MAXIMUM CARDINALITY MATCHING FOR BIPARTITE GRAPHS

Thomas Meier, Isabelle Roesch, Conradin Roffler, Samuel Ueltschi

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We implement and optimize two existing parallel algorithms that solve the problem of maximum cardinality matching in bipartite graphs: Parallel Pothen Fan (PPF) and Parallel Tree Grafting (PTG). Our work is focused mainly on the analysis and possible optimizations of the PPF algorithm. We reason about the theoretical performance of PPF using the PRAM model and give a detailed description of the additional optimizations we implemented. We benchmark and test our implementations in a highly multithreaded environment using a Xeon Phi multicore processor. The results confirm that our optimizations for PPF improve the original algorithm.

1. INTRODUCTION

Matching in bipartite graphs has several applications in computer science, for example the marriage problem or computing the block triangular form (BTF) of a sparse matrix [1].

Two recent publications by Azad et al. are concerned with solving the bipartite maximum cardinality matching problem in parallel. The first paper [2] presents a parallel version of the Pothen Fan algorithm (PPF) that achieves good performance on architectures with few fat cores. In the second paper [3] a novel Parallel Tree Grafting (PTG) algorithm is presented that is designed to perform better on architectures with many thin cores. The authors of said paper also claim that PPF is not suited for architectures of this kind.

In this project we recreated some results of the original paper [2] by writing an implementation of PPF that matches the performance stated in the original paper. We give detailed insight into our implementation and explain implementation details and tricks that were omitted by the original authors of the algorithm. Our main contribution is a modified version of PPF that reduces the synchronization overhead of the original (see section 3).

We also implemented the PTG algorithm but were not able to reproduce the results of the original paper. Because we could not invest enough time to properly optimize the algorithm, we are not able to make any statements concerning the performance of this algorithm. Nevertheless we give a brief introduction into PTG in section 3.2.

To verify the claim that PPF does not perform well on many thin cores, we conducted a series of experiments on the Xeon Phi platform [4] (see section 4). Our experiments show that PPF does in fact scale well on this platform by using our improved version of PPF. We also discovered that PPF performs especially well when starting with a bad initial matching (see section 4). By doing so, we achieved superlinear speedup for PPF on Xeon Phi.

2. BACKGROUND: ALGORITHMS FOR MAXIMUM MATCHING IN BIPARTITE GRAPHS

Maximum Cardinality Matching in Bipartite Graphs. Given a bipartite graph $G = (V = X \cup Y, E)$. A matching $M[\cdot] : |V| \to |V| \cup \{v_{null}\}$ is a mapping that assigns each vertex v a unique mate $M[v]$ s.t. $\forall v \in V.\ M[v] \neq v_{null} \implies v = M[M[v]] \wedge \{v, M[v]\} \in E$. Each vertex can have at most one mate. A vertex v is called matched iff $M[v] \neq v_{null}$. An edge is matched if its vertices are matched. We call a matching a maximum cardinality matching if it maximizes the number of edges expressed by M. See [5] for a detailed introduction.

Augmenting Path based Algorithms. The algorithms presented in this report are based on augmenting paths. There are other classes of algorithms for this problem, namely push-relabel [6] and auction based [7]. These however are not the focus of this report.

The best known sequential algorithm for bipartite maximum cardinality matching was discovered by Hopcroft and Karp [8] and runs in $O(|E| \cdot \sqrt{|V|})$. Another algorithm which runs in $O(|E| \cdot |V|)$ was discovered by Pothen and Fan [1]. Although this algorithm asymptotically performs worse than Hopcroft-Karp, in practice it performs better

when parallelized according to Azad et al. [2].

The core idea of both algorithms is to find so called augmenting paths. An augmenting path is a path where the first and last edge are unmatched and where the edges along the path alternate between matched and unmatched. An augmenting path can be inverted by adding every unmatched edge to the matching and removing every matched edge from the matching. This will increase the size of the matching by one. Augmenting path based algorithms try to find and invert such paths. If there are no more augmenting paths in a graph then the matching is maximal.

Initial Matching. Augmenting path based algorithms benefit from starting with an initial matching. A good algorithm for finding a suitable initial matching was discovered by Karp and Sipser [9]. We use both Karp-Sipser and an augmented greedy version based on Karp-Sipser to get our initial matchings.

3. ALGORITHMS AND OPTIMIZATIONS

In this section, we describe the details of our PPF implementation and how we optimized it. In addition to that, we give a brief theoretical analysis of PPF using the PRAM model. Furthermore we also present our Tree Grafting implementation.

3.1. Parallel-Pothen-Fan

Implementation and Optimizations. Azad et al. [2] only give a very abstract description of their PPF implementation with no source code available. Coming up with an efficient PPF implementation proved to be challenging. In this section we describe our PPF implementation with a focus on the techniques used to match the performance of Azad et al.

PPF computes the maximum cardinality matching by finding and inverting disjoint augmenting paths in parallel through multiple depth-first-searches (DFS) that start from all unmatched vertices. A detailed explanation how PPF works can be found in [2]. Our implementation is given in `PPF` (see algorithm 1). Functionally, our algorithm description is equivalent to then one given by [2] but goes into more detail and describes more optimizations than the original.

The main loop of `PPF` iteratively performs multiple DFS in parallel to find augmenting paths. When a path is found, it is inverted to increase the matching by one. The description for this DFS is given in the recursive algorithm `FIND_AND_AUGMENT` (see algorithm 2). Our DFS implementation finds a path and then inverts the matching while returning from the recursion. In practice, we use a stack based implementation of the recursive algorithm to avoid overflowing the stack as suggested by [2]. Note that our algorithm only updates the matching of vertices on the right side of the graph ($M[y], y \in Y$). The left side ($M[x], x \in X$) can be updated later, when no more augmenting paths can be found. This saves unnecessary store instructions if an edge is added and removed from the matching multiple times.

Each thread locks the vertices it visits during the DFS. The original PPF uses a $visited$ array of atomic boolean flags to lock vertices. After each iteration, all flags in the $visited$ array are reset. This requires $O(|Y|)$ store instructions in the sequential phase of the algorithm. We solve this problem by storing the current iteration number in the $visited$ array when locking a vertex instead of just a flag. The iteration number is increasing, therefore we know that a vertex v is unvisited if the value in $visited[v]$ is smaller than the current iteration. By using this technique, we were able to reduce the number of store instructions in the sequential phase of the algorithm from $O(|Y|)$ to $O(1)$.

Because all threads read and write the $visited$ array in parallel, read and store operations must be performed atomically to guarantee correctness. This is a large bottleneck of the algorithm because threads have to wait for the store operations of other threads to complete. In the original algorithm by Azad et al. [2], Test-and-Set is used to set the flags in the $visited$ array. An equivalent implementation of this approach applied to our algorithm is given in $CLAIM_{TAS}$ (see algorithm 3). Using Test-and-Set is problematic because it adds an unnecessary overhead to each failed attempt of locking a vertex when there is no data race. We solve this issue by using the Test-and-Test-and-Set approach to lock vertices (see $CLAIM_{TTAS}$ in algorithm 4). Before using an atomic operation to lock a vertex, we first try a regular memory load. If the vertex is already locked we can omit the atomic operation. This has the benefit of increased caching possibilities and reduced memory traffic. We experimentally verified that PPF performs better with Test-and-Test-and-Set than with Test-and-Set in section 4.

The source code of our PPF implementation is available online[1] for further reference.

PRAM Analysis. The original authors of PPF did not provide a theoretical analysis of the runtime of the algorithm. Doing such an analysis is non-trivial as the runtime of the algorithm mainly depends on the structure of the provided input graph. If a graph has many non overlapping augmenting paths, then PPF performs well as there are no inter-thread conflicts. However if there are many overlaps then the threads will block each other often and the algo-

[1] `https://github.com/suem/dphpc-project/blob/master/src/ppf3.cpp`

Algorithm 1 Parallel Pothen Fan

```
1:  procedure PPF(G = (V = X ∪ Y, E), M)
2:      ▷ Initialization
3:      lookahead[x] ← first neighbor of x                ▷ x ∈ X
4:      iter ← 0
5:      visited[y] ← iter                                 ▷ y ∈ Y
6:      unmatched ← {x ∈ X, x unmatched}
7:      ▷ Discover augmenting paths
8:      repeat
9:          iter ← iter + 1
10:         path_found ← false
11:         for each x ∈ unmatched do parallel
12:             if x matched then
13:                 continue      ▷ skip x if already matched
14:             end if
15:             found ← find_and_augment(x)
16:             if found then
17:                 path_found ← true
18:             end if
19:         end for
20:     until path_found = false
21:     ▷ Complete matching
22:     for each y ∈ Y, y matched do parallel
23:         M[M[y] ← y
24:     end for
25: end procedure
```

Algorithm 2 Find and Augment

```
1:  procedure FIND_AND_AUGMENT(x)
2:      ▷ Lookahead Step
3:      for each y ∈ adj[x], starting at lookahead[x] do
4:          lookahead[x] ← next neighbor of x
5:          if y is unmatched then
6:              if CLAIM_TTAS(y) then
7:                  M[y] ← x          ▷ make x the mate of y
8:                  return true
9:              end if
10:         end if
11:     end for
12:     ▷ Recursive Path Search
13:     for each y ∈ adj[x] do
14:         if CLAIM_TTAS(y) then
15:             success ← find_and_augment(M[y])
16:             if success then
17:                 M[y] ← x          ▷ make x the mate of y
18:                 return true
19:             end if
20:         end if
21:     end for
22: end procedure
```

Algorithm 3 Claim with Test-and-Set

```
1:  procedure CLAIM_TAS(y)
2:      y_iter ← atomic_exchange(visited[y], iter)
3:      return y_iter < iter
4:  end procedure
```

Algorithm 4 Claim with Test-and-Test-and-Set

```
1:  procedure CLAIM_TTAS(y)
2:      if visited[y] < iter then
3:          y_iter ← atomic_exchange(visited[y], iter)
4:          return y_iter < iter
5:      end if
6:      return false
7:  end procedure
```

rithm can only make little progress. In this section we give a brief analysis of PPF using the PRAM model to illustrate the average parallelism of PPF for the best and worst case scenario.

The work PPF has to perform is $O(|V| \cdot |E|)$. Finding an augmenting path requires a DFS of length at most $|E|$. This has to be performed for all $|V|$ vertices. Note that PPF might perform useless work that does not contribute to the solution if it is blocked by another thread and has to abort the DFS. The PTG algorithm described in section 3.2 attempts to mitigate this problem.

Figure 1 illustrates the two program DAGs of PPF for the best case scenario (left) and the worst case scenario (right). In the best case scenario, the input graph only consists of non-overlapping augmenting paths, which can then all be found in parallel without conflicts. This is illustrated by the vertical gray node paths in the DAG. Therefore the maximal depth of the DAG is $|E|$, matching the length of a DFS. Hence PPF has an average parallelism of $O(|V|)$ in the best case. In the worst case scenario, the input graph consists of augmenting paths that always overlap. In this case, only one thread will succeed per iteration and the other threads cannot perform any work. Unsuccessful threads are indicated by the white nodes on the DAG on the right. Therefore the depth of the DAG is the same as the work because there is no parallel work, resulting in an average parallelism of $O(1)$.

The average parallelism of PPF lies between $O(1)$ and $O(|V|)$ depending on the input graph. In section 4 we show that for real-world input data PPF achieves a speedup that is close to the optimal average parallelism.

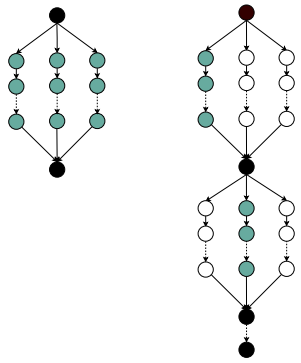


Fig. 1. DAG for PRAM analysis of PPF. Worst case scenario on the right, best case scenario on the left.

3.2. Tree Grafting

The second algorithm we examined that solves the problem of maximum matching on bipartite graphs is the Tree Grafting algorithm, as described in [3]. One of the paper's claims is that Tree Grafting performs better than PPF on architectures with many thin cores such as the Xeon Phi.
We give a short overview of the algorithm we have implemented and on the optimizations we did.

Parallel Tree Grafting. The Parallel Tree Grafting algorithm (PTG) takes a bipartite graph and an initial matching M as input and returns a maximum matching by updating M. PTG is - like PPF - a multi-source searching algorithm. It starts to search for augmenting paths from different unmatched vertices and constructs a forest of alternating trees.
The main difference between PPF is the reuse of already found trees. PPF forgets about the trees it previously found but could not use and has to reconstruct them in every new iteration.
PTG however keeps track of augmenting trees which can still be extended (active trees) and then grafts other trees to the active trees to prolong the augmenting paths contained in them.
For a more detailed explanation of the PTG algorithm including pseudocode, see [3].

Optimizations. Our implementation of the PTG algorithm follows the implementation described in the paper very closely. To build the augmenting paths, we have used the same optimizations as already described in our optimized version of PPF. To store the pointers to the root, leaf and other nodes the PTG algorithm utilizes, we have implemented our own version of a non-blocking queue as a data structure.

	$\|V\|$	$\|E\|$	Density
coPaperDBLP	1'080'872	15'245'732	$5.22\text{e}{-5}$
Wikipedia	7'030'396	45'030'392	$3.64\text{e}{-6}$
Amazon0312	801'454	3'200'440	$1.99\text{e}{-5}$
Gnutella	73'364	176'656	$1.31\text{e}{-4}$

Table 1. Test data used for benchmarks. $|V|$ is the number of vertices, $|E|$ the number of edges. The density describes the sparseness of the graph, where 0.0 represents an empty graph and 1.0 a fully connected bipartite graph.

The source code of our PTG implementation is available online[2] for further reference.

4. EXPERIMENTAL RESULTS

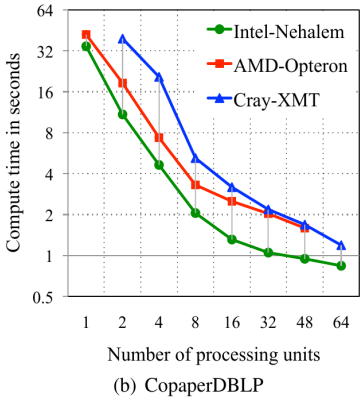
In this section we explain the benchmarking process. We first describe the experimental setup on which we performed our benchmarks on. Then we explain the input data we use. We then proceed to describe the test cases we benchmarked. Finally, we discuss the results from our benchmarks. The raw data of our measurements is available online[3] for further reference.

Experimental Setup. We executed our benchmarks on an Intel Xeon Phi coprocessor (7120 Series) which is based on the Intel MIC architecture. The Xeon Phi has a total of 61 cores which run at a frequency of 1.24 GHz. Each core can run up to four threads, which gives us a total of 244 threads. The Xeon Phi coprocessor has a fully coherent L2 cache with a size of 30.5 MB. To compile our code we use the Intel `icpc` compiler with the following flags: `-openmp -O3 -mmic -lrt`.

Test Data. To test our algorithms, we used several real world graphs from the SuiteSparse Matrix Collection[4]. The graphs and their attributes are listed in table 1. To convert regular graphs to bipartite graphs, we applied the same technique as described in [3]. Let M be the $m \times n$ adjacency matrix of an input graph. The corresponding bipartite graph $G = (V = X \cup Y, E)$ is defined as $X = \{x_1, \ldots, x_m\}$, $Y = \{y_1, \ldots, y_n\}$ and $\{x_i, y_j\} \in E \iff M_{ij} \neq 0$.

Benchmarks. We measured the execution times of the algorithms specified in section 3 on different graphs and with different initial matchings. We compare the execution times of the different algorithms for three reasons: To verify the impact of our optimizations, to reproduce the results

[2] `https://github.com/suem/dphpc-project/blob/master/src/tree_grafting.cpp`
[3] `https://github.com/suem/dphpc-project/tree/master/plot/data`
[4] `http://www.cise.ufl.edu/research/sparse/matrices`



(b) CopaperDBLP

Fig. 2. Runtimes of PPF on coPaperDBLP graph. Plot taken from paper [2].

of [2] and [3] and to observe the impact of different initial matchings on the execution times. As a baseline for the speedup we use the sequential Pothen-Fan algorithm executed on a single core of the Xeon Phi.

Verification. In order to verify the correctness of our implementations, we compare our results to the size of a matching obtained with the Edmonds Maximum Cardinality Matching algorithm [10] from the Boost Graph Library.

Results. First, we compare the PPF implementation from Azad et al. [2] with our optimized implementation. Figure 2 shows the runtime of the original PPF implementation on the coPaperDBLP graph. We ran our implementation on the same graph, once with Test-and-Set and once with Test-and-Test-and-Set (see figure 3). Even though the two experiments were conducted on different architectures, we can nevertheless observe that the achieved runtimes are in the same area. This supports our claim that we were able to match the performance achieved in [2]. We were even able to beat the original in terms of absolute runtime. Furthermore, as figure 3 demonstrates, our proposed improvement to PPF using Test-and-Test-and-Set performs better than Test-and-Set as the number of cores increases. The reason for this is that with a higher number of threads working on the graph, the contention on the $visited$ array is higher and thus $CLAIM_{TAS}$ invalidates a cache-line with each unsuccessful attempt to lock a vertex.

Next, we compare our implementation of PTG against PPF on the coPaperDBLP graph (figure 4). As we can see, even though the PTG algorithm scales up to 240 threads, it is still outperformed by the PPF algorithm. As a consequence of that, we cannot confirm the claim of Azad et al. that the PTG algorithm is better suited for an architecture with many thin cores than the PPF algorithm [3]. A possible explana-

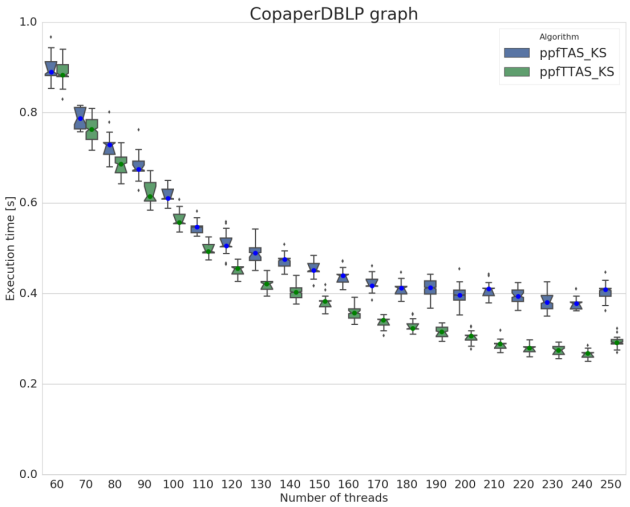


Fig. 3. Execution time of our PPF implementation on coPaperDBLP graph using Test-and-Set (TAS) and Test-and-Test-and-Set (TTAS). Karp-Sipser heuristic was used for the initial matching.

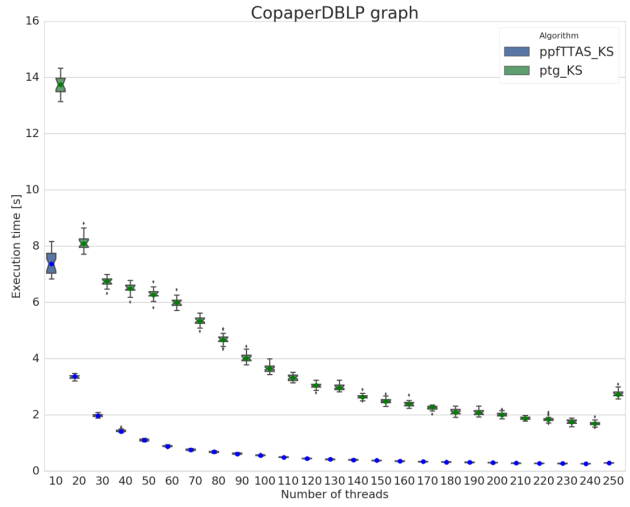


Fig. 4. Execution time of our PTG implementation compared to our PPF implementation. Both implementations use the Karp-Sipser heuristic for the initial matching.

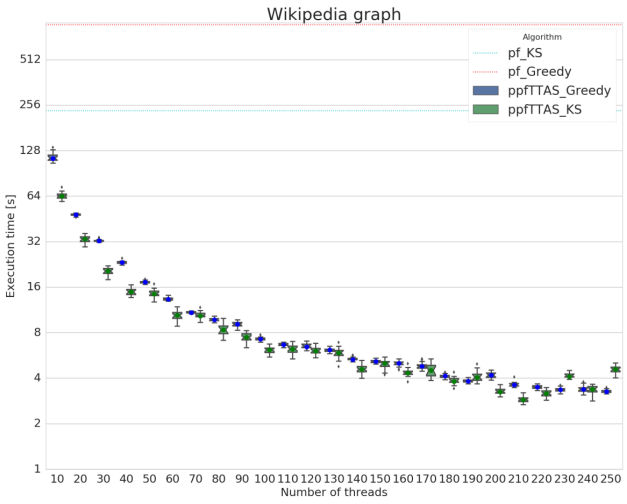


Fig. 5. Execution times of our PF and PPF implementations using the Karp-Sipser and the Greedy initial matching.

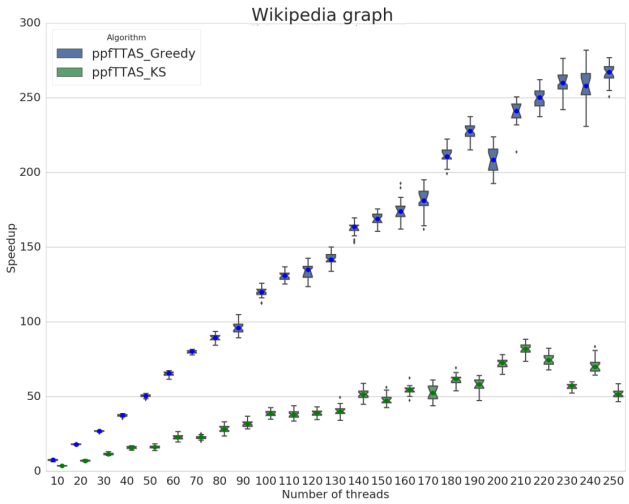


Fig. 6. Speedups of our PPF implementation using both the Karp-Sipser and the Greedy initial matching. The baseline is the sequential PF using the respective initial matching.

tion for this could be that our PTG implementation is not optimal.

Finally, we analyze how the initial matching impacts the performance of our PPF implementation. Figure 5 shows the execution times of our PPF implementation on the Wikipedia graph using both the Greedy (91% optimal) and the Karp-Sipser (99% optimal) initial matching, as well as the corresponding execution times for sequential PF. The quality of the initial matching has a substantial impact on the sequential version of the algorithm. For PPF however, the difference vanishes with an increasing number of threads. Figure 6 shows the difference in speedup. Graphs with bad initial matchings contain many augmenting paths. These can be found more efficiently by PPF with increased parallelism. This shows that PPF benefits from increased parallelism in situations where no good initial matching can be found.

5. CONCLUSIONS

We analyzed and implemented the PPF algorithm, building upon the pseudocode provided in the original paper [2]. We then designed and evaluated new optimizations for the PPF algorithm, which resulted in performance benefits over the original implementation. Our experimental results show that our approach using Test-and-Test-and-Set to lock a vertex outperforms the Test-and-Set approach used in the original PPF algorithm by Azad et al. [2]. Using a more clever way to update the *visited* array further improved the runtime of the algorithm.

The claim of Azad et al. [3] that PPF is not well suited for architectures with many thin cores could not be verified by our experiments on the Xeon Phi multicore processor. Our optimized PPF implementation yields a superlinear speedup compared to the sequential PF implementation and scales up to 240 threads. We have to mention however that the superlinear speedup is also due to caching effects on the Xeon Phi. The sequential PF algorithm only runs on a single core which is very weak compared to the resources of all cores combined.

While our implementation of the PTG algorithm also scales up to 240 threads, it is still an order of magnitude slower than our PPF implementation, independent of the number of threads used. We could thus not confirm that PTG is an improvement over PPF.

6. REFERENCES

[1] Alex Pothen and Chin-Ju Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 303–324, Dec. 1990.

[2] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen, "Multithreaded algorithms for maximum matching in bipartite graphs," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 860–872.

[3] Ariful Azad, Aydin Bulu, and Alex Pothen, "A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs," in *Proceedings of the 2015 IEEE International Parallel and Distributed*

Processing Symposium, Washington, DC, USA, 2015, IPDPS '15, pp. 1075–1084, IEEE Computer Society.

[4] "Intel xeon phi coprocessor: Software developers guide," https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html, Accessed: 2017-01-20.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms (3. ed.)*, MIT Press, 2009.

[6] Andrew V. Goldberg and Robert Endre Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, 1988.

[7] D. P. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Ann. Oper. Res.*, vol. 14, no. 1-4, pp. 105–123, June 1988.

[8] John E. Hopcroft and Richard M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973.

[9] Richard M. Karp and Michael Sipser, "Maximum matchings in sparse random graphs," in *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, 1981, pp. 364–375.

[10] "Boost edmonds: Maximum cardinality matching," http://www.boost.org/doc/libs/1_36_0/libs/graph/doc/maximum_matching.html, Accessed: 2017-01-05.