A Parallel 2/3-Approximation Algorithm for Vertex-Weighted Matching[*]

Ahmed Al-Herz [†] Alex Pothen[‡]

Abstract

We consider the maximum vertex-weighted matching problem (MVM), in which non-negative weights are assigned to the vertices of a graph, and the weight of a matching is the sum of the weights of the matched vertices. Although exact algorithms for MVM are faster than exact algorithms for the maximum edge-weighted matching problem, there are graphs on which these exact algorithms could take hundreds of hours. For a natural number k, we design a $k/(k+1)$-approximation algorithm for MVM on non-bipartite graphs that updates the matching along certain short paths in the graph: either augmenting paths of length at most $2k+1$ or weight-increasing paths of length at most $2k$. The choice of $k = 2$ leads to a 2/3-approximation algorithm that computes nearly optimal weights fast. This algorithm could be initialized with a 2/3-approximate maximum cardinality matching to reduce its runtime in practice. A 1/2-approximation algorithm may be obtained using $k = 1$, which is faster than the 2/3-approximation algorithm but it computes lower weights. The 2/3-approximation algorithm has time complexity $O(\Delta^2 m)$ while the time complexity of the 1/2-approximation algorithm is $O(\Delta m)$, where m is the number of edges and Δ is the maximum degree of a vertex. Results from our serial implementations show that on average the 1/2-approximation algorithm runs faster than the Suitor algorithm (currently the fastest 1/2-approximation algorithm) while the 2/3-approximation algorithm runs as fast as the Suitor algorithm but obtains higher weights for the matching.

One advantage of the proposed algorithms is that they are well-suited for parallel implementation since they can process vertices to match in any order. The 1/2- and 2/3-approximation algorithms have been implemented on a shared memory parallel computer, and both approximation algorithms obtain good speedups, while the former algorithm runs faster on average than the parallel Suitor algorithm. Care is needed to design the parallel algorithm to avoid cyclic waits, and we prove that it is live-lock free.

1 Introduction

We describe a parallel algorithm for computing a 2/3-approximation for the maximum vertex-weighted matching problem (MVM). Here we are given a graph $G = (V, E, \phi)$, where V is the set of vertices, E is the set of edges, and $\phi : V \mapsto R_{\geq 0}$ is a non-negative weight function on the vertices. A matching is a subset of vertex-disjoint edges, and in the MVM, we seek a matching M where the sum of the weights on the endpoints of M has the maximum value. Earlier we had designed a serial 2/3-approximation algorithms for the MVM problem in bipartite and non-bipartite graphs; however, these algorithms have little concurrency. In this paper we design a new $k/(k+1)$-approximation algorithm for MVM, where k is a natural number. With $k = 2$, it yields a 2/3-approximation algorithm that can be implemented on shared-memory parallel machines.

The MVM problem can be solved *exactly* in $O(\sqrt{|V|}|E| \log |V|)$ time; however since this algorithm uses recursion and is challenging to implement, we have designed a simpler ($O(|V||E|)$ time algorithm, first for bipartite graphs [3], and then for non-bipartite graphs [1]. The latter algorithm has the advantage that it can be easily modified to obtain approximation algorithms, and it is fairly simple to describe. It sorts the vertices in non-increasing order of weights, and then searches for augmenting paths from unmatched vertices in this order. For each unmatched vertex, the algorithm chooses an augmenting path that leads to a heaviest unmatched vertex it can reach. It then augments the matching using this path, and proceeds to the next unmatched vertex. We call this the Direct algorithm since each unmatched vertex is processed exactly once.

In earlier work, we had obtained a 2/3-approximation algorithm from the exact algorithm described above by restricting the lengths of augmenting paths to three (edges). While this algorithm is both elegant and easy to state, proving that it computes 2/3-approximation to the MVM problem is quite involved [1]. It is accomplished by a careful study of the properties of the augmenting paths, involving new concepts such as the origin and the terminus of an augmenting path, and a heaviest unmatched neighbor of a matched vertex. Restricting the augmenting path length to three does not lead to a similar approximation for the maximum edge weighted matching problem (MEM). MVM problems can be solved with MEM algorithms by computing edge weights as sums of the weights on the endpoints of every edge. However, we have shown that this can increase the run times of exact

[*]Research supported by NSF grant 1637534, DOE grant DE-SC00010205, and the ExaGraph project funded by DOE and NNSA.

[†]Computer Science Department, Purdue University, West Lafayette IN 47907 USA, aalherz@purdue.edu. Current address: Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, alherz@kfupm.edu.sa

[‡]Computer Science Department, Purdue University, West Lafayette IN 47907 USA, apothen@purdue.edu

MEM matching algorithms by several orders of magnitude over algorithms that solve the MVM problem directly [3]. We show here that approximation algorithms for MVM are faster than approximation algorithms for MEM.

Here we describe a new approach that computes a $k/(k+1)$-approximation algorithm for MVM, where k is a natural number less than $|V|$. Specializing this algorithm to $k = 2$ leads to a new 2/3-approximation algorithm, and with $k = 1$ we obtain a 1/2-approximation algorithm. The former algorithm computes better weights than the Suitor algorithm, while the latter algorithm is faster than the Suitor algorithm.

The MVM problem arises in the context of internet advertising [8], and an online algorithm with a competitive ratio $(1 - 1/e)$ is available for this problem. Mehta, the author of the survey cited, states that "this is the largest matching problem in the world, both in terms of money and the number of items matched." He asks for "a fast, simple, off-line approximation algorithm, when the data is big." Our work on MVM addresses this question, and parallel algorithms enable us to solve larger problems than would be possible on serial computers.

The remainder of this paper is organized as follows. Section 2 describes a more general k/(k+1)-approximation Iterative algorithm for MVM on non-bipartite graphs (here k is a natural number), proves the approximation ratio and establishes its time complexity. This algorithm could be specialized to a 2/3- or 1/2-approximation algorithm. Next, Section 3 describes a parallel version of the 2/3-approximation algorithm. Section 4 briefly discusses the exact and other approximation algorithms that we compare the new algorithms to. Results from sequential and parallel implementations of the Iterative algorithms are included in Section 5; so are comparisons with other $1/2-$, $(2/3 - \epsilon)-$ and 2/3-approximation algorithms for MVM and MEM. We conclude in the final Section 6.

2 New Iterative Approximation Algorithms

Recall that we denote a graph as $G = (V, E, \phi)$, and we let $|V| \equiv n$ and $|E| \equiv m$. Given a matching M, an alternating path has alternate edges that belong to the matching. An augmenting path is an alternating path of odd length (number of edges) that begins and ends with edges not in the matching. A weight-increasing alternating path is an alternating path of even length that begins with a non-matching edge and ends with a matching edge. If the unmatched terminal vertex has higher weight than the matched terminal vertex of the path, then by exchanging the matching and non-matching edges, we can increase the weight of the matching. The gain of an increasing path is the difference between the weights of its unmatched endpoint and its matched endpoint. A k-augmentation is an M-alternating path that has at most k edges not in the matching. Thus an M-augmenting path of length one or three, or a weight-increasing path of length two or four, is a 2-augmentation. We will assume that the reader is familiar with matching concepts and terminology; these may be found in books such as [12]. We also refer the reader to our earlier papers on MVM [1, 3] and the survey [11].

Now we will describe the $\frac{k}{k+1}$-Iterative algorithm (and a variant) for any natural number k. This algorithm starts with the empty matching M and iterates over all vertices in an arbitrary order. Although the algorithm could search for any $k-$augmentation, we will be more specific here. From an unmatched vertex u, the algorithm searches for an augmenting path of length at most $2k-1$ that reaches an unmatched vertex v; if such a vertex is found then it augments the matching and proceeds to the next unmatched vertex. If no such augmenting path is found, then it searches for an increasing path from u of length at most $2k$ with the highest gain (corresponding to a lightest matched vertex reached). If the algorithm finds an increasing path P then it updates the matching with the symmetric difference $M \oplus P$. The algorithm iterates until no unmatched vertices remain, or we fail to find augmenting or increasing paths of the specified lengths from them.

Note that a 2/3-approximation algorithm is obtained from the above algorithm by choosing $k = 2$, i.e., by restricting augmenting path lengths to at most three, and increasing path lengths to at most four. With $k = 1$, we obtain a 1/2-approximation algorithm.

A variant of this algorithm, the $\frac{k}{k+1}$-Init-Iter algorithm employs a $\frac{k}{k+1}$-approximate cardinality matching to initialize the $\frac{k}{k+1}$-Iter algorithm. The $\frac{k}{k+1}$-cardinality matching initialization algorithm starts with an empty matching and processes the vertices in arbitrary order. For each unmatched vertex u the algorithm searches for an augmenting path of length at most $2k-1$. If an augmenting path is found, then the matching is augmented by the path. If the search fails to find an augmenting path, we consider the next unmatched vertex. This phase terminates after one pass over the unmatched vertices.

Now we will prove that this algorithm computes a $\frac{k}{k+1}$-approximate matching for MVM. Let $\phi(A)$ denote the sum of the weights of the vertices in a set A; we also write $\phi(M)$ to denote the sum of weights of the matched vertices in a matching M.

Lemma 2.1. *Let M_{opt} be an optimal matching and M_A be an approximate matching computed by Algorithm 1*

Algorithm 1 $k/(k+1)$-Iter approximation algorithm.

```
1: procedure $k/(k+1)$-ITER($G = (V, E, \phi)$, $M$)
2:     do
3:         done = true;
4:         for all $u \in V$ do
5:             if $u$ is unmatched then
6:                 Search for an aug. path $P$ from $u$ s.t. $|P| \leq 2k - 1$;
7:                 if $P$ is found then
8:                     $M \leftarrow M \oplus P$; done = false;
9:                 else
10:                    Search for a highest gain increasing path $P'$ from $u$ s.t. $|P'| \leq 2k$;
11:                    if $P'$ is found then
12:                        $M \leftarrow M \oplus P'$; done = false;
13:                    end if
14:                end if
15:            end if
16:        end for
17:    while done = false
18: end procedure
```

in a graph $G = (V, E, \phi)$. For every M_A-unmatched vertex u that is matched in M_{opt}, there are k distinct M_A-matched vertices that are at least as heavy as u.

Proof. The symmetric difference of M_A and M_{opt} consists of vertices with degrees 0, 1 or 2. Hence this subgraph is a union of alternating paths and alternating cycles. The cycles contribute the same weight to the matchings M_{opt} and M_A, and hence we need consider only alternating paths. Each vertex $u \in U = V(M_{opt}) \setminus V(M_A)$ has degree one in the subgraph, and hence must be an end point of an alternating path such that its length is at least $2k$. At the termination of the approximation algorithm since there are no weight-increasing paths of length at most $2k$, the k vertices at distances of even lengths from u on the symmetric difference path must be at least as heavy as u. For if not, the algorithm should have reversed the increasing path of length at most $2k$, and u would have been matched. Hence the result is true if the alternating path has only one endpoint from U. If both endpoints of an alternating path belong to U, then it has length at least $2k+1$ since there no augmenting paths of length at most $2k-1$. Then each endpoint chooses k distinct vertices at distances of even lengths from it. Note that the symmetric difference paths are vertex disjoint, and thus for every unmatched vertex u we have found k distinct matched vertices in M_A that are at least as heavy as u. ∎

THEOREM 2.1. *Let $G = (V, E, \phi)$ be a graph and $\phi : V \mapsto R_{\geq 0}$ a weight function. Then Algorithm 1 computes a $\frac{k}{k+1}$-approximate matching for the MVM problem on G.*

Proof. Let M_A, M_{opt} and U be all as defined in the Lemma, and consider paths in the symmetric difference between M_A and M_{opt}. Each vertex $u \in U$ is an endpoint of an alternating path in the symmetric difference, since no edge from M_A is incident on it. Thus

$$\begin{aligned}\phi(M_{opt}) &= \phi(M_A) + \phi(M_{opt} \setminus M_A) - \phi(M_A \setminus M_{opt}) \\ &\leq \phi(M_A) + \phi(M_{opt} \setminus M_A).\end{aligned}$$

By Lemma 2.1 we have $\phi(M_{opt} \setminus M_A) \leq \frac{1}{k}\phi(M_A)$. Hence

$$\begin{aligned}&\phi(M_{opt}) - \phi(M_A) \leq \frac{1}{k}\phi(M_A) \\ \Rightarrow &\phi(M_A) \geq \frac{k}{k+1}\phi(M_{opt}).\end{aligned}$$

∎

THEOREM 2.2. *The time complexity of the $(\frac{k}{k+1})$-Iter approximation algorithms is $O(\Delta^k m)$, where Δ is the maximum degree of a vertex.*

Proof. The $(\frac{k}{k+1})$-cardinality matching can be found in $O(mk)$ time using k rounds of the Micali and Vazirani algorithm [9].

In each iteration of the **for** loop in the $(\frac{k}{k+1})$-Iter algorithm we choose an unmatched vertex u and examine all neighbors of u. Denote the adjacency set of u by $N(u)$. In the worst case, we search an alternating tree of height $2k$. We have k even levels in the alternating tree. At each even level i the algorithm examines all neighbors of each vertex v except the vertex matched to it, and at an odd level the search continues with the matched edge. We renumber the even levels from 0 to $k-1$. Since the degree of each vertex v is upper bounded by Δ, we have at most $N(u)(\Delta - 1)^i$ vertices that are searched at an even level i. So for a vertex u the algorithm examines at most a number of vertices equal to

$$\begin{aligned}&d(u) + \sum_{i=1}^{k-1} d(u)(\Delta - 1)^i \\ &= d(u)(\Delta - 1)^k/((\Delta - 1) - 1) \\ &= O(d(u)(\Delta^{k-1})).\end{aligned}$$

Since a vertex can be unmatched at most Δ times in total $O(d(u)\Delta^k)$ vertices are searched. Summing over all vertices we have $\sum_{u \in V} O(d(u)\Delta^k) = O(\Delta^k m)$. ∎

Thus the time complexity of the 1/2-approximation algorithm is $O(\Delta m)$; that of the 2/3-approximation algorithm is $O(\Delta^2 m)$.

3 Parallel Algorithms for MVM in non-Bipartite Graphs

Algorithm 2 Parallel 2/3-Init-Iter approximation algorithm.

```
1:  $M \leftarrow$ PAR-2/3-APPROXCARD($G = (V, E, \phi)$);
2:  do
3:      done = true;
4:      for all unmatched $u \in V$ do in parallel
5:          Find an aug. path $P$ from $u$ to $v$ s.t. $|P| \leq 3$;
6:          if $P$ is found and $u < v$ then
7:              if LOCK($P$,$u$) = true then
8:                  $M \leftarrow M \oplus P$; done = false;
9:                  release all locks;
10:             else
11:                 continue;
12:             end if
13:         else
14:             Find a highest gain increasing path $P'$ from $u$ s.t. $|P'| \leq 4$;
15:             if $P'$ is found then
16:                 if LOCK($P'$,$u$) = true then
17:                     $M \leftarrow M \oplus P'$; done =false;
18:                     release all locks;
19:                 else
20:                     continue;
21:                 end if
22:             end if
23:         end if
24:     end for
25: while done = false
```

Now we turn to the parallelization of the 2/3-approximation Iterative Algorithm, and it is described in Algorithm 2. While there are unmatched vertices, the algorithm searches for augmenting paths (of length at most three) or increasing paths (of length at most four). Once a thread finds one such path, it locks vertices on the path such that no other thread should augment or update the matching since these paths can overlap. If a thread cannot acquire all locks needed, then it releases all locks and proceeds to search from other unmatched vertices. There is an implicit synchronization barrier across all threads at the end of each iteration of the **while** loop.

Note that for the 1/2-approximation algorithm, the same method of parallelization may be employed by restricting the length of an augmenting path to one, and the length of an increasing path to two.

Now we discuss how the test and set locks are employed in the parallel algorithm. The lock is free if its value is zero, and not free otherwise. If a thread reads a value of zero for a lock, then it has atomic access to the lock variable and can set it to a nonzero value. If a thread reads a nonzero value for a lock, then it is unavailable. We allow an augmenting path joining the vertices u and v to augment the matching only if $u < v$, and in this way we prevent two threads from attempting to acquire locks and augmenting the same path from opposite directions.

For a single matched edge (u, v) on an augmenting path, we lock its lower-numbered endpoint; for two matched edges (u, v) and (x, y) on an increasing path, we need to lock the lower-numbered endpoint of both edges, but with the lowest numbered endpoint locked first. Hence the lock for first_min, the minimum among all four vertices is acquired first, and then the lock for second_min, the lower numbered endpoint of the other matched edge, is acquired.

The locking procedure is described in detail in Algorithm 3. As an example, when a thread finds an augmenting path of length one, $\{u, q\}$, then it tests $lock(u)$. If $lock(u) \neq 0$ then the algorithm continues to the next unmatched vertex. If it equals 0, it sets lock(u) with 1, and then it tests the status of the lock on q. If a thread finds a $lock(u)$ value to be nonzero, then it abandons the attempt to lock the remaining vertices on the augmenting or increasing path, releases any locks that it has acquired on the path, and processes the next unmatched vertex. If $lock(q) = 0$ then it sets $lock(q)$ to 1. After acquiring lock q, the thread checks if the augmenting path has not been changed by another thread; if it has changed then the thread releases all acquired locks and continues to the next unmatched vertex. After augmenting the matching using the path, the thread then releases locks on u and q. Similar processes are described in Algorithm 3 for augmenting paths of length three, and increasing paths of length two and four.

In Algorithm 2, we have to consider the possibility that in an iteration of the **for** loop, none of the threads is able to augment or update the matching because they are unable to acquire the locks. This can happen in the case of a cyclic wait, where each thread is unable to acquire all the locks it needs because other threads have acquired some of the locks, and there is a cyclic dependence on a subset of threads. We illustrate this in Fig. 1 for a set of increasing paths of length four that overlap with each other and induce a cycle in the graph. A thread T_i processing the unmatched vertex u_i needs to lock u_i and endpoints of two consecutive matched edges (we consider the increasing paths in the clockwise direction). Thus T_1 needs to lock u_1, and the lower numbered endpoints of the edges (v_1, v_2) and (v_3, v_4); and so on, with the last thread T_k needing to lock u_k, and the lower endpoints of (v_{2k-1}, v_{2k}) and

Algorithm 3 Locking procedure.

```
procedure LOCK($P$,$u$)
    if lock($u$)=0 then
        lock($u$)=1;
        if $P$ is an augmenting path then
            if lock($q$)=0 then                ▷ $q > u$ is other unmatched end point
                lock($q$)=1;
            else
                release any locks;
                return false;
            end if
        end if
        for each matched edge $e = (v_i, v_j)$ on $P$ do
            min_$v$_in_$e$ = min($v_i, v_j$);
        end for
        for each min_$v$_in_$e$ in increasing order do
            if lock(min_$v$_in_$e$)=0 then
                lock(min_$v$_in_$e$)=1;
            else
                release any locks;
                return false;
            end if
        end for
        if $P$ has not changed then
            return true;
        else
            release any locks;
            return false;
        end if
    else
        return false;
    end if
end procedure
```

(v_1, v_2). If each thread T_i succeeds in acquiring only one lock (say v_{2i-1}), we could have deadlock depending on what the threads do when they fail to acquire the second lock, and none of the threads might be able to update the matching. (The reader could verify that a similar situation can arise with a set of augmenting paths of length 3 that induce a cycle as well.)

We now consider the situation with deadlock in more detail. In the context of Algorithm 2, there are three potentially bad things that could happen. First is deadlock, when some thread cannot acquire the locks it needs and cannot execute another instruction. This does not happen here by design since when a thread fails to acquire a lock it proceeds to the next unmatched vertex or to the next iteration. Second is starvation, when a thread is not able to acquire locks and match vertices throughout the algorithm, although it has a

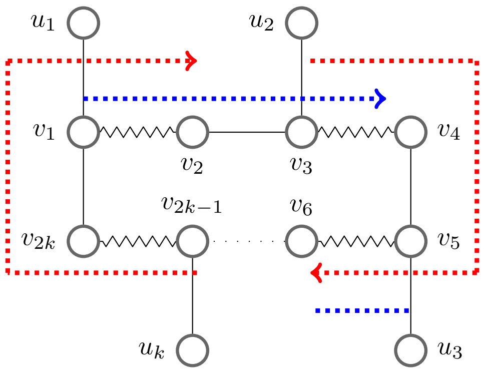

Figure 1: A set of increasing paths of length four that could induce a cyclic wait among threads. Edges drawn in wavy lines are matching edges and the edges drawn with straight lines are non-matching edges.

valid 2-augmentation, because other threads have higher priority. This also cannot happen here because we show that at least one thread responsible for augmenting or updating the matching for overlapping paths succeeds in each iteration, and thus in at most n iterations, all vertices will be processed. Third is livelock, when a cyclic wait renders every thread unable to acquire the locks it needs, and we prove now that it cannot happen in this parallel algorithm. Campbell [2] discusses these issues and provides several references.

THEOREM 3.1. *In each iteration of the* **for** *loop in Algorithm 2, at least one thread among a set of threads competing for locks for overlapping augmenting or increasing paths will be able to acquire the locks it needs and update the matching.*

Proof. We distinguish between the locks for unmatched vertices and matched vertices, and say they are locks of different types. In any iteration of the **for** loop, there will be no dependence between locks of two distinct types. Cyclic dependencies among threads $\{T_1, ...T_k\}$ occur when these threads need to acquire two locks of the same type, and one lock is acquired by a thread T_i and the other by another thread T_j, and so on, and together these create a cyclic wait. In this case these threads fail to acquire the locks they need and release them, and no thread can update the matching.

We consider cases where such dependencies may occur, and hence we do not need to consider several cases:

1. A set of increasing paths of length two, since each thread requires a lock of a distinct unmatched vertex and a lock of a distinct matched vertex, and these locks are disjoint.

2. A set of increasing paths consisting of both lengths two and four for the same reason as above.

3. A set including increasing paths of length four and augmenting paths of length three, since a thread that locks an augmenting path needs two locks of unmatched vertices and one lock for a matched vertex, whereas the thread locking an increasing path needs one lock of an unmatched vertex and two locks of matched vertices.

We consider the following four cases.

Case 1: An overlapping set of augmenting paths of length three that induce a (longer) path. There is no cyclic dependency here, but we consider the case to identify a thread that could augment the matching. Let the k augmenting paths be listed as $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$, for $1 \leq i \leq k$. Let thread T_i be assigned to augment $\{u_i, v_{2i-1}, v_{2i}, u_{i+1}\}$. Then since there is no contention for the vertices u_1 and u_{k+1}, the thread T_1 can lock the former and T_k can lock the latter. If all threads lock their first unmatched vertices, then thread T_k acquires both its locks and can augment. If not, some thread T_j for $j \geq 2$ cannot lock its first unmatched vertex since thread T_{j-1} has acquired it. Choose T_j to be the lowest numbered such thread. Then by choice of j, T_{j-1} has locked both its unmatched vertices, and hence can augment.

Case 2: An overlapping set of augmenting paths of length three that induce a cycle. Let the k augmenting paths be $\{u_i,\ v_{2i-1},\ v_{2i}, u_{i+1}\}$ for $1 \leq i < k$ and $\{u_i, v_{2i-1}, v_{2i}, u_1\}$ for $i = k$. Let thread T_i be assigned to augment $\{u_i, v_{2i-1}\ ,v_{2i}, u_{i+1}\}$ and T_k be assigned to augment $\{u_k, v_{2k-1},\ v_{2k}, u_1\}$. Consider the lowest numbered unmatched vertex u_i in this cycle. Then the previous and the next unmatched vertices in the cycle, u_{i-1} and u_{+1}, are numbered higher than u_i. Since in Algorithm 2 we augment only from a lower-numbered unmatched vertex to a higher-numbered unmatched vertex, such a cyclic set of dependencies among augmenting paths requiring locks cannot exist. Thus this case reduces to a non-cyclic set of augmenting paths, and from Case 1, one thread must succeed.

Case 3: An overlapping set of increasing paths of length four that induce a path. Again, there is no cyclic wait here, but we go through the exercise to identify a thread that can update the matching. Let the k increasing paths be $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$, for $1 \leq i \leq k$. Let thread T_i be assigned to update $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$. We denote (v_{2i-1}, v_{2i}) as the first matched edge of T_i and (v_{2i+1}, v_{2i+2}) as the second matched edges of T_i. If all threads lock a vertex in the first matched edge first then T_k (the last thread) will lock the vertex in the second matched edge since it is not shared. If not, there is some thread T_i such that its neighbor thread T_{i+1} locks a vertex in its second matched edge first. Choose T_i to be the lowest numbered such thread. If T_{i+1} succeeds in locking a vertex in its first matched edge also, then it can augment the matching. If it fails, then by choice of i thread T_i has acquired its second matched edge, and can augment.

Case 4: An overlapping set of increasing paths of length four that induce a cycle in the graph (see Fig. 1). Let the k increasing paths be denoted by $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$ for $1 \leq i < k$ and $\{u_i, v_{2i-1}, v_{2i}, v_1, v_2\}$ for $i = k$. Let thread T_i be assigned to the path $\{u_i, v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}\}$ and T_k be assigned to the path $\{u_k, v_{2k-1}, v_{2k}, v_1, v_2\}$. Consider the lowest numbered matched vertex v_m in the cycle, and denote the two threads competing for it by T_i and T_{i+1}. The one which fails to lock v_m will not seek to lock any other vertex, and thus the cyclic dependence is now broken. Hence again, we have reduced this case to Case 3, and thus one thread must succeed in acquiring locks and updating the matching.

This completes the proof. □

4 Algorithms for Comparison

We compare the new 2/3-Iter, 2/3-Init-Iter, 1/2-Iter and 1/2-Init-Iter approximation algorithms with an exact MVM algorithm and four other approximation algorithms. The Exact MVM algorithm we used is the variant that is fastest among the Direct and Iterative algorithms. This variant is initialized with a maximum cardinality matching, and then the algorithm transforms the matching to an MVM using increasing paths from unmatched vertices. We call it the Direct-Increasing Exact algorithm; the exact algorithm that sorts vertices and matches them in order by looking for augmenting paths only (see the Introduction) is the Direct-Augmenting algorithm. The exact algorithm that searches for k-augmentations without restricting the value of k (simiilar to the $k/(k+1)$-approximation Iterative algorithm) is the Exact Iterative algorithm. We proceed to list the other approximation algorithms now, and due to space considerations, refer the reader to earlier papers [1, 11] for more details.

The 2/3-Direct approximation algorithm for MVM is based on sorting vertices in non-increasing order of weights and finding augmenting paths of length at most three that reach a heaviest vertex [1]. The time complexity of the 2/3-Direct algorithm is $O(m \log \Delta + n \log n)$. The Suitor algorithm employs a proposal-based approach, and is a 1/2-approximation algorithm for maximum edge weighted matching [5].

We compare our algorithms against a $(2/3 - \epsilon)$ approximation algorithm (with $\epsilon = 0.01$), the Random

Order Augmentation Matching Algorithm (ROMA). The ROMA algorithm is a variant of an algorithm due to Pettie and Sanders [10] and was implemented by Maue and Sanders [6]. The algorithm randomly orders the unmatched vertices, and then performs a 2-augmentation of highest gain from each in a round-robin fashion. The ROMA algorithm can be initialized with the 1/2-approximation algorithm called the Global Paths algorithm (GPA) [6], which constructs sets of paths, and cycles of even length by considering the edges in non-increasing order of their weights. Then it computes a maximum weight matching for each path and cycle by dynamic programming. The time complexity of the GPA algorithm is $O(m \log n)$, and that of the ROMA algorithm is $O(m \log \epsilon^{-1})$.

The final algorithm we implemented is a $(1 - \epsilon)$-approximate Scaling algorithm due to Duan and Pettie [4] (with the choice of $\epsilon = 1/3$). This algorithm detects and processes blossoms, and performs updates of dual variables during the course of the augmentations. Its time complexity is $O(m\epsilon^{-1} \log \epsilon^{-1})$.

5 Experimental Results

5.1 Computational Platform and Test Set For the experiments we used Intel Xeon E5-2660 processor based systems (part of the Purdue University Community Cluster), called *Rice* and *Snyder*[1]. The machines consist of two processors, each with ten cores running at 2.6 GHz (20 cores in total), with 25 MB unified L3 cache, and 64 GB of memory for Rice and 256 GB for Snyder. The operating system is Red Hat Enterprise Linux release 6.7. All code was developed using C++ and compiled using the g++ compiler (version: 6.3.0) using the -O3 flag.

Our test set consists of nine real-world graphs taken from the well-known SuiteSparse collection covering several application areas; and synthetic datasets generated by the RMAT generator. We generated three different synthetic datasets varying the RMAT parameters. These are (i) rmatG500 representing graphs with skewed degree distribution from the Graph 500 benchmarkwith parameter set (0.57, 0.19, 0.19, 0.05), (ii) rmatSSCA from HPCS Scalable Synthetic Compact Applications graph analysis benchmark with parameter set (0.6, 0.133, 0.133, 0.133), and (iii) rmatER Erdos-Renyi random graphs with uniform degree distributions and parameter set (0.25, 0.25, 0.25, 0.25). Table 1 gives some statistics on our test set. The graphs are listed in increasing order of the total number of vertices. Integer weights of vertices were generated uniformly at random in the range [1 1000]. The reported results are averages of ten trials of randomly generated weights.

5.2 Results on Serial Algorithms Since the initialized versions of the 1/2- and 2/3-Iter algorithms are slightly faster than their uninitialized versions, and the weights of the matchings are close, we report results only from 1/2- and 2/3-Init-Iter algorithms. The time taken by the exact algorithm and relative performance of the approximation algorithms with respect to the exact algorithm are reported in Table 2. (Bold fonts indicate the algorithm with the fastest time or the highest weight for each problem.) The 1/2-Init-Iter is 110 times faster than the exact algorithm in geometric mean, and it is the fastest of the approximation algorithms for ten of the twelve graphs; for the remaining two graphs, the Suitor algorithm is the winner. The 2/3-Init-Iter algorithm is about 39 times faster, while the 2/3-Dir and the Suitor are 23 and 43 times faster, respectively, than the exact algorithm, all in geometric mean. The GPA-ROMA approximation algorithm is 1.3 times faster, and the scaling approximation algorithm is slower than the exact algorithm by a factor of 1.7.

On the nlpkkt200 problem, the only Exact algorithm that terminated (in 161 hours) is the Direct-Increasing MVM algorithm. None of other Direct algorithms, or the Iterative Exact MVM algorithm, or an exact MEM algorithm in LEDA [7], terminated in 200 hours. The fastest approximation algorithm on this problem was the 1/2-approximate Init-Iter algorithm which ran in under two seconds; the Suitor algorithm took 12 seconds, and the 2/3-Init-Iter algorithm needed 23 seconds. The rmat-ER and rmat-SSCA are the problems on which approximation algorithms needed the most time because these graphs have the largest numbers of edges. The greatest time taken by the 1/2-Init-Iter algorithm on any graph is 72 seconds on rmat-ER, while for the 2/3-Init-Iter algorithm the greatest time taken is 405 seconds on rmat-G500. The Suitor algorithm spent the longest time on rmat-ER which finished in 1004 seconds, while the 2/3-Dir algorithm spent much longer time on rmat-ER which took 1960 seconds.

The 1/2-Init-Iter approximation algorithm is faster on average because initially it finds a maximal cardinality matching without considering the weights, and this step can be performed fast. The algorithm then tries to increase the weight using the remaining unmatched vertices. The Suitor algorithm is slightly faster than the 1/2-Init-Iter on two graphs where the average degree is low and the number of vertices is large, since on these two graphs the Suitor has a low number of annulments due to their small average degrees. All higher approximation ratio algorithms run slower than these two

[1] https://www.rcac.purdue.edu/compute/rice/ , https://www.rcac.purdue.edu/compute/snyder/

Graph	$\lvert V\rvert$	Degree			$\lvert E\rvert$
		Max.	**Mean**	**SD/Mean**	
kron_g500-logn21	2 097 152	213 904	117.92	7.47	91 040 932
M6	3 501 776	10	5.99	0.14	10 501 936
hugetric-00010	6 592 765	3	2.99	0.01	9 885 854
rgg_n_2_23_s0	8 388 608	40	15.14	0.26	63 501 393
hugetrace-00010	12 057 441	3	2.99	0.01	18 082 179
nlpkkt200	16 240 000	27	26.60	0.09	215 992 816
hugebubbles-00010	19 458 087	3	2.99	0.01	29 179 764
road_usa	23 947 347	9	2.41	0.39	28 854 312
europe_osm	50 912 018	13	2.12	0.23	54 054 660
rmat-G500	48 877 747	2 407 313	85 28	15.48	2 084 251 521
rmat-SSCA	93 488 461	641 453	45.29	9.96	2 117 212 258
rmat-ER	134 217 728	241	32.00	0.29	2 147 483 625

Table 1: The set of test problems and various associated measures.

1/2-approximation algorithms, but the 2/3-Init-Iter algorithm is quite close to Suitor in geometric mean. The GPA-ROMA algorithm spends considerable time to find highest gain 2-augmentations among all possible paths and cycles. The scaling algorithm is the slowest in all instances which is due to high number of iterations and the cost of processing blossoms and dual variables at each scale.

We have plotted the serial run times of the 2/3-Direct and GPA-ROMA algorithms against the number of edges scanned by the algorithm in a log-log plot in our earlier paper [1], and have seen a linear relationship for these algorithms. Here we have observed the same for the 2/3-Init-Iter, Suitor and the GPA-ROMA algorithms, showing that the run times are primarily determined by the number of edges scanned by the algorithms. We do not show the plot due to space limitations.

Now we turn to the weights and cardinalities of the matchings. In general the approximation algorithms obtain much greater weights than the guaranteed lower bounds. In order to distinguish the performance of the algorithms, we measure the gap to optimality as a percentage given by

$$100 * \left(1 - \frac{\text{weight obtained by approx. algorithm}}{\text{optimum weight}}\right).$$

We take the same approach for the cardinality (a maximum vertex weighted matching is also a maximum cardinality matching when the vertex weights are positive).

As shown in Table 3 the weights obtained by the 2/3-Init-Iter approximation algorithm are better than the values obtained by the other approximation algorithms, and the Suitor algorithm is the worst. In geometric mean the gap to optimality of the 2/3-Init-Iter algorithm is 0.08% for weight (and 0.99% for cardinality). The weights and cardinalities obtained by the GPA-ROMA algorithm are worse those obtained by 2/3-Iter and 2/3-Init-Iter algorithms. The 2/3-Dir algorithm has a gap to optimal weight of 0.3% (and gap to optimal cardinality of 2.9%.) Surprisingly the quality of 1/2-Init-Iter matching is better than the matching obtained by the 2/3-approximate scaling algorithm on average.

5.3 Results from Parallel Algorithms We implemented the parallel algorithm using C++ and OpenMP 3.1, using the g++ compiler functions __sync_lock_test_and_set and __sync_lock_release for locking. We pinned threads to cores to reduce overhead of thread migration between cores by setting the environment variable GOMP_CPU_AFFINITY="0-$(t-1)$" where t is the number of threads. Using 20 threads with 20 cores, thread i is pinned to core i. We used static scheduling, and experimented with chunk sizes of 256 and the default value, and we report the faster running time from these two options.

We compare the 2/3-Iter, 2/3-Init-Iter, 1/2-Iter and 1/2-Init-Iter approximation algorithms and the Suitor algorithm. Suitor is known to be the most concurrent approximation algorithm for edge-weighted matching since it processes vertices in arbitrary order, and vertices are free to make proposals to their highest weight available neighbor. We report running times (in seconds) and speedups in Table 5.3. The speedup for an α-approximation algorithm is computed as the ratio of the time of the fastest serial α-approximation algorithm and the time needed by the parallel α-approximation algorithm on twenty threads, for $\alpha = 1/2, 2/3$. Thus the baseline serial algorithm is different for algorithms with different approximation ratios.

Clearly the 1/2-Iter and 1/2-Init-Iter algorithms are the fastest parallel algorithms on average, and both are faster than the Suitor algorithm on all but two problems. The uninitialized variant of the Iterative

Table 2: Running time (seconds) of the exact algorithm and relative performance of six approximation algorithms for the MEM and MVM problems.

Graph	Time Exact MVM (s)	Relative performance $1-\epsilon$-Scal. $\epsilon = 1/3$	$2/3-\epsilon$ GPA-ROMA $\epsilon = 0.01$	2/3-Dir	2/3-Init-Iter	1/2-Init-Iter	Suitor
kron_g500-logn21	683.9	3.869	7.293	403.2	159.2	**867.4**	266.7
M6	9.967	0.107	0.329	5.876	9.458	**17.16**	7.336
hugetric-00010	21.25	0.200	0.550	11.44	20.01	**32.37**	22.69
rgg_n_2_23_s0	20.96	0.116	0.164	1.424	33.38	**46.14**	5.253
hugetrace-00010	32.09	0.173	0.453	9.242	20.23	**31.43**	21.59
nlpkkt200	5.8E5	1.8E3	1.7E3	1.1E4	2.5E4	**4.1E5**	3.2E4
hugebubbles-00010	66.06	0.198	0.541	10.79	20.84	**33.77**	20.51
road_usa	14.81	0.041	0.110	2.173	2.027	4.675	**8.505**
europe_osm	31.71	0.052	0.115	2.222	3.024	5.841	**10.24**
rmat-G500	6.0E4	8.268	15.10	556.7	146.9	**954.9**	497.0
rmat-SSCA	3.3E4	3.767	6.151	216.6	89.08	**515.1**	164.6
rmat-ER	1.7E3	0.103	0.215	0.849	16.73	**23.22**	1.657
Geom. Mean	1.000	0.632	1.295	22.85	39.45	**109.5**	42.90

Table 3: Gap to optimal weight as a percentage.

Graph	$1-\epsilon$-Scal. $\epsilon = 1/3$	$2/3-\epsilon$ GPA-ROMA $\epsilon = 0.01$	2/3-Dir	2/3-Init-Iter	1/2-Init-Iter	Suitor
kron_g500-logn21	5.049	1.522	1.793	**0.721**	5.438	14.09
M6	0.701	0.156	0.208	**0.082**	0.913	2.388
hugetric-00010	1.572	0.500	0.811	**0.171**	0.863	4.433
rgg_n_2_23_s0	0.204	0.026	0.035	**0.000**	0.005	0.558
hugetrace-00010	1.556	0.488	0.793	**0.134**	0.650	4.391
nlpkkt200	0.305	0.143	**0.074**	0.186	0.494	0.534
hugebubbles-00010	1.569	0.497	0.806	**0.150**	0.750	4.419
road_usa	3.354	1.156	1.740	**0.923**	4.172	7.810
europe_osm	3.083	**0.527**	2.001	0.773	1.834	6.766
rmat-G500	4.691	1.025	1.349	**0.496**	4.824	13.05
rmat-SSCA	5.157	1.503	1.845	**0.903**	5.958	13.29
rmat-ER	0.068	0.000	0.000	**0.000**	0.032	0.186
Geom. Mean	1.249	0.229	0.321	**0.084**	0.755	3.266

algorithm is faster than the initialized variant for the 1/2-approximation algorithm, while the ordering is reversed for the 2/3-approximation Iterative algorithms. The 1/2-Iter and 1/2-Init-Iter algorithms also achieved higher speedups, 8.9 and 7.4, respectively, in geometric mean on these problems, while the speedup of the Suitor algorithm was 3.5.

The fastest 2/3-approximation algorithm is slower than the fastest 1/2-approximation algorithm in parallel as well, by a factor of 2.50 in geometric mean. The speedup of 2/3-Iter and 2/3-Init-Iter are 9.7 and 10.7 respectively. There are four problems on which the fastest 2/3-approximation algorithm is faster than the 1/2-approximate Suitor algorithm, which is surprising.

We define scalability to be the ratio of the time of a serial approximation algorithm to the time taken by that algorithm in parallel on twenty threads. The 2/3-Init-Iter algorithm scales well on twenty threads, and is slightly better than the Suitor algorithm in geometric mean (11.13 for the former and 10.18 for the latter), while the 1/2-Init-Iter algorithm scales slightly worse than Suitor, again on average. The parallel Suitor obtains the same weight as the single thread implementation, while the weights differ slightly in arithmetic mean for the Iterative algorithms (by 0.02% for the parallel 2/3-Init-Iter and 0.2% for the parallel 1/2-Init-Iter). However, the weight and cardinality of matchings from the parallel Iterative approximation algorithms are always better than that of the parallel Suitor algorithm. The invariance of the matching obtained in serial and parallel is an advantage of the Suitor algorithm.

Table 4: Run time (seconds) and speedup obtained with twenty threads of an Intel Xeon processor.

	2/3-Iter		2/3-Init-Iter		1/2-Iter		1/2-Init-Iter		Suitor	
Graph	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
kron_g500-logn21	0.271	14.14	0.286	13.39	0.065	11.33	0.096	7.646	0.175	4.175
M6	0.135	7.805	0.116	9.102	0.068	8.574	0.079	7.389	0.128	4.542
hugetric-00010	0.116	9.139	0.106	10.05	0.062	10.31	0.080	8.038	0.139	4.626
rgg_n_2_23_s0	0.069	9.164	0.072	8.704	0.073	6.189	0.064	7.116	0.298	1.525
hugetrace-00010	0.180	8.790	0.171	9.255	0.109	8.534	0.139	6.721	0.238	3.911
nlpkkt200	3.224	7.240	1.808	12.91	0.188	7.654	0.239	6.008	1.504	0.955
hugebubbles-00010	0.405	7.826	0.364	8.696	0.186	10.52	0.240	8.148	0.412	4.748
road_usa	0.656	10.47	0.621	11.06	0.308	5.659	0.447	3.898	0.221	7.876
europe_osm	0.842	11.54	0.740	13.13	0.448	6.913	0.646	4.797	0.466	6.650
rmat-G500	22.63	16.75	23.79	15.94	3.604	16.52	4.216	14.12	8.389	7.097
rmat-SSCA	25.55	13.00	24.52	13.55	5.101	12.29	5.686	11.03	13.98	4.487
rmat-ER	17.03	5.843	14.85	6.699	9.671	7.411	8.287	8.648	72.67	0.986
Geom. Mean		9.716		10.73		8.911		7.396		3.542

6 Conclusion

This is the *first* parallel algorithm for approximating a weighted matching problem with *approximation ratio better than* 1/2 that we know of. For the MEM and the maximum weighted b-matching problems, the Suitor algorithm with approximation ratio 1/2 has been implemented successfully on shared-memory and distributed-memory multiprocessors.

The Iterative 2/3-approximation algorithm designed here has several advantages over the Direct algorithm. First, it can be initialized with a 2/3-approximate maximum cardinality matching. Previously known algorithms for maximum cardinality and maximum edge-weighted matchings employ initialization algorithms to make them fast in practice. The Direct algorithms (both exact, 1/2- and 2/3-approximation) could not be initialized since they need to process unmatched vertices in a specific order. Second, the Iterative 2/3-approximate MVM algorithm is faster than the Direct algorithm and other $(2/3-\epsilon)$-approximation algorithms for MEM on a serial computer; it also computes heavier weights for the matching. Finally, because the Iterative algorithm processes unmatched vertices in any order, we have designed a (shared memory) parallel algorithm for MVM using it. The parallel algorithm obtains good speed-ups on modest numbers of threads.

References

[1] Al-Herz, A., and Pothen, A. A 2/3-approximation algorithm for vertex-weighted matching. *Discrete Applied Mathematics* (2019). Published online Oct. 19, 2019. DOI 10.1016/j.dam.2019.09.013.

[2] Campbell, R. H. Deadlocks. In *Encyclopedia of Parallel Computing, Volume 1*, D. Padua, Ed. Springer Verlag, 2011, pp. 524–527.

[3] Dobrian, F., Halappanavar, M., Pothen, A., and Al-Herz, A. A 2/3-approximation algorithm for vertex-weighted matching in bipartite graphs. *SIAM J. Sci. Comput. 41*, 1 (2019), A566–A591.

[4] Duan, R., and Pettie, S. Linear time approximation for maximum weight matching. *J. ACM 61*, 1 (2014).

[5] Manne, F., and Halappanavar, M. New effective multithreaded matching algorithms. In *28th International Parallel and Distributed Processing Symposium* (2014), IEEE, pp. 519–528.

[6] Maue, J., and Sanders, P. Engineering algorithms for approximate weighted matching. In *Lecture Notes in Computer Science, Vol. 4525.* Springer Verlag, 2007, pp. 242–255.

[7] Mehlhorn, K., and Näher, S. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[8] Mehta, A. Online matching and ad allocation. *Foundations and Trends in Theoretical Computer Science 8*, 4 (2012), 265–368.

[9] Micali, S., and Vazirani, V. V. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Found. Comp. Sci.* (1980), IEEE, pp. 17–27.

[10] Pettie, S., and Sanders, P. A simpler linear time 2/3- ε approximation for maximum weight matching. *Information Processing Letters 91*, 6 (2004), 271–276.

[11] Pothen, A., Ferdous, S., and Manne, F. Approximation algorithms in combinatorial scientific computing. *Acta Numerica 28* (2019), 541–633.

[12] Schrijver, A. *Combinatorial Optimization: Polyhedra and Efficiency. Volume A: Paths, Flows and Matchings.* Springer, 2003.