Tiled Linear Algebra a System for Parallel Graph Algorithms

Saeed Maleki(✉), G. Carl Evans, and David A. Padua

Department of Computer Science, University of Illinois at Urbana-Champaign,
201 North Goodwin Avenue, Urbana, IL 61801-2302, USA
{maleki1,gcevans,padua}@illinois.edu

Abstract. High performance parallel kernels for solving graph problems are complex and difficult to write. Some systems have been developed to facilitate the implementation of these kernels but the code they produce does not always perform as well as custom software. In this space, we propose Tiled Linear Algebra (TLA), a multi-level system based on linear algebra but with explicit parallel extensions. Programs can be first written in a conventional manner using linear algebra and then tuned for parallel performance using our extension. This separation allows programmers with different expertise to focus on their strengths with writing original codes that can then be tuned by parallel experts.

This paper presents the background on using linear algebra to express graph algorithms and describes the extensions TLA provides to implement their parallel versions. The key extensions supported by TLA are: data distribution, partial computation, delaying updates, and communication. With these extensions to the traditional linear algebra operators, we could produce linear algebra based versions of several problems including single source shortest path that should preform close to custom implementations. We present results on several single source shortest path algorithms to demonstrate the features of TLA.

1 Introduction

In recent years, the importance of graph algorithms has been on the rise. While graph algorithms have always been a part of computer science, graph analytics have become increasingly important in the recent past. Graph analytics is used for the analysis of large network systems, capturing the interactions in social networks, natural language analysis, and cyber-security algorithms. Furthermore, in many cases, the size of the graphs to be analyzed has grown and continues to grow. For example, Facebook as a social network graph (with users as vertices and friendships as edges) has grown from 1 million to 845 million users in 7 years (2004–2011). Analysis of graphs of this size requires large parallel machines whose programming is a complex task due to correctness and performance.

Linear algebra operators can represent many graph algorithms in a concise and clear manner as discussed in [3,12]. The use of this notation allows for rapid development of complex algorithms. Today, the power and flexibility of using

J. Brodman and P. Tu (Eds.): LCPC 2014, LNCS 8967, pp. 116–130, 2015.
DOI: 10.1007/978-3-319-17473-0_8

linear algebra primitives comes with drawbacks. Standard sparse linear algebra kernels do not always take advantage of the structure of the graphs or parameters of the target machine and, as a result, fall short of the performance of custom implementations. To address this, we propose *Tiled Linear Algebra (TLA)* that is a multi-level parallel system with high-level linear algebra structure. In TLA, linear algebra primitives are used to construct a correct program and then performance features controlled by "knobs" are used to tune the kernels. These features include controlling distribution and communication frequency.

We organize this paper as follows. Section 2 describes the use of linear algebra for graph algorithms and Sect. 3 describes our extensions. We describe the algorithms to solve the Single Source Shortest Path (SSSP) problem and our experiments in Sect. 4. We wrap up with a discussion of related work in Sect. 5 and our conclusions in Sect. 6.

2 Graph Algorithms Using Linear Algebra

Using linear algebra as an abstraction for programming parallel graph algorithms is not new. CombBLAS [3] is perhaps the best known system using this approach. This section overviews linear algebra for representing graph algorithms.

There is a correspondence between a graph and a matrix. A graph $G = (V, E)$ with n vertices (set V) and m edges (set E) can be represented by its adjacency matrix A which is an $n \times n$ matrix such that $A(i, j) = 1$ if there is an edge e_{ij} from vertex v_i to vertex v_j and $= 0$ otherwise. This allows for directed and undirected graphs and can be extended to weighted graphs by using the weight rather than 1 to represent an edge. This representation is at the core of using linear algebra to describe graph algorithms. It should be noted that typically this matrix will be sparse and performance efficiency will depend, just as in conventional linear algebra, on how sparsity is handled.

Reachability Example: Reachability is the problem of finding all the reachable vertices in a directed graph $G = (V, E)$ from a source vertex $s \in V$. More formally, $Reach(G, s) = \{v \in V | \exists v_1, v_2, \ldots, v_k \in V, v_i v_{i+1} \in E, v_1 = s, v_k = v\}$. There exists a duality between reachability and matrix vector multiplication. Consider a vector r with $|V|$ elements (i.e. one element per vertex of the graph) with values $r(s) = 1$ and 0 everywhere else and the adjacency matrix A of graph G. All neighbors of s reachable in 1 step correspond to the non-zero entries of the vector $A^T \cdot r$. Consider Fig. 1 which shows a graph with its transposed adjacency matrix, A^T, with non-zeros shown as dots. Vertex 7 is the source vertex, s, of the reachability problem and as just mentioned, r has only one non-zero element (represented by the dot in position 7 for s). The result of the matrix vector multiplication will produce a vector r' with non-zeros represented by dots which corresponds to the non-zeros in the matrix shown by unfilled dots in the figure. $r' = A^T \cdot r$ computes the vertices that can be reached from 7 by traversing one edge (shown by dotted arrows in the figure). $r'' = r + r' = r + A^T \cdot r$ represents all vertices that can be reached in 1 or fewer (0) edge traversals. In general,

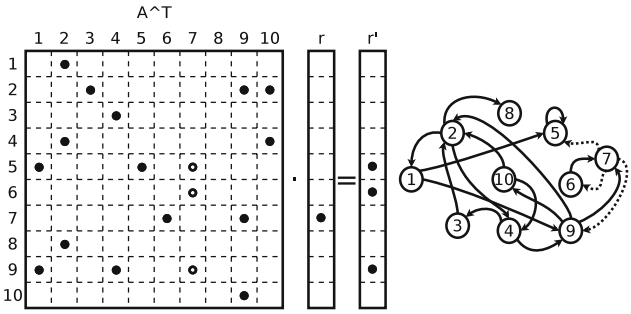


Fig. 1. Matrix-vector multiplication for reachability.

if r_0 includes 7 as a reachable vertex, an iterative matrix-vector multiplication $r_{i+1} = r_i + A \cdot r_i$ will find all of the vertices which are in $i+1$ or fewer edge traversals. The algorithm would terminate when a fixed point has been reached which in this case means that r_{i+1} and r_i has non-zeros in the same positions. These non-zeros in the final vector corresponds to all vertices reachable in any number of edge traversals. Note that, the elements of r_i could have different (positive) values and these values do not have a clear meaning. However, if we had a different algebra and replaced 1 and 0 by *true* and $false$, regular multiplication by $\wedge$, and regular addition by $\vee$, the result would have been the same except that all the non-zeros would all have been *true*.

A **semiring** is a five-tuple $(D, \oplus, \otimes, \mathbb{0}, \mathbb{1})$, where D is the set of elements of the semiring and D is closed under $\oplus$ and $\otimes$. $\mathbb{1} \in D$ is the identity for $\otimes$ which means that $\forall x \in D : x \otimes \mathbb{1} = \mathbb{1} \otimes x = x$ and $\mathbb{0} \in D$ is the identity for $\oplus$ which means that $\forall x \in D : x \oplus \mathbb{0} = \mathbb{0} \oplus x = x$. Also, $\mathbb{0}$ nullifies any elements of D with $\otimes$: $\forall x \in D : x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$. Given two matrices, $A_{l,m}$ and $B_{m,n}$, with elements from a semiring, D, their product is denoted $A \odot B$ and results in an $l \times n$ matrix defined such that

$$(A \odot B)(i,j) = A(i,1) \times B(1,j) + A(i,2) \times B(2,j) + ... + A(i,m) \times B(m,j)$$

In this notation, the semiring $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$ with the real numbers extended with ∞ as the domain, min as the additive operation $\oplus$, and $+$ as the multiplicative operation $\otimes$ is called the **tropical** semiring. Tropical semiring is specifically useful for computing single source shortest path from a source vertex to every other vertex in a graph which is discussed in more details below. The other useful semirings for other algorithms are the real field $(\mathbb{R}, +, \times, 0, 1)$ for page-rank computation or the boolean semiring $(\{0,1\}, \vee, \wedge, 0, 1)$ which, as mentioned in the example above, is a natural algebra for reachability problems.

A directed and weighted graph $G = (V, E)$ can be represented by its adjacency matrix A_G of size $n \times n$ where $|V| = n$ and whose values are $A_G(i,j) = weight(v_i v_j)$ if $v_i v_j \in E$ and $A_G(i,j) = \infty$ otherwise. That is, the elements of A_G are from the tropical semiring. A_G is a sparse matrix where the sparsity comes from the $\mathbb{0} = \infty$ elements in the matrix which represent the non-existent

edges in the graph. Let d be an $n \times 1$ vector where its i^{th} element, $d(i)$ is the distance to v_i, then $d' = A_G^T \odot d$ would also be a distance vector where $d'(i) = \min_{v_j v_i \in E(G)}(d(j) + weight(v_j v_i))$. $d'' = d \oplus A_G^T \odot d = d \oplus d'$ is another distance vector where in the computation of $d''(i)$, $\forall j : d(j) + w(v_j v_i)$ are considered as well $d(i)$ itself. This is equivalent to first, computing a new distance $d'(i)$ for each vertex v_i considering distance d of its incoming neighbors and the weight of corresponding edge $(\forall j : d(v_j) + w(v_j v_i))$ and second, comparing this new $d'(i)$ distance with $d(i)$ and setting $d''(i)$ to the smaller one.

Using adjacency matrix representation, one can express algorithms to find the shortest path using matrix-vector product in tropical semiring. For example, assuming that d_0 is a distance vector where $d_0(s) = 0$ and $\forall v \in V(G) \backslash \{s\}$: $d_0(v) = \infty$. At step i, the well known Bellman-Ford algorithm computes $d_{i+1} = d_i \oplus A_G^T \odot d_i$ and it iterates for $|V(G)| - 1$ times. In Sect. 3.2, we will explain how linear algebra can be used to express other algorithms.

3 Tiled Linear Algebra

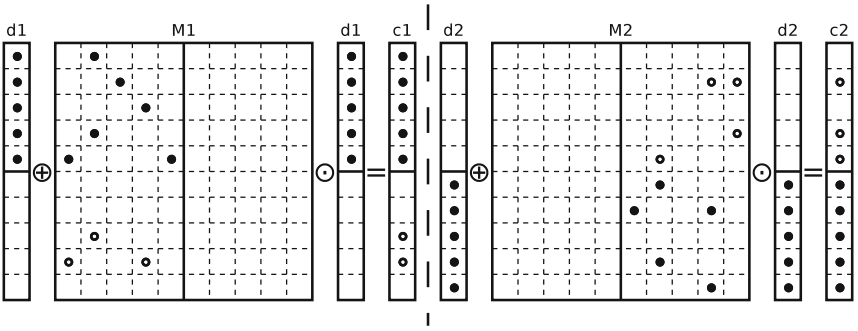
One of the most important aspects of linear algebra for graph algorithms is that the adjacency matrix of a graph G, A_G, is sparse and the system needs to use sparse algorithms. Otherwise, for example, a matrix-vector multiplication will require $O(|V|^2)$ operations instead of $O(|E|)$. Furthermore, different ways of representing a sparse matrix can impact the performance. Therefore, we believe that it is important for the programmer to have control over the representation.

Papers [3,8] discuss how graph algorithms can be represented in terms of matrix operations on different semirings. However, there are many ways to parallelize a matrix operation. In particular, the parallelization of matrix operations can be represented using *tiling* which partitions an array into subarrays by dividing each of the dimensions of the original array into segments. Each tile is assigned to a processor which will be responsible for the values of that tile. This assignment is done by a mapping function from tiles to processors.

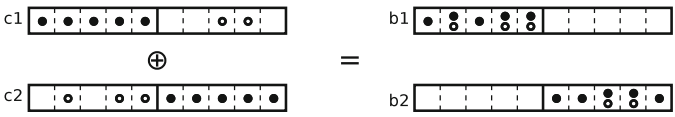
In our notation, we tile and assign tiles to processors at the same time. Let's say matrix A is read from file `input.txt` (which, for example, contains the edge list of a graph) and we want to divide the rows and columns into two segments. We will use command `A = ReadAndTile('input.txt', 2, 2, f);` to read from the file and tile it accordingly. `f` is a function that assigns each tile to a processor. Therefore, there are four tiles which we denote by $A_{1,1}$, $A_{1,2}$, $A_{2,1}$ and $A_{2,2}$. Each tile is conceived as having the size equal to the whole matrix, but contain non-$\mathbb{0}$ only in the regions associated with the tile. Therefore, $A_{1,1} \oplus A_{1,2} \oplus A_{2,1} \oplus A_{2,2} = A$. To create a vector of size $n \times 1$ and tile it into 2 segments, we will use command `v = CreateArray(n,2,f2);` where `f2` is another mapping function. Even though we explicitly ask the user for a tiling pattern, we do not explicitly use the tiling when representing operations.

3.1 Delaying Updates

As is well known, we can use the tiling to control where each component of the computation occurs and how the processors communicate. In regular parallel



(a) Saving off-tile values locally to be communicated later.



(b) Communicating off-tile values.

Fig. 2. Sparse matrix vector multiplication with delaying updates.

linear algebra, updates need to be visible by all the processors as soon as they occur. However, the updates can be postponed as, for example, is done in asynchronous algorithms [1,9].

Let A_G be the adjacency matrix of G and $M = A_G^T$. As discussed above, if d is a distance vector for the vertices of G, $c = d \oplus M \odot d$ will be another distance vector with distances updated by traversing 0 or 1 edges. Figure 2a represents the computation of $c = d \oplus M \odot d$. Let M be tiled 1×2 and $f(x, y) = y$ be the tile to processor mapping function such that tile $(1, 1)$ is assigned to processor p_1 and tile $(1, 2)$ is assigned to processor p_2. Also assume that d and c are vectors of size $n \times 1$ ($n = |V(G)|$ which is 10 in Fig. 2a) and they are tiled 2×1 and $f_2(x, y) = x$ is their mapping function. Figure 2a shows these tilings and distributions of M, d and c. The dots in the Figure represent the non-zeros. Processor 1 stores `M1`, `d1` and `c1` and processor 2 stores `M2`, `d2` and `c2`. Notice that `M1` and `M2` are each the size of the original matrix `M`, but with zeros outside the tile each of them represents. `d1` and `d2` have the same property. On the other hand, `c1` and `c2` are the size of the original vector `c` but there are non-zeros outside of the tiles that they are representing shown by unfilled dots.

It is easy to see that $c = d \oplus M \odot d = (d_1 \oplus M_{1,1} \odot d_1) \oplus (d_2 \oplus M_{1,2} \odot d_2)$. Therefore, without any communication, p_1 can compute $d_1 \oplus M_{1,1} \odot d_1$ and p_2 can compute $d_2 \oplus M_{2,1} \odot d_2$. However, since c_1 and c_2 have non-zeros everywhere, it is necessary to do a global computation to prepare for the next iteration. The value of d in the next iteration is $c = (d_1 \oplus M_{1,1} \odot d_1) \oplus (d_2 \oplus M_{1,2} \odot d_2)$ which requires adding c_1 and c_2.

To save communication time, we assign c_1 to d_1 and c_2 to d_2 before going to the next iteration. In this way, we do not carry out a global reduction. In other

words, we do not add c_1 to c_2 to get c. Instead, the values in the second tile of c_1 and the first tile of c_2 continue accumulating separately. At some point in time a global reduction is performed. As discussed below, by avoiding numerous global reductions, the performance of the algorithm improves. Postponing the reduction as just described is useful in the cases where it is not necessary to communicate the computed values immediately. This is the case of the reachability problem in which a processor can compute multiple iterations locally and find more reachable vertices before it communicates the remote reachable vertices.

To be able to postpone updates using our notation, we introduce $\leftarrow$ as an assignment operand which computes the linear algebra operation using only local values on the processor. In the case where an element on the left hand side of $\leftarrow$ is assigned to a different process, that value is only updated locally. For example, `c <`*`andthenaglobal`*`- d+M*d` will perform the local computation and will produce values that go to another processor. These values are the unfilled dots shown in Fig. 2a for both processors involved.

For communicating the values saved locally to the processor which owns the value, we introduce the $\oplus =$ operation which communicates all of the saved values to the owner processor where they are accumulated to the local copies of the elements using $\oplus$ as Fig. 2b depicts it. In the figure, `b1` and `b2` are the values assumed by `c1` and `c2` in the next iteration. After the owner processors update their values, communicated values become $\mathbb{0}$.

3.2 Partial Computation

In the simple version of reachability described in Sect. 2, all the reached vertices are processed in each iteration. Processing a vertex v in this problem means that marking all neighbors of v as reachable. This is clearly suboptimal since after one iteration all neighbors of a vertex have been reached. To improve upon this, we limit the processing to only vertices who were reached for the first time in the previous iteration. To be able to support this feature, we propose using mask vectors with $\mathbb{0}$ representing false and $\mathbb{1}$ representing true from the boolean semiring. Mask vectors are not different from other vectors except for the purpose they are used. A mask vector is used with *element-wise multiplication* represented by the operator "$\otimes$.". If a and b are two vectors in a semiring, $a \otimes .b$ is another vector where $(a \otimes .b)(i) = a(i) \otimes b(i)$. Now if b is a mask vector, $a \otimes .b$ will be a sparse subvector of a with some elements set to $\mathbb{0}$.

Note that mask vectors are used to avoid unnecessary computation. Therefore, the system should be aware of the fact that a vector can be sparse. For example, in the case of $A \odot v$ where A is a sparse matrix and v is a sparse vector, only a corresponding columns of non-zero elements of v should be considered.

Partial computation is important for many algorithms where an update to a vertex will only affect a few neighboring vertices. For example, in the case of SSSP, if a vertex is updated, an algorithm needs only to update its outgoing neighbors. Another example is the PageRank problem where if an update to a vertex is higher than the threshold, it only affects the neighbors of that vertex.

4 Single Source Shortest Path

The Single-Source Shortest Path (SSSP) problem finds the shortest distance from a source vertex to every other vertex in a graph. An instance of the problem is denoted by (G, w, s) where $G = (V, E)$ is a graph with the set of vertices, V, and the set of edges, E, and a source vertex, $s \in V$. Each edge $vu \in E$ has a tail, $v \in V$, and a head, $u \in V$. The map $w : E \rightarrow \mathbb{R}$ associates a weight for each edge $vu \in E$. Vertex $s \in V$ is the source whose distances to all other vertices is desired. This section assumes that all the weights are positive. The shortest distance from s to v is denoted by $d(s, v)$.

There are several algorithms to solve SSSP which we will discuss about how some of them can be expressed in TLA. In spite of their differences, the main operation in these algorithms is matrix-vector multiplication in tropical semiring.

4.1 Algorithms

The four best-known algorithms to solve SSSP problem are: Dijkstra [6], Bellman-Ford [2], Chaotic-Relaxation [5], and Δ-Stepping [13]. Bellman-Ford is the only algorithm that is capable of solving SSSP with negative edge weights but this aspect will not be discussed further in this paper and we assume all the graphs have positive edge weights. The basic operation that all four algorithm use is **relaxation** which takes an edge vu and **checks** if $d(s, v) + w(vu) < d(s, u)$ where d is not necessarily the final minimum distance but the shortest path "so-far" in the computation. If the check condition is true, $d(s, u)$ is updated with $d(s, v)+w(vu)$. The difference between the algorithms mentioned above is in the **order** in which relaxations are applied which directly affects the amount of work each algorithm performs. We measure the amount of work done by each algorithm in terms of the number of checks (for $d(s, v)+w(vy) < d(s, u)$) which is equal to the total number of relaxations.

Below, we assume that $G = (V, E)$ is the input graph and that the transpose of its adjacency matrix is M. Initially, in all four algorithm $d(s, v) = \infty$ for all $v \in V - \{s\}$ and $d(s, s) = 0$. The values of d are stored in a tiled vector. Next, we will explain each algorithm and express them in TLA.

Bellman-Ford: Our implementation of the Bellman-Ford algorithm in TLA (shown in Fig. 3) relaxes all the vertices during each iteration. The algorithm terminates after $|V| - 1$ iterations. As we discussed in Sect. 2, `d+M*d` which corresponds to the formula $d \oplus M \odot d$ computes a new distance vector for G by relaxing all the edges. Parallelizing this algorithm is straightforward by partitioning the vertices and having each processor relax one or more of the resulting subsets. As shown in Fig. 3, in every iteration of the for loop, each processor relaxes its own portion of edges assignment ("`<-`" in line 2) and then a global communication (operation `+=`) sends remote updates (line 3).

```
1  for (int i = 0; i < n-1; i++){
2          d <- d + M*d;
3          d += d;  }
```

Fig. 3. Bellman-Ford algorithm main loop using TLA.

Chaotic-Relaxation: The Chaotic-Relaxation algorithm is the same as the Bellman-Ford algorithm except that at each iteration, it only relaxes those vertices which changed distances in the previous iteration. TLA code for this algorithm is shown in Fig. 4. This algorithm is a small improvement over Bellman-Ford obtained by avoiding redundant relaxations. To this end, we use the mask vector `r` which has one element for each vertex and is used to keep track of the vertices whose distances did not change in the previous iteration. The vector `r` is initialized so that it is false everywhere except for the position corresponding the vertex s. The element-wise multiplication (`*.`) on line 2 prunes elements which did not change their distance and sparse matrix-sparse vector multiplication (`M*(d*.r)`) takes advantage of it.

In the following algorithms, the scalar `notDone` (replicated across processors) is used to decide when to terminate the algorithm. The last iteration is that in which $d(s, v)$ remain constant for all $v \in V$. In other words, the algorithm is finished when `r` (set on line 4) is all `0` (all false) for each tile. Note that `r <- b != d` sets `r(i)` to `1` if `b(i) != d(i)` and sets it to `0` otherwise. Finding out when `r` is all `0` is done by the local reduction `notDone <- any(r)` on line 6 followed by the global reduction `notDone += notDone` on line 7. If `notDone` is `0`, it means that there was at least one `0` in one of the tiles of `r`.

```
1  do {
2          c <- d + M*(d*.r);
3          b += c;
4          r <- (b != d);
5          d <- b;
6          notDone <- any(r);
7          notDone += notDone; /* global reduction */
8  } while (notDone != 0);
```

Fig. 4. Chaotic-Relaxation main loop using TLA.

Dijkstra's Algorithm: Dijkstra's algorithm is the fastest sequential SSSP algorithm. At each iteration in this algorithm, only one vertex is **processed** which is the vertex with minimum distance among the vertices not processed before. Processing a vertex means relaxing all of its outgoing edges. The TLA code for this algorithm is shown in Fig. 5.

The major difference between this algorithm and the previous two algorithms is that the matrix-vector multiplication for this algorithm (line 3) is with a vector which has only one non-$\mathbb{0}$ element in it (`d *. r`) and that element corresponds to the vertex with the minimum distance among the unprocessed vertices. Finding the element with minimum distance is done with the help of vector `m` which keeps track of the processed vertices (line 5). Lines 6 and 7 find the index and

```
1  m <- 0;
2  for (int i = 0; i < n; i++){
3          d <- d + M*(d*.r);
4          d += d;
5          m <- m+r;
6          ind <- argmin(d*.(!m));
7          minVal <- min(d*.(!m));
8          globalMinVal += minVal; /* global reduction */
9          r <- 0;
10         if (minVal == globalMinVal)
11                 r(ind) = 1;       }
```

Fig. 5. Dijkstra's algorithm main loop using TLA.

the minimum distance vertex in each processor locally and it is communicated globally on line 8. Lines 10 and 11 determines in each processor if the local minimum value is equal to the global minimum value and, if so, sets `r` accordingly. The algorithm terminates after all n vertices are processed.

***Δ*-Stepping:** Δ-Stepping is another SSSP algorithm which is half way between the Dijkstra's and the Chaotic relaxation algorithms. Δ-Stepping processes a bucket of vertices in each iteration not just one as in the case of Dijkstra's algorithm nor all vertices as in the case of the Chaotic relaxation algorithm. Δ-Stepping distributes vertices into buckets $\{b_0, b_1, b_2, \dots\}$ where bucket $b_i = \{v | \Delta i \leq d(v) < \Delta(i+1)\}$. Note that $d(v)$ is dynamic and as the algorithm advances, it changes value. Therefore, the algorithm should update the bucket for each vertex that is updated. In iteration i, Δ-Stepping only considers vertices from bucket b_i. Since relaxing outgoing edges of a vertex from b_i may add more vertices in it, this process has to continue until there are no more vertices in b_i whose outgoing edges are not relaxed. The algorithm terminates when there are no vertices to process. Note that if b_i is completely processed and the algorithm advances to b_{i+1}, it will never again need to process vertices from b_i since all the weights are positive.

Our version of Δ-Stepping in TLA code is shown in Fig. 6. It is similar to the code in Fig. 4 with a few modifications. First, the main matrix-vector multiplication for relaxation is different which is shown on line 2. There are two mask vectors for this algorithm: (1) `r` which is similar to `r` from in Fig. 4 and it holds the vertices that needs to be processed; (2) `bucket` which is set on line 6 and contains the vertices that belong to bucket b_i. Initially, it contains vertices that are in the range of $[0 \dots \Delta)$. To find out whether there are more vertices in bucket b_i to relax, the scalar `notDoneBucket` is used on line 7. Similar to `notDone` scalar from Fig. 4, a reduction on array `bucket*.r` determines if `i` should be incremented (line 10) and `bucket` is set accordingly (line 11).

None of these 4 algorithms required delaying updates and all of them could have been done by fusing local computation with global communication. Next, we will describe our parallel SSSP algorithm which takes advantage of this feature.

Partially Asynchronous *Δ*-Stepping: This algorithm is similar to Δ-Stepping with the exception that Δ-Stepping algorithm is used locally only. In other words,

```
1  do {
2          c <- d + M*((d*.r)*.bucket);
3          c += c;
4          r <- (c != d);
5          d <- c;
6          bucket <- d >= i*Δ & d < (i+1)*Δ;
7          notDoneBucket <- any(bucket*.r);
8          notDoneBucket += notDoneBucket;
9          if (!notDoneBucket){
10                 i++;
11                 bucket <- d >= i*Δ & d < (i+1)*Δ;
12         }
13         notDone <- any(r);
14         notDone += notDone;
15 } while (notDone != 0);
```

Fig. 6. Δ-Stepping main loop using TLA.

each processor applies Δ-Stepping to process its own vertices. Local relaxation occurs for multiple iterations (for **pipeline** iterations) and after that a global communication exchanges the distance updates. The invariant from Δ-Stepping algorithm explained before does not hold in here; in a processor, a global distance update may add vertices for processing from lower buckets since other processors may have been processing vertices from lower buckets. Therefore, after a global update, each processor needs to start from the smallest bucket which has unprocessed vertices.

The TLA code for this algorithm is shown in Fig. 7. As described above, the algorithm applies a local Δ-Stepping for its vertices as shown in lines 2–10. The code in this loop is similar to the one from the original Δ-Stepping except that every computation is local (note that only `<-` is used). After this loop, a global update exchanges distances and updates mask vector `r` (line 12). After that, the variable `minDist` will be computed which is the minimum distance among all of the local unprocessed vertices. Line 15 finds the minimum bucket with unprocessed vertices in it. The rest of the code is similar to the other SSSP algorithms.

4.2 Performance Comparison

This Section compares the parallel performance of each of the SSSP algorithms described in Sect. 4.1. We will also how each feature of TLA affects the performance.

Figure 8 shows the parallel performance of Chaotic Relaxation, Dijkstra, Δ-Stepping and Partially Asynchronous Δ-Stepping. We excluded Bellman-Ford from this figure since it is significantly slower than the other four algorithms ($\sim 2000\times$). The X axis in this figure represents different number of processors and the Y axis is for running time. Each algorithms is specified by its color: blue for Chaotic Relaxation, gray for Dijkstra, red for Δ-Stepping and purple for Δ-Stepping with pipelining (Partially Asynchronous Δ-Stepping). The orange color is for the communication cost. Therefore, each group of 4 bars represents

```
do {
        for (int j = 0; j < pipeline; j++){
                b <- d + M*((d*.r)*.bucket);
                r <- (b != d);
                d <- b;
                bucket <- d >= i*Δ & d < (i+1)*Δ;
                notDoneBucket <- any(bucket*.r);
                if (!notDoneBucket){
                        i++;
                        bucket <- d >= i*Δ & d < (i+1)*Δ; }}
        b += d;
        r <- (b != d);
        d <- b;
        minDist <- min(d*.r);
        i = minDist / Δ;
        notDone <- any(r);
        notDone += notDone;
} while (notDone != 0);
```

Fig. 7. Partially Asynchronous Δ-Stepping algorithm in TLA.

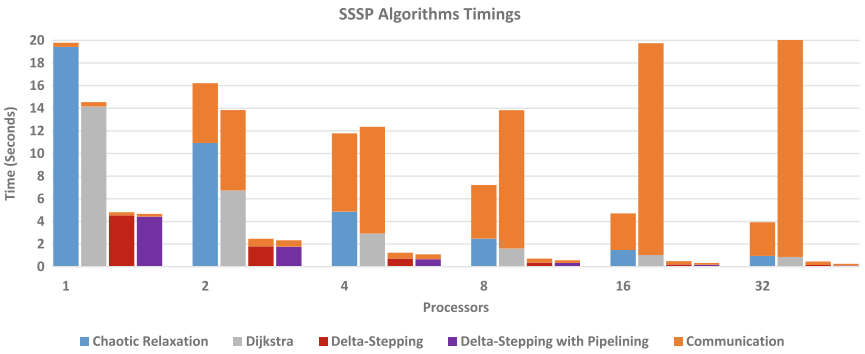


Fig. 8. SSSP algorithms performance comparison

the running time for each algorithm with one specific number of processors. The input graph is an R-MAT graph with $SCALE = 20$.

The fact that Bellman-Ford is significantly slower than Chaotic-Relaxation (order of 2000×) shows that how crucial is partial computation for SSSP algorithms. On the other hand, Chaotic Relaxation algorithm does not scale well because of the communication cost. But as it can be seen, just the blue portion of running time is scaling well. This is because the algorithm is massively parallel and the work is balanced well since each processor owns roughly the same number of edges.

Dijkstra algorithm in Fig. 8 has a better sequential performance than Chaotic relaxation but since it only processes one vertex at a time, the parallel performance is poor and most of the communication cost is just the idle time. However, Δ-Stepping is performing faster than both Dijkstra and Chaotic Relaxation and it is providing decent speed up. Δ-Stepping with pipelining is almost as fast as Δ-Stepping for 1, 2 and 4 processors. It is hard to see in Fig. 8 how they compare with higher number of processors. Therefore, Fig. 9 directly compares them.

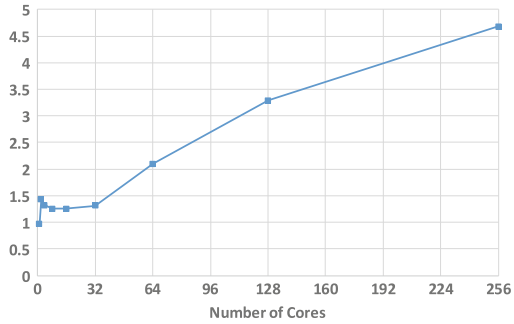


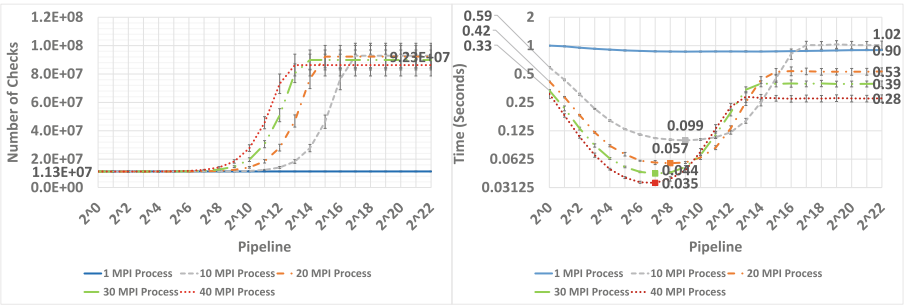
Fig. 9. Delaying updates speed up.

Figure 9 shows the speed up of Partially Asynchronous Δ-Stepping over Δ-Stepping for different number of processors. As it can be seen, up to 16 processors, it does not provide significant improvement but for larger number of processors, it is certainly effective which in this case is more than 4× faster. This shows the importance of delaying updates feature in TLA. Next Section, will study how pipeline factor itself can control the performance of our algorithm.

4.3 Delaying Updates Impact on Partially Asynchronous Δ-Stepping

Our Partially Asynchronous Δ-Stepping has two tunable parameters: the Δ factor and the *pipeline* factor. The best Δ value can be experimentally found and it will neither be affected by the number of processors used nor by the source vertex. On the other hand, the **pipeline** factor which impacts the total number of relaxations, has different best values for different number of processors. Again, each relaxation is a **check** to find whether we can reach a vertex with a shorter distance. We will use the number of checks as a measurement for the amount of work the algorithm does. `pipeline` is a variable as shown in Fig. 7 that controls the number of iterations of local Δ-Stepping between each global communication for updating distances. In other words, it controls the frequency of global communication which impacts the performance in two ways. If the intercommunication interval is too long, a processor almost always computes distances of paths that go through local vertices. This may result in useless checks and updates for those vertices whose shortest path goes through vertices owned by different processors. Thus, it is better for the exchange of relaxation requests not to be too infrequent so that vertices can reach their final distance sooner. However, updating too frequently may add significant overhead because of the initial cost of each communication.

Figure 10a and b show the execution of Partially Asynchronous Δ-Stepping algorithm of Fig. 7 with an R-MAT graph [4] with $SCALE = 20, a = 0.55, b = 0.1, c = 0.1, d = 0.25, M = 8 \times N$ and $\Delta = 2^{16}$ on a shared memory machine with 40 cores as the value of the pipeline factor changes. Figure 10a shows the



(a) Number of checks (b) Running time

Fig. 10. Pipeline factor impact on the number of checks and the running time.

number of checks and Fig. 10b shows the running times. Each line corresponds to a different number of processor. We use a value of $\Delta = 2^{16}$, as it results in the lowest execution time independent of the number of processors. Numbers are computed by averaging the running times of the algorithm for 16 randomly chosen source vertices. All axes are logarithmic. The error bars represent the standard deviation. The marked points on the plot in Fig. 10b show the best performing value of the pipeline factor.

The number of checks, increases with the pipeline factor, however, it is almost constant at first and then increases drastically. The left most points represent frequent global updates. On the other hand, a high pipeline factor has the same effect as if there were no pipelining at all. With low pipeline factor the numbers of checks is the same for all number of processors. As the pipeline factor increases, there is a factor of ~ 8 increase in the number of checks for all number of processes (except, of course, for the case of 1 processor where it remains constant). As it can be seen in Fig. 10b, low values for pipeline factor do not deliver the best performance because for these values, the algorithm sends many short messages. In fact, for low values, the algorithm is $6\times$ slower than the optimal. On the other hand, high values of the pipeline factor slows down the algorithm because of the large number of checks (relaxations). In fact, for high pipeline values the algorithm is $\sim 8\times$ slower than the best execution times. This tracks the factor of 8 increase in the number of checks. The point at which the pipeline factor delivers the best performance is different for each number of processors. The best pipeline factors for different graphs are different but the optimum is never at too low or too high values. This suggests that using the idea of delaying updates is effective and it increases the performance by multiple factors.

5 Related Work

Our work is most similar to the combinatorial BLAS [3] but we differ in that where they handle the parallelism entirely under the abstraction of linear algebra but we make the parallelism and distribution explicit. In TLA, the programmer

can write the same type of program that was expressible in the combinatorial BLAS since we are both based on the linear algebra but in TLA the programmer can directly control distribution, communication, and grain size. These features allow an expert programmer to take code and tune it to take advantage of hardware and algorithmic features that are not exposed in a system.

Another model of parallel graph computation is the vertex programing model. This model is used by PowerGraph [7], Pregl [11], and GraphLab [10]. In this model, the programmer thinks of graph algorithms as running in parallel and interaction on the edges between vertices. Also, the very fine grained work is aggregated by the runtime system and not under programmer control. It also lacks the ability to restrict computation when available under programmer control. These limitations would prevent expressing algorithms such as Δ-Stepping where the work does flow directly from neighboring vertices.

6 Conclusion

In this paper, we presented TLA, a system for graph algorithms using linear algebra. We have demonstrated the express-ability of our library with implementations of several SSSP algorithms. Our experiments have shown that by using the extensions in TLA, we achieve performance comparable to custom implementations of the same algorithms.

In the future, we intend to develop TLA in to a full featured library with more included semirings as well as support for user defined ones. We believe that as we implement more algorithms with TLA, we will find more extensions to the underlaying liner algebra. Extensions that we have considered included asynchronous messaging, control over updating, and support for dynamic graphs.

References

1. Arnal, J., Migallón, V., Penadés, J.: Synchronous and asynchronous parallel algorithms with overlap for almost linear systems. In: Hernández, V., Palma, J.M.L.M., Dongarra, J. (eds.) VECPAR 1998. LNCS, vol. 1573, pp. 142–155. Springer, Heidelberg (1999)
2. Bellman, R.: On a routing problem. Q. Appl. Math. **16**, 87–90 (1958)
3. Buluç, A., Gilbert, J.R.: The combinatorial blas: design, implementation, and applications. Int. J. High Perform. Comput. Appl. **25**(4), 496–509 (2011)
4. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: a recursive model for graph mining. In: SDM (2004)
5. Chazan, D., Miranker, W.: Chaotic relaxation. Linear Algebra Appl. **2**(2), 199–222 (1969)
6. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1), 269–271 (1959)
7. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI 2012, pp. 17–30, USENIX Association, Berkeley(2012)

8. Kepner, J., Gilbert, J. (eds.): Graph Algorithms in the Language of Linear Algebra. Society for Industrial and Applied Mathematics, Philadelphia (2011)
9. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 235–244, ACM, New York (2007)
10. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
11. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 135–146, ACM, New York (2010)
12. Mattson, T., Bader, D.A., Berry, J.W., Buluç, A., Dongarra, J., Faloutsos, C., Feo, J., Gilbert, J.R., Gonzalez, J., Hendrickson, B., Kepner, J., Leiserson, C.E., Lumsdaine, A., Padua, D.A., Poole, S., Reinhardt, S., Stonebraker, M., Wallach, S., Yoo, A.: Standards for graph algorithm primitives. In: HPEC, pp. 1–2 (2013)
13. Meyer, U., Sanders, P.: Delta-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)