racing conditions without requiring any locks. Thus, while the dynamic strategy works well for dense power-law graphs, the static partitioning of nodes is surprisingly better on a wide range of other graph classes. This surprising finding has interesting implications for application developers: while traditionally the dynamic approach has been popular, our results imply that this is not always the best choice. In many settings, the static partitioning may be significantly better.

2 Background and Related Work

2.1 Single Source Shortest Path Algorithms

SSSP algorithms compute the shortest paths from the given source node $s \in V$ to every other node in the graph. The SSSP algorithms discussed here all initialise and update a *tentative distance*, $d^*(v)$, to each node $v \in V$, which upper bounds the true shortest distance, $d(v)$. As they proceed, $d^*(v)$ is reduced to $d(v)$, using the *edge relaxation* process. Given an edge $e = (v, u)$, relaxation updates $d^*(u)$ with

$$d^*(u) \leftarrow \min(d^*(u), d^*(v) + w(v, u)) .$$

A node v is said to be *settled* when $d^*(v) = d(v)$. Initially all nodes are unsettled with $d^*(s) = 0$ and $d^*(v) = \infty$ for $v \neq s$.

In the Δ-stepping algorithm, a parameter Δ is used to partition the nodes into buckets according to their tentative distances from the source, such that node v is placed in bucket B_k when $k\Delta \leq d^*(v) < (k+1)\Delta$. Using the terminology of [2], the algorithm proceeds in a number of *epochs*, such that, during an epoch all nodes in B_k, where k is the minimum index such that B_k is non-empty, are settled and hence removed, so that the bucket is emptied. Within an epoch, the algorithm proceeds in phases. At the start of the first phase, all nodes in B_k are active. Relaxation is applied across each edge of each active node. This may result in a new tentative distance for the neighbouring node along the relaxed edge, possibly requiring it to be moved from its current bucket and *re-inserted* in a bucket of lower index, which may be the current bucket B_k. Those nodes in B_k whose tentative distances have changed are selected as the active nodes for the next phase. An epoch ends when there are no active nodes remaining in B_k, which implies all nodes up to distance $(k+1)\Delta$ from the source have been settled. The value $\Delta = \infty$ corresponds to the Bellman-Ford algorithm, while as $\Delta \to 0$, Δ-stepping reduces to a variant of Dijkstra's algorithm.

It is useful to classify edges into two categories. Edges e such that $w(e) < \Delta$ are *light* edges and the remaining edges with $w(e) \geq \Delta$ are *heavy*. Since the relaxation of a heavy edge can never result in a re-insertion into B_k, it is possible to break the processing of an epoch into multiple light edge phases during which only the light edges incident on the current set of active nodes are relaxed. Such light edges may be relaxed more than once, if the node remains as an active node in B_k at the end of the phase. A single heavy edge phase ends the epoch, during which all heavy edges incident on all active nodes encountered during the epoch are relaxed.

Parallelisation is achieved in Δ-stepping by processing each bucket in parallel across multiple threads, see Algorithm 1 in which a source node $s \in V$ is given as input and the distances to s, $d(v)$ as well as the paths, represented by the parent array $p(v)$ are output. The processing of an epoch involves the following three loops which can be carried out in parallel:

- **Loop 1**: generate light and heavy request sets (R_l and R_h respectively) for each node $v \in B_k$ as a set of triples (v, u, w), consisting of the node v, its neighbour u and the connecting edge weight w.
- **Loop 2**: relax light edges.

- **Loop3**: relax heavy edges

The first two loops are repeated for each light edge phase of the epoch, while the last loop occurs only once in each epoch. In loop 2 and 3 the `Relax` subroutine updates the tentative distance of u and re-assigns it to a new bucket if necessary.

2.2 Parallel SSSP implementations

Numerous parallel SSSP algorithms have been proposed to date, including asynchronous parallel Dijkstra's algorithm [22], parallel SSSP algorithm for large dynamic graphs [20] or an approach of translation of traditional graph algorithms into the GraphBLAS interface [19]. Our focus is to efficiently solve SSSP on large static weighted graphs using standard shared memory multi-core architectures. In this setting, Δ-stepping has been a highly popular algorithm and state-of-the-art implementations of parallel Δ-stepping or its variants have been found to be among the fastest (with a notable exception of experiments reported in DSMR [15]) on massively parallel computing environments [14], GPGPUs [21], multicores [4, 17] and distributed architectures [7, 11, 18].

We implemented the Δ-stepping algorithm from scratch, as it enabled us to explore design choices for the Δ-stepping, that have not been considered in the previous work. Nonetheless, we expect that the relative performance between these different design choices will be similar for other graph-processing frameworks [4, 17] for multicores.

3 Engineering parallel Δ-stepping algorithm

Parallelisation in Δ-stepping is achieved by organising nodes into so-called relaxed priority queues, which we call buckets, using a tunable parameter Δ. Each bucket can then be processed in parallel across multiple threads. As explained in Section 2, the processing of each bucket involves three parallel loops.

Our two implementations differ in the way they balance the work of each bucket among the different threads and the way in which they avoid race conditions—a situation that arises when different threads update the tentative distances of the same node with different values.

The first implementation avoids the race conditions by statically assigning nodes to threads at the start of the computation. This approach can lead to an imbalanced workload during the edge relaxation process, however, as we will show in the later sections, this imbalance is not as high as one might expect in many graph classes. The other implementation achieves more balanced workload with the use of dynamic thread allocation during the process of edge relaxation in each bucket. While this approach results in more balanced workload, it also encounters race conditions when more than one thread attempts to update the tentative distance of a particular node concurrently. As a result, this implementation relies on a locking mechanism to protect concurrent updates. Surprisingly none of the state-of-the-art multicore shared-memory Δ-stepping implementations considered the static approach to avoid locking, however, as we will show in later sections, the locking mechanism slows the dynamic approach on a wide range of graph classes.

Algorithm 1 Parallel Δ-stepping `sssp`

Require: $G(V, E)$, s, Δ

for $\forall(v \in V)$ **do**
  define N_v^l and N_v^h
  $d^*(v) \leftarrow \infty$, $B_\infty \leftarrow \{v\}$
end for
$B_\infty \leftarrow B_\infty \setminus \{s\}, \quad B_0 \leftarrow B_0 \cup \{s\}, \quad d^*(s) \leftarrow 0$
$k \leftarrow 0$
while $k < \infty$ **do**
  $R_h \leftarrow \emptyset$, $R_l \leftarrow \emptyset$
  while $B_k \neq \emptyset$ **do**
    for $\forall(v \in B_k)$ **do in parallel** ▷ Loop 1
      $B_k = B_k \setminus \{v\}$
      $R_l = GenRequests(R_l, N_v^l)$
      $R_h = GenRequests(R_h, N_v^h)$
    end for
    for $\forall((v, u, w) \in R_l)$ **do in parallel** ▷ Loop 2
      $R_l \leftarrow R_l \setminus \{(v, u, w)\}$
      $\text{relax}(v, u, w)$
    end for
  end while
  for $\forall((v, u, w) \in R_h)$ **do in parallel** ▷ Loop 3
    $R_h \leftarrow R_h \setminus \{(v, u, w)\}$
    $\text{relax}(v, u, w)$
  end for
  $k \leftarrow \min_i\{i : B_i \neq \emptyset\}$
end while
function GenRequests(R, N_v)
  for $\forall(u \in N_v)$ **do**
    $R = R \cup \{(v, u, w(v, u))\}$
  end for
  return R
end function
function Relax($v, u \in V, w \in \mathbb{R}$)
  if $d^*(v) + w < d^*(u)$ **then**
    $i \leftarrow \lfloor \frac{d^*(u)}{\Delta} \rfloor$, $j \leftarrow \lfloor \frac{d^*(v)+w}{\Delta} \rfloor$
    $B_i \leftarrow B_i \setminus \{u\}$, $B_j \leftarrow B_j \cup \{u\}$
    $d^*(u) \leftarrow d^*(v) + w$, $p(u) \leftarrow v$
  end if
end function

Ensure: $d^*(v), p(v) \forall v \in V$

3.1 Static Partitioning of Nodes

The first implementation relies on a static partitioning of nodes to threads. The nodes are initially randomly partitioned among the threads and all updates related to a node are performed only by the thread to which it has been assigned. As may be observed from Algorithm 1, parallelisation requires concurrent updates of the buckets B_i and the request lists consisting of light edges R_l and heavy edges R_h.

Recall that request lists consist of triples (v, u, w), where $v \in V$ is the node being processed, $u \in V$ is its neighbour and $w \in \mathbb{R}$ is the weight along the edge connecting v to u over which relaxation is to be performed. In our static implementation, these data-structures are split across the threads. Each bucket B_i is partitioned as $B_i = \cup_t B_i^t$, where nodes assigned to thread t and B_i are placed in bucket B_i^t. Similarly the triple $(v, u, w) \in R_l$ is stored in $R_l^{t,r}$, where $R_l = \cup_{t,r} R_l^{t,r}$ such that node v is assigned to thread t and node u is assigned to thread r. A similar partitioning is applied to R_h. Loop 1 is partitioned across the threads according to the thread assignment to node v, while Loops 2 and 3 are partitioned across the threads according to the assignment of neighbour node u in each triple (v, u, w). In Loop 1, thread t builds B_i^t, $R_l^{t,r}$ and $R_h^{t,r}$ for all r, while each thread r in the request processing Loops 2 and 3, processes the requests in $R_l^{t,r}$ or $R_h^{t,r}$ for all threads t. Thus, the thread assigned to u carries out the relaxation that updates $d^*(u)$ and re-assigns u to a new bucket. Note that, because of this data partitioning, none of the write operations incurs race conditions. Since the partitioning is based on a random assignment of nodes to threads, the workload is fairly balanced, but not perfectly balanced.

To provide $O(1)$ insertions of nodes to, and deletions of nodes from buckets, the buckets are implemented as doubly linked lists. For each bucket and each thread, we store the front and back pointers of this doubly linked list corresponding to the part of the bucket stored by the thread. In order to find the bucket and the thread of the node efficiently, we store all the doubly linked lists in the form of one global array. The i^{th} entry in this array contains the forward and backward pointers of the i^{th} node corresponding to its bucket and thread. When a node is moved from one bucket to another, the corresponding thread simply updates the node's forward and backward pointers appropriately to assign it into its new bucket. It also updates the forward pointer of the predecessor node and backward pointer of the successor node in the old bucket to remove the node from the old bucket.

3.2 Dynamic Partitioning with Global Buckets

An alternative to the static partitioning scheme, is to allocate the work of the parallel loops dynamically. B_i, R_l and R_h are stored as global data-structures shared among all threads. The dynamic partitioning approach implemented in this work is similar in structure to the parallel implementation of Madduri et al. [14] for the Cray MTA-2. However, because of the differences in architecture and compiler, we did not have access to the atomic operations on code blocks that allowed them to use global arrays for the request set.

In our implementation, we store the buckets as arrays to enable a simple invocation of dynamic parallelisation over the array structure (using `OpenMP`'s `parallel`

`for` in our implementation). During each light edge phase, the nodes in B_k are distributed among the threads and R_l and R_h are built concurrently. Write operations on R_l and R_h must be handled so that race conditions are avoided. In Loops 2 and 3, the request triples are distributed dynamically among the threads, so that relaxations are balanced. As relaxation can require the moving of nodes from one bucket to another, again care needs to be taken to avoid race conditions on bucket updates. The optimisation suggested in [14] of only storing the *strictest request* for each node in R_l and R_h is applied. That is, at most one request per neighbour node is stored in R_l and R_h, corresponding to the request that will reduce its tentative distance the most. This is achieved during the execution of Loop 1. For each triple (v, u, w), the position of u in the request set is recorded. Before a new triple (v', u, w') is added into the request set, it is compared against the one previously added, to determine which will improve $d^*(u)$ the most. Before updating the request set, a lock associated with u must be obtained, ensuring that no other updates involving u can occur simultaneously. Bucket updates require the removal of a node u from its current bucket and adding it to a new bucket. For removal, the bucket array location is simply updated to create an empty element, which can be done without race conditions, since u appears in the request list only once. Nodes are inserted at the end of each bucket. During each phase, each thread keeps a local record of the number of nodes to add to each bucket. A prefix-sum across the threads then allows all threads to complete the insertion in parallel at the end of the phase. Note that our storage scheme implies that gaps appear in the bucket arrays as nodes are removed. When the proportion of gaps reaches a threshold, a bucket is collapsed to reuse empty space.

4 Experimental Results

4.1 Experimental Framework

Multicore Architectures. We consider the following architectures:

- **Config A:** an Intel Ivybridge E5-4607 processor with 2200 MHz CPU, 128 GB memory, 2 NUMA nodes and 2 sockets with 10 cores each.
- **Config B:** an Intel Ivybridge E5-2660 v2 processor with 2200 MHz CPU, 768 GB memory, 4 NUMA nodes and 4 sockets with 6 cores each.

For the sake of simplicity, the maximum number of threads used fit within one NUMA node to avoid the NUMA effects. Further, for smaller input graphs we report results obtained from Config A, while for large input problems such as the ones presented in Section 4.6, we use Config B. However, we note that the relative performance of both algorithms is along the same lines on both architectures.

Graph Classes.

We focus on numerous graph classes ranging from short to large diameter graphs as described bellow:

- Random graphs G(n, m) with fixed number of edges $m = cn$ and edges chosen uniformly at random (closely related to Erdos-Renyi graphs $G(n, p)$ with $p = \frac{c}{n}$).

- The RMAT-1 graphs RMAT(n, m) with $m = cn$ created using the PaRMAT generator [12]. These random graphs produce a skewed degree distribution, depending on the setting of three parameters, A, B and C. We choose $A = 0.45$, $B = C = 0.22$, which leads to graphs with highly skewed degrees.
- Grid graphs `Grid`(n_1, n_2) consisting of a grid of $n1 \times n2$ vertices. In the full grid, each vertex is connected to each of its immediate neighbours, namely, the vertices to the left, right, above and below it, if they exist. However, edges are removed independently at random, with probability p, which we set to 0.1.
- Real World Graph:
  - Orkut social network [13] with 3,072,441 nodes and 162,185,083 edges.

For all graphs, we remove duplicate edges, extract the largest connected component and randomise the node order.

In all synthetic datasets, we assign the weights on the edges as floating-point values in the range [0,1). We set the edge weights according to two different random distributions, (i)Uniform random distribution and (ii) Power-law distribution i.e. $P(w) \propto w^{-\alpha}$, such that most connecting weights are weak, but there exist some weights much stronger than the average weight. We choose $\alpha = 1.287$, based on the values reported in [3].

Measurements Apart from running time and parallel speed-up we collect a number of additional statistics. In order to find the best partitioning for the parallel Δ-stepping algorithm, one must find the best trade-off between workload and parallelism. The workload associated with each implementation can be decomposed into the following operations:

- **Bucket update size**: the total size of the request lists R_l and R_h summed over all epochs and phases.
- **Relax size**: the total number of requests in R_l and R_h that result in a relaxation.
- **Request size**: the total number of relaxations that result in a bucket update.

Further to better compare the draw-back of each Δ-stepping partitioning we measure the following:

- **Imbalance**: $\text{imbal}(p) = (W_{\max} - W_{\min})/W^T$, where $W_{\max} = \max\{W_1, \ldots, W_p\}$ and $W_{\min} = \min\{W_1, \ldots, W_p\}$ and W_i is the work carried out by processor i and W^T is the overall work carried out.
- **Max waiting**: time spent in locks in Dynamic Δ-stepping algorithm.

4.2 Finding Best Δ

The tuneable parameter Δ is used to organise the nodes into relaxed priority queues, called buckets. Smaller values of Δ increase the overall number of buckets and decrease the number of nodes in each bucket. This in turn increases the overall number of epochs and decreases the available amount of parallelism within each epoch.

Madduri et al. [14] observed, that short diameter graphs generally result in smaller number of epochs with large number of light edges in each epoch when $\Delta = n/m$. On the other hand, using long diameter graphs and the same value of Δ, the algorithm needs very large number of epochs with small number of light edges within each epoch. This suggests that large diameter graphs could benefit from larger value of Δ

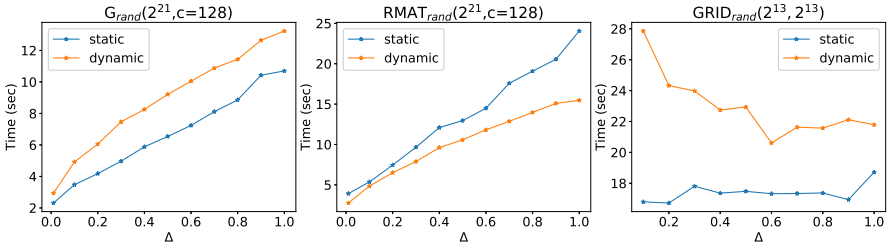


Figure 1: Parallel running time (using 10 threads) as a function of Δ. (Config A).

to increase the number of requests in each epoch. Our results are along the same lines. As can be observed in Figure 1 short diameter graphs perform best with small Δ value while large diameter grid graph performs best with larger value of Δ. Based on this results in the rest of the experiments we use $\Delta = 0.01$ for small diameter graphs and $\Delta = 0.6$ for large diameter graphs.

4.3 Workload in Δ-stepping Algorithm

In order to find the best partitioning for the parallel Δ-stepping algorithm, one must find the best trade-off between workload and parallelism. By design, the dynamic implementation minimises the size of the request list, by maintaining only the strictest request for each node. It distributes the request list triples dynamically across the threads, so that the resulting relaxations and bucket updates should be well balanced. The static implementation forms a request for the neighbours of each active node and hence a greater number of requests can be expected. Since, in the static implementation, request (v, u, w) is processed by the thread to which u is assigned, some load imbalance is expected, roughly corresponding to the imbalance in the sum of the degrees of the active nodes. The benefit in workload reduction and work balance of the dynamic implementation is offset by the time spent in locking that is required to ensure safe write operations when the request lists are constructed, as well as some memory management overhead, when buckets are collapsed. While in both implementations, bucket updates are safely carried out without locking, the actual number of bucket updates depends on the number of times a new tentative distance results in a move to a new bucket. While both implementations will carry out more or less the same number of relaxations for a given value of Δ[1], each implementation has a different bottleneck. The major bottleneck of the dynamic implementation is write contention when request lists are being constructed and the memory management overhead, while the main issue with the static implementation is load imbalance.

These trade-offs are illustrated in Figure 2 for small diameter and large diameter graphs using a random edge weight distribution. The maintenance of the strictest request for each node offers great advantage on dense small diameter graphs, for instance on $\mathrm{G}(2^{21}, 128)$ the request list maintained by static Δ-stepping contains approximately 6.25 times more requests than the request list of dynamic Δ-stepping.

[1]The exact number will be impacted also by the order in which nodes within a bucket are processed

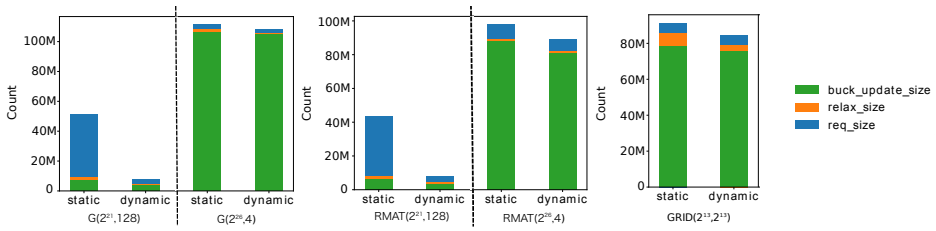


Figure 2: Workload of Dynamic and Static partitioning (Config A).

Table 1: Static vs. Dynamic Partitioning on various graph classes, Config A).

Graph	Imbalance %		Max Wait. (sec)	Time %
	Static	Dynamic	Dynamic	Dynamic
$G(2^{21}, 128)$	0.3753	0.0008	0.94	20.22
$G(2^{26}, 4)$	4.6082	0.0007	3.33	9.26
$RMAT(2^{21}, 128)$	8.3308	0.0020	0.93	21.92
$RMAT(2^{26}, 4)$	5.9243	0.0006	5.32	16.63
$GRID(2^{13}, 2^{13})$	4.6082	0.0007	1.65	6.24

On the other hand, sparse small diameter graphs demonstrate little advantage of the maintenance of strictest request lists. With lower density, the number of neighbours used to generate the request lists is smaller resulting in a smaller probability of multiple requests for the same node and hence showing little advantage of the strictest request list used in dynamic partitioning. In large diameter graphs the density is low in both cases with an average degree of 4 and therefore similar sizes of request lists can be observed in both implementations.

Next, we examine the imbalance[2] presented in both Δ-stepping implementations. In Table 1, it can be seen that the random distribution of nodes to threads generally results in more than 4.6% imbalance, while the imbalance of the dynamic implementation is generally less than 0.002%. However the locking mechanism of the dynamic implementation is a significant overhead, making it noncompetitive with the static version in most cases.

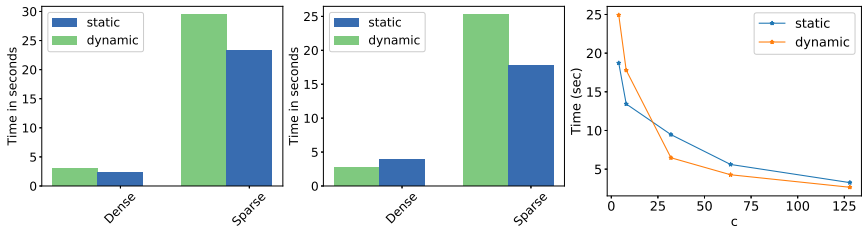
4.4 The Effect of Graph Density

In this experiment we further investigate the effect of graph density on both implementations. On both sparse small diameter graphs, static Δ-stepping is more than 1.35 times faster than dynamic Δ-stepping (see Figure 3). Recall that the parallelism of both Δ-stepping implementations comes from the size of request list, which in turn depends on the neighbourhood size of nodes being processed. Even though, the imbalance of dynamic partitioning remains less than 0.0007% on sparse small diameter graphs, the percentage of time spent waiting for lock release can be as high as 16% on

[2]i.e. the imbalance in relaxations across the threads.

sparse power-law graphs allowing the static partitioning to outperform the dynamic implementation.

With increasing density, the highly skewed degree distribution present in power-law graphs leads to increased imbalance of 8.3% in static partitioning. Although the time spent in locking by the dynamic implementation is also increased to 21%, the number of generated requests is 5.6 times fewer than the static partitioning, essentially resulting in the dynamic implementation becoming 1.4 times faster than the static implementation. In fact, dynamic implementation outperforms the static partitioning on power-law graphs when density $c > 32$, where $c = \frac{m}{n}$ (Figure 3(c)).



(a) Random Graph. (b) Power-law Graph. (c) Random Graph: $c = \frac{m}{n}$

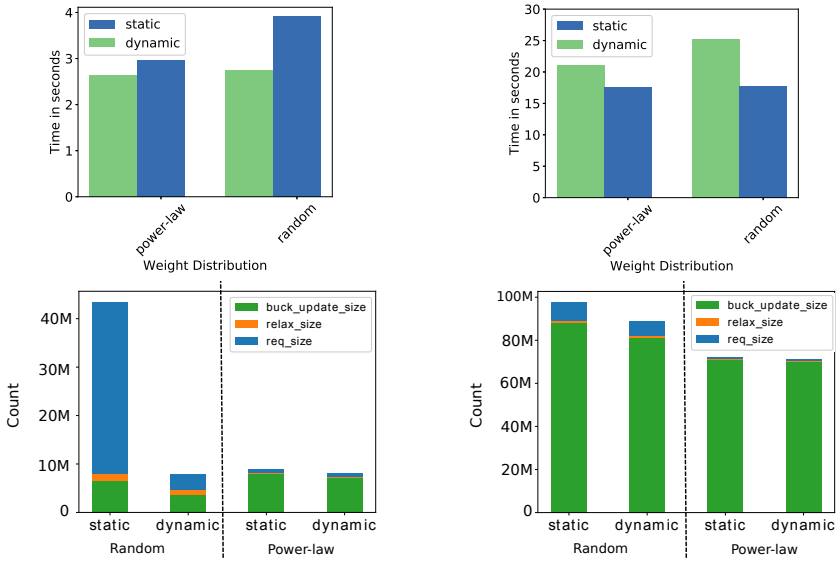
Figure 3: Effect of graph density (using 10 threads, Config A).

4.5 The Effect of Weight Distribution

Next we evaluate the effect of weight distribution on both implementations of parallel Δ-stepping algorithm. Due to space limitation, we focus on dense and sparse power-law graphs, as the highly skewed degree distribution presents a bigger challenge to static partitioning implementation.

On dense power-law graphs, the weight distribution affects the performance of parallel static partitioning, for instance it is 1.32 times slower with random weight distribution than with the power-law weight distribution (see Figure 4(a)). On the other hand, the performance of dynamic partitioning remains unchanged regardless of the weight distribution. In the power-law weight distribution most weights are above the average value and using $\Delta = 0.01$ classifies most edges as heavy. Recall that heavy edges are processed only once and therefore no requests will be reinserted into the same bucket. On the other hand with a random weight distribution and $\Delta = 0.01$, a good proportion of weights are light and therefore result in the reinsertion of nodes into the bucket currently being processed. Since the dynamic partitioning processes only the strictest requests, the static partitioning generates 5.5 times more requests with the random weight distribution. However with the power-law weight distribution, static partitioning generates only 1.2 times more requests than dynamic partitioning.

Contrary to dense RMAT graphs, in sparse RMAT graphs the weight distribution affects the running time of dynamic Δ-stepping, while the running time of static Δ-stepping remains unchanged (see Figure 4(b)). As discussed previously, although the request size generated for both implementations is bigger with random distribution,



(a) RMAT($2^{21}, c = 128$) (b) RMAT($2^{26}, c = 4$)

Figure 4: The effect of weight distribution (using 10 threads, Config A).

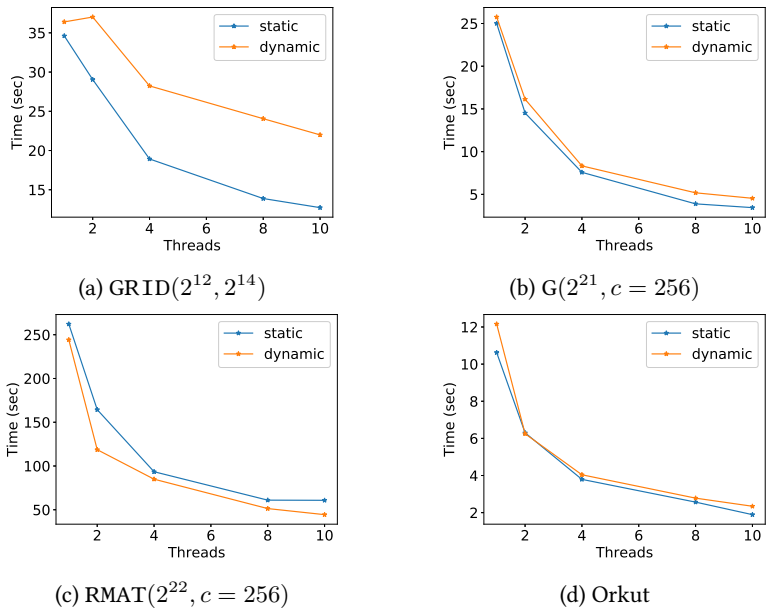
Table 2: The effect of weight distribution on RMAT($2^{26}, 4$) (using 10 threads, Config A).

Weight	% Runtime Spent in Locks	Imbalance %	
	Dynamic	Static	Dynamic
Power-law	7.74	5.9686	0.0007
Random	16.63	5.9243	0.0006

the imbalance remains the same for both implementations (see Table 2). However the increased request size in dynamic partitioning results in 2.15 times more time spent waiting for lock release with random weight distribution, essentially slowing down the parallel performance of dynamic Δ-stepping.

4.6 Parallel Performance of Δ-stepping Implementations

Next, we test the parallel performance of both Δ-stepping algorithm implementations on dense small-diameter graphs, on large-diameter grid graphs and on the real-world Orkut social network. For the synthetic large-diameter graphs, we choose graphs with more than 268 million edges. For the synthetic small-diameter graphs, we choose random graphs with more than 500 million edges and a graph density of $c = 256$. Finally, we choose RMAT graphs with more than 1 billion edges and the same graph density as the random graph. For comparison, the Orkut social network has over 100 million edges with density $c = 38.1$.



(a) GRID($2^{12}, 2^{14}$)

(b) G($2^{21}, c = 256$)

(c) RMAT($2^{22}, c = 256$)

(d) Orkut

Figure 5: Parallel Running Time of Dynamic and Static partitioning (Config A).

Similarly to the results observed by Madduri et. al [14] on the massively parallel Cray machine, neither of our parallel Δ-stepping implementations perform well on long-diameter graphs. For instance the static partitioning achieves a parallel efficiency[3] of 0.22 on a square grid graph (see Figure 5a for parallel running times). Fine grained assessment of the performance of the Δ-stepping algorithm presented in [14] revealed that long-diameter graphs result in a large number of phases with relatively small number of requests to be computed per phase. Therefore, even with larger values of Δ there is not enough work present in each phase to efficiently utilise all processors available.

From the point of view of available parallelism presented in the Δ-stepping algorithm, small-diameter graphs suit best in order to efficiently utilise the available parallelism. For instance the parallel efficiency for static partitioning scales up to 0.72 on random graphs using 10 threads (see Figure 5b). For comparison [14] reports parallel efficiency of 0.78 on a massively parallel machine using 40 processors and graphs consisting of over 1 billion edges. Our dynamic Δ-stepping is able to achieve 0.55 parallel efficiency on small-diameter RMAT graphs (see Figure 5c). Note that our implementation can run on any standard multicore machine, while the implementation in [14] is restricted to the specialised massively parallel MTA-2 Cray machine.

On the Orkut social network, both implementations achieve parallel efficiency higher than 0.5 (see Figure 5d). Recall that the parallel efficiency of Δ-stepping algorithm grows with the increased problem size and the Orkut social network is

[3]Parallel Efficiency $E = \frac{S_t}{t}$, where S_t refers to speed-up on t threads.

Table 3: Summary of best approaches to parallel Δ-stepping computation.

Graph class	Graph type	Δ-stepping implementation	Best Δ
Random	Sparse	Static	0.01
	Dense	Static	0.01
RMAT	Sparse	Static	0.01
	Dense	Dynamic	0.01
GRID	Square	Static	0.6
	Rectangle	Static	0.6
Real World	Orkut	Static	0.01

considerably smaller than the static graphs considered above. For comparison note that the Δ-stepping implementation provided in the Julienne framework [5] reports parallel efficiency of 0.26 on the Orkut social network when run on a 72-core shared-memory machine with 1TB of available memory. Similarly to the observation in [14], the parallel performance of the Δ-stepping implementation in the Julienne framework grows with increasing problem size.

4.7 User Guide

In this paper, we explored two different design choices in implementing the parallel Δ-stepping SSSP algorithm, namely static and dynamic parallel workload approaches. The major draw back of the dynamic approach is the need to implement locking mechanism in order to avoid race conditions while the major draw back of static approach is the load imbalance.

Table 3 summarises the best Δ-stepping implementation under a range of settings. Surprisingly the static approach performs the best in most cases apart from the dense RMAT graph, in which case the dynamic implementation yields the best performance. In terms of the best value of the input parameter Δ, small diameter graphs generally require small values such as $\Delta = 0.01$. On the other hand, large diameter graphs, such as grid graphs, require larger values such as $\Delta = 0.6$.

5 Conclusion

Parallel Δ-stepping algorithm is an effective parallel SSSP algorithm for multicore architectures, when applied to short-diameter graphs. Our detailed analysis of two different implementations shows the sensitivity of the algorithm to its control parameter, Δ, but also to design choices, such as the application of the strictest request optimisation. We have shown that a definitive implementation choice and configuration that is suitable for all graph types is not possible. However, the static implementation that avoids racing and memory contention works best in most scenarios considered.

From the point of view of scalability and parallel efficiency, we showed that both our implementations of Δ-stepping algorithm can handle input graphs consisting of more than 1 billion edges. Moreover our results show that the parallel efficiency of

our implementation of static partitioning is comparable to the parallel efficiency of the Δ-stepping implementation for massively parallel environments.

Acknowledgement

The work is supported by the Insight Centre for Data Analytics under Grant Number SFI/12/RC/2289_P2.

References

[1] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[2] Venkatesan T Chakaravarthy, Fabio Checconi, Fabrizio Petrini, and Yogish Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 889–901. IEEE, 2014.

[3] Chien-Chou Chen, Terry Hui-Ye Chiu, Yuh-Jzer Joung, and Shy Min Chen. An examination of online social networks properties with tie-strength. In *PACIS*, page 41, 2011.

[4] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 293–304, 2017.

[5] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304. ACM, 2017.

[6] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. URL: `http://gdzdoc.sub.uni-goettingen.de/sub/digbib/loader?did=D196313`.

[7] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel Boost graph library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, pages 219–248, 2006.

[8] Lester R Ford Jr. Network flow theory. Technical report, DTIC Document, 1956.

[9] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[10] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[11] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[12] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[14] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, volume 7, pages 23–35. SIAM, 2007.

[15] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. Dsmr: A parallel algorithm for single-source shortest path problem. In *Proceedings of the 2016 International Conference on Supercomputing*, page 32. ACM, 2016.

[16] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[17] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[18] Thap Panitanarak and Kamesh Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. In *CSC14: The Sixth SIAM Workshop on Combinatorial Scientific Computing*, page 60, 2014.

[19] Upasana Sridhar, Mark Blanco, Rahul Mayuranath, Daniele G Spampinato, Tze Meng Low, and Scott McMillan. Delta-stepping sssp: from vertices and edges to graphblas implementations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 241–250. IEEE, 2019.

[20] Sriram Srinivasan, Sara Riazi, Boyana Norris, Sajal K Das, and Sanjukta Bhowmick. A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 245–254. IEEE, 2018.

[21] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 265–266. ACM, 2015.

[22] Weidong Zhang, Lei Zhang, and Yifeng Chen. Asynchronous parallel dijkstra's algorithm on intel xeon phi processor. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 337–357. Springer, 2018.