A Simple Parallel Approximation Algorithm for the Weighted Matching Problem

Alicia Thorsen, Phillip Merkey, Fredrik Manne (University of Bergen)

Michigan Technological University
Computer Science Technical Report
CS-TR-07-01
September, 2007

Contents

List of Algorithms

1 Introduction

1.1 Weighted Matching Problem

A *matching* M in a graph $G = (V, E)$ is a subset of edges such that no two edges in M are incident to the same vertex. If $G = (V, E, w)$ is a weighted graph with edge weights $w : E \rightarrow \mathbb{R}_+$, the *weight of a matching* is defined as $w(M) := \sum_{e \in M} w(e)$. The weighted matching problem is to find a matching with maximum weight. Matchings are used as a part of many important real-world applications including scheduling, network routing and load balancing.

1.2 Background

The first exact polynomial time algorithm for this problem was given by Edmonds in 1965, which runs in $O(n^2 m)$ [8], where n and m are the number of vertices and edges of a graph respectively. Since then, much work has been done to improve the worst case running time of this algorithm. The fastest known result is due to Gabow who reduced it to $O(nm + n^2 \log n)$ [8]. In the area of parallel algorithms, it is still an open problem to find a maximum weight matching using an NC algorithm [4]. NC is the class of problems that are computable in polylogarithmic time with polynomially many processors.

1.3 Approximation Algorithms

Some matching applications require graphs of such large size, that the exact algorithm is too costly. These include the refinement of Finite Element Method nets, the partitioning problem in Very Large Scale Integration (VLSI) Design, and the gossiping problem in telecommunications [8]. These applications need approximation algorithms which are fast and produce near optimal results.

An approximation algorithm is measured by its performance ratio c, which guarantees it will find a weight at least c times the optimal solution. The best known serial approximation algorithm is due to Drake and Hougardy [8], which runs in linear time and has a performance ratio of $\frac{2}{3} - \epsilon$. They also presented a $1 - \epsilon$ approximation algorithm for the theoretical Parallel Random Access Machine (PRAM) model [4], however it requires a large number of processors and is therefore not as useful in practice. Hoepman [3] developed a practical parallel distributed approximation algorithm which runs in linear time however it has a performance ratio of $\frac{1}{2}$.

1.4 Contribution

We present a parallel approximation algorithm for the shared memory model which is based on the best serial $\frac{2}{3} - \epsilon$ algorithm. Drake and Hougardy and Pettie and Sanders both presented linear time algorithms with the same approximation ratio, however the Pettie and Sanders version is simpler and converges to $\frac{2}{3} - \epsilon$ faster [5].

One advantage of using the shared memory model is the ability to design parallel algorithms which are very similar to their serial counterparts. Even though this problem has an irregular access pattern, our resulting algorithm is simple to understand and easy to implement.

2 Algorithm

Our parallel approximation algorithm is composed of two phases. In the first phase we quickly create an initial matching, and in the second phase we perform a series of short augmentations to iteratively increase the weight of the current matching.

Algorithm 1 Parallel Matching Algorithm

```
Require: $G = (V, E), w : E \to \mathbb{R}_+$
 1: create a shared initial matching $M$ using the Parallel Path Growing Algorithm
 2: synchronize
 3: for $i = 1$ to $k/p$ do
 4:     choose a vertex $x \in V$ randomly
 5:     let $aug(x)$ be the heaviest short augmentation centered at $x$
 6:     if no conflicts exist then
 7:         $M \leftarrow M \oplus aug(x)$
 8:     end if
 9: end for
10: return $M$
```

In this algorithm p represents the number of processors and $k = \frac{1}{3}n \ln \frac{1}{\epsilon}$.

2.1 Initial Matching

To create the initial matching we use a parallel version of the Drake and Hougardy Path Growing Algorithm [1], which creates two matchings in the form of vertex disjoint paths.

2.1.1 Serial Path Growing Algorithm

The algorithm starts by randomly choosing a vertex x, then finds the heaviest edge (x, y) incident to x. This edge is placed in the matching M_1 and all other edges incident to x are removed from the graph. The heaviest edge incident to y is then added to M_2 because it cannot be added to M_1. M_1 and M_2 are valid matchings so they cannot contain adjacent edges. All other edges incident to y are then deleted.

This process continues and a path is eventually created from x by repeatedly choosing the heaviest edge available. Once an edge is chosen from a vertex, all other edges incident to the vertex are removed to ensure the paths are disjoint. If a path can no longer be extended, a new path is started from another randomly chosen vertex. This continues until all vertices of the graph belong to some path, even if the path is of length 0.

The edges on each path are alternately inserted into M_1 and M_2 to ensure they contain valid matchings. The algorithm returns the matching with the heavier weight.

Algorithm 2 Serial Path Growing Algorithm

```
Require: $G = (V, E), w : E \to \mathbb{R}_+$
 1: $M_1 \leftarrow \emptyset, M_2 \leftarrow \emptyset, i \leftarrow 1$
 2: while $E \neq \emptyset$ do
 3:     choose a vertex $x \in V$ randomly
 4:     while $\exists$ a neighbor of $x$ do
 5:         let $\{x, y\}$ be the heaviest edge incident to $x$
 6:         add $\{x, y\}$ to $M_i$
 7:         $i \leftarrow 3 - i$
 8:         remove $x$ from $G$
 9:         $x \leftarrow y$
10:     end while
11: end while
12: return max$(w(M_1), w(M_2))$
```

2.1.2 Parallel Path Growing Algorithm

Since the paths are disjoint, they can be easily created in parallel. In our algorithm, each processor starts with a unique vertex chosen uniformly at random, then grows a set of disjoint paths. This phase of the algorithm is asynchronous since each processor finds its paths independently. The algorithm ends when all vertices have been visited by some processor.

As the paths are created, each processor marks its vertices to prevent other processors from claiming them. We use atomic memory operations to mark vertices to avoid race conditions. We used the *compare-and-swap (CAS)* operation to allow a processor to claim a vertex only if it is listed as available. If the vertex has already been marked by another processor the *CAS* operation fails. A processor must mark both end vertices of an edge to claim that edge.

If a processor is unable to acquire a vertex to start a new path, it randomly restarts on another vertex. If the vertex was needed to extend a path, the processor proceeds to the next heaviest neighboring edge available. If there are no more available neighbors, the processor looks for another vertex to start a new path.

2.1.3 Difficulties in the Parallel Version

The parallel version of the Path Growing Algorithm is different from the serial one in the matching that it finds. If two different processors have marked the end vertices of an edge, then neither of them can add the edge to their paths. If this is a dominating edge, the matching found by the parallel version can be significantly less than the one found by the serial version.

If we allow one processor to claim the edge and move forward with its path, the other processor will be forced to rescind at least one edge. In the worst came the advancing processor may

Algorithm 3 Parallel Path Growing Algorithm

```
Require: $G = (V, E), w : E \to \mathbb{R}_+$
1: $globalM \leftarrow \emptyset$
2: while $\exists$ an available vertex $x \in V$ do
3:     $M_1 \leftarrow \emptyset, M_2 \leftarrow \emptyset, i \leftarrow 1$
4:     choose a vertex $x \in V$ randomly
5:     success $\leftarrow$ mark $x$ atomically with $myProcessorID$
6:     while success $= true$ do
7:         success $\leftarrow false$
8:         while (success = $false$) and ($\exists$ an available neighbor of $x$) do
9:             let $\{x, y\}$ be the heaviest available edge incident to $x$
10:            success $\leftarrow$ mark $y$ atomically with $myProcessorID$
11:        end while
12:        if success $= true$ then
13:            add $\{x, y\}$ to $M_i$
14:            $i \leftarrow 3 - i$
15:            $x \leftarrow y$
16:        end if
17:    end while
18:    $globalM \leftarrow globalM \cup$ max$(w(M_1), w(M_2))$
19: end while
20: return $globalM$
```

travel the same path as the receding processor(s) and eventually traverse the entire graph. In this case the parallel version would have no gain over the serial version.

The second option is to merge the two paths along the edge wanted by both processors. This does not ruin the time complexity however it may not produce the same path as the serial version. If the path is started simultaneously from both ends and joined in the middle, its weight can be less than the path created by starting at one end and progressing to the other end.

Another issue with merging is that it requires two processors to synchronize since both must agree before paths can be merged. In the worst case we can have a queue where each processor is waiting on another processor. The heaviest edge available is always chosen by each processor so it is not possible to have a waiting cycle. This means there will always be at least one processor that is not waiting on another processor so the algorithm will not livelock.

2.2 Short Augmentations

In the second phase of our parallel approximation algorithm, the processors locate short alternating paths, and use them to increase the weight of the matching. The idea of short augmentations was introduced by Drake and Hougardy [8] in their $\frac{2}{3} - \epsilon$ algorithm.

2.2.1 Serial Short Augmentations Algorithm

An *alternating* path/cycle contains edges alternately from M and $E \setminus M$. An augmentation is an alternating path/cycle where the weight of the edges in $E \setminus M$ is strictly greater than the weight of the edges in M. The gain of an augmentation P is $g(P) = w(P \setminus M) - w(P \cap M)$. If a matching M is not maximum, it contains at least one augmentation which can be used to increase the weight of the matching [8]. Given an augmentation P, a matching M can be *augmented* by removing the edges $P \cap M$ and adding the edges $P \setminus M$ to M.

Pettie and Sanders [5] presented a randomized matching algorithm which repeatedly chooses a random vertex x and augments the current matching with the highest-gain augmentation centered at x. To achieve the linear time complexity, we only consider *short augmentations* which contain up to 5 edges. Pettie and Sanders showed that after performing $\frac{1}{3}n \ln \frac{1}{\epsilon}$ short augmentations, the weight of the matching approaches $\frac{2}{3} - \epsilon$.

Algorithm 4 Random Match

```
Require: $G = (V, E), w : E \rightarrow \mathbb{R}_+, k$
1: $M \leftarrow \emptyset$
2: for $i = 1$ to $k$ do
3:     choose a vertex $x \in V$ uniformly at random
4:     let $aug(x)$ be the heaviest short augmentation centered at $x$
5:     $M \leftarrow M \oplus aug(x)$
6: end for
7: return $M$
```

2.2.2 Parallel Short Augmentations Algorithm

The parallel short augmentations algorithm basically divides the k iterations among p processors, and each processor finds their short augmentations independently. Once an augmentation is found a processor tries to claim all the vertices it contains, because only then can it augment the graph. Using atomic memory operations we are able to "lock" a neighborhood of vertices before changing the portion of the matching containing them. This guarantees that simultaneous augmentations will not affect the validity of the matching.

When a processor is claiming its vertices and it discovers one of the vertices it needs has already been marked by another processor, it abandons the short augmentation. It does this by backtracking to those vertices it had already claimed and marks them as available. It continues on to another randomly chosen vertex and starts looking for a short augmentation there.

2.2.3 Difficulties in the Parallel Version

In the parallel version there are many difficulties as compared to the serial algorithm. First of all there is the possibility of wasted iterations if a processor is unable to secure all of its vertices. Furthermore, the iterations are wasted for all processors involved since none of them

Algorithm 5 Parallel Random Match

```
Require: $G = (V, E), w : E \to \mathbb{R}_+, k$
 1: let $M$ be a shared initial matching
 2: for $i = 1$ to $k/p$ do
 3:     choose a vertex $x \in V$ randomly
 4:     let $aug(x)$ be the heaviest short augmentation centered at $x$
 5:     success $\leftarrow true$
 6:     for all vertices $y$ in $aug(x)$ do
 7:         if success = $true$ then
 8:             success $\leftarrow$ mark $y$ atomically with $myProcessorID$
 9:         end if
10:     end for
11:     if success = $true$ then
12:         $M \leftarrow M \oplus aug(x)$
13:     end if
14:     for all vertices $y$ in $aug(x)$ do
15:         if $mark(x)$ = $myProcessorID$ then
16:             mark $x$ as available
17:         end if
18:     end for
19: end for
20: return $M$
```

are allowed to succeed. We have yet to fully determine how these wasted iterations affect the approximation ratio, and if they are redone how it will affect the time complexity.

It is possible to create a priority system where one processor always wins, however it is hard to determine at what point a process is guaranteed none of its claimed vertices can be taken away. It is also possible that a processor sees one of its vertices as unavailable and abandons the augmentation, but the owning processor happens to be in the cleanup stage and was just about to release the vertex.

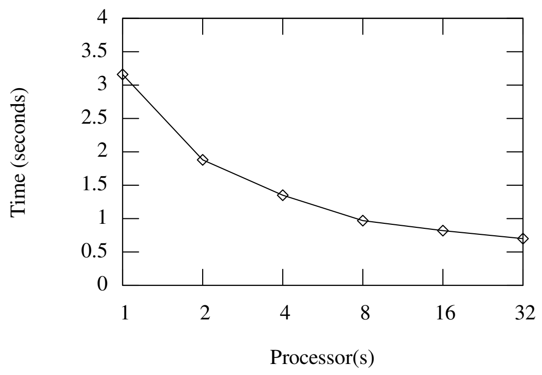
3 Implementation

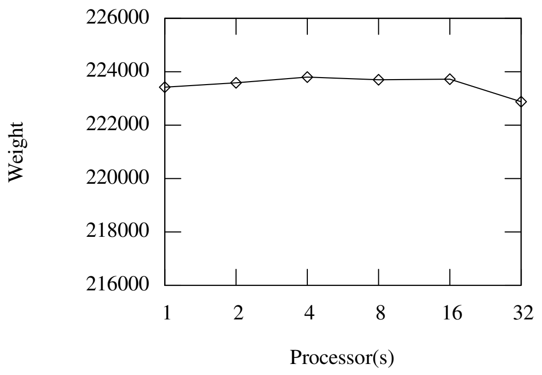
This algorithm was implemented using Unified Parallel C (UPC) on the Army High Performance Computing Research Center Cray X1. UPC is a parallel extension of ANSI C for partitioned global address space programming. It supports the development of parallel applications over many hardware platforms and does not require a shared memory architecture.

This algorithm relies on atomic memory operations (AMOs) to provide data driven synchronization. Currently, UPC does not offer AMOs as a language feature, but the Cray X1 provides a native implementation. The algorithm uses the atomic memory operations compare-and-swap and fetch-and-add.

4 Results

We ran the program on a sparse graph generated from real-world data with 6245 vertices and 42,581 edges. The results we obtained showed that there is some speedup and the weights obtained fluctuate slightly from the serial result.





5 Future Work

It is a non-trivial task to analyze the complexity and approximation ratio for randomized parallel algorithms, however it needs to be done for this algorithm. We need to investigate other options for dealing with the difficulties in the parallel versions of the serial algorithms. For the parallel short augmentations algorithm we need to allow at least one processor to succeed when trying to claim vertices for an augmentation. We also need to determine what should be done when a processor is unable to apply a short augmentation because there was a conflict.

The program needs to be run on a range of graphs to observe how it performs on varying densities. We also need to compare it to other matching algorithms, using both exact and approximation algorithms.

References

[1] D.E. Drake, S. Hougardy, A Simple Approximation Algorithm for the Weighted Matching Problem, *Information Processing Letters* 85 (2003), 211-213.

[2] D.E. Drake, S. Hougardy, Linear Time Local Improvements for Weighted Matchings in Graphs, *Workshop on Efficient Algorithms (WEA)* (2003), 107-119.

[3] Jaap-Henk Hoepman, Simple Distributed Weighted Matchings, eprint cs.DC/0410047 (2004).

[4] S. Hougardy, D. E. Vinkemeier, Approximating Weighted Matchings in Parallel, *Information Processing Letters* 99(3) (2006), 119-123.

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

[6] R. Preis, Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs, *Symposium on Theoretical Aspects of Computer Science (STACS)* (1999) 259-269.

[7] R. Uehara, Z.-Z. Chen, Parallel approximation algorithms for maximum weighted matching in general graphs, *Information Processing Letters* 76:1-2 (2000), 13-7.

[8] D. E. D. Vinkemeier, S. Hougardy, A linear time approximation algorithm for weighted matchings in graphs, *ACM Transactions on Algorithms* 1 (2005).