Efficient Parallel and External Matching

Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, Nodari Sitchinava

Karlsruhe Institute of Technology, Karlsruhe, Germany
marcelbirn@gmx.de,{osipov,sanders,christian.schulz}@kit.edu,nodari@ira.uka.de

Abstract. We study a simple parallel algorithm for computing matchings in a graph. A variant for unweighted graphs finds a maximal matching using linear expected work and $\mathcal{O}(\log^2 n)$ expected running time in the CREW PRAM model. Similar results also apply to External Memory, MapReduce and distributed memory models. In the maximum weight case the algorithm guarantees a 1/2-approximation. Although the parallel execution time is linear for worst case weights, an experimental evaluation indicates good scalabilty on distributed memory machines and on GPUs. Furthermore, the solution quality is very good in practice.

1 Introduction

A matching M of a graph $G = (V, E)$ is a subset of edges such that no two elements of M have a common end point. Many applications require the computation of matchings with certain properties, like being maximal (no edge can be added to M without violating the matching property), having maximum cardinality, or having maximum total weight $\sum_{e \in M} w(e)$. Although these problems can be solved optimally in polynomial time, optimal algorithms are not fast enough for many applications involving large graphs where we need near linear time algorithms. For example, the most efficient algorithms for graph partitioning rely on repeatedly contracting maximal matchings, often trying to maximize some edge rating function w. Refer to [13] for details and examples. For very large graphs, even linear time is not enough – we need a parallel algorithm with near linear work or an algorithm working in the external memory model [1].

Here we consider the following simple *local max* algorithm [12]: Call an edge locally maximal, if its weight is larger than the weight of any of its incident edges; for unweighted problems, assign unit weights to the edges. When comparing edges of equal weight, use tie breaking based on random perturbations of the edge weights. The algorithm starts with an empty matching M. It repeatedly adds locally maximal edges to M and removes their incident edges until no edges are left in the graph. The result is obviously a maximal matching (every edge is either in M or it has been removed because it is incident to a matched edge). The algorithm falls into a family of weighted matching algorithms for which Preis [24] shows that they compute a 1/2-approximation of the maximum weight matching problem. Hoepman [12] derives the local max algorithm as a distributed adaptation of Preis' idea. Based on this, Manne and Bisseling [18] devise sequential and parallel implementations. They prove that the algorithm

needs only a logarithmic number of iterations to compute maximal matchings by noticing that a maximal matching problem can be translated into a maximal independent set problem on the *line graph* which can be solved by Luby's algorithm [17]. However, this does not yield an algorithm with linear work since it is not proven that the edge set indeed shrinks geometrically.[1] Manne and Bisseling also give a sequential algorithm running in time $\mathcal{O}(m \log \Delta)$ where Δ is the maximum degree. On a NUMA shared memory machine with 32 processors (SGI Origin 3800) they get relative speedup < 6 for a complete graph and relative speedup ≈ 10 for a more sparse graph partitioned with Metis. Since this graph still has average degree ≈ 200 and since the speedups are not impressive, this is a somewhat inconclusive result when one is interested in partitioning large sparse graphs on a larger number of processors.

Parallel matching algorithms have been widely studied. There is even a book on the subject [16] but most theoretical results concentrate on work-inefficient algorithms. The only linear work parallel algorithms that we are aware of are randomized CRCW PRAM algorithms by Israeli and Itai [14] and Blelloch et al. [4]. We will call them IIM and BFSM, respectively. IIM runs in expected $\mathcal{O}(\log n)$ time and BFSM runs in $\mathcal{O}\big(\log^3 n\big)$ time with high probability.

Fagginger Auer and Bisseling [8] study an algorithm similar to [14] which we call red-blue matching (RBM) here. They implement RBM on shared memory machines and GPUs. They prove good shrinking behavior for random graphs, however, provide no analysis for arbitrary graphs.

Our contributions. We give a simple approach to implementing the local max algorithm that is easy to adapt to many models of computation. We show that for computing maximal matchings, the algorithm needs only linear work on a sequential machine and in several models of parallel computation (Section 2). Moreover it has low I/O complexity on several models of memory hierarchies.

Our CRCW PRAM local max algorithm matches the optimal asymptotic bounds of IIM. However, our algorithm is simpler (resulting in better constant factors), removes higher fraction of edges in each iteration (IIM's proof shows less than 5% per iteration, while we show at least 50%) and our analysis is a lot simpler. We also provide the first CREW PRAM algorithm which performs linear work and runs in expected $\mathcal{O}\big(\log^2 n\big)$ time.[2]

In Section 3 we explain how to implement local max on practical massively parallel machines such as MPI clusters and GPUs. Our experiments indicate that the algorithm yields surprisingly good quality for the weighted matching problem and runs very efficiently on sequential machines, clusters with reasonably partitioned input graphs, and on GPUs. Compared to RBM, the local max implementations remove more edges in each iteration and provide better quality results for the weighted case. Some of the results presented here are from the diploma thesis of Marcel Birn [2].

[1] Manne and Bisseling show such a shrinking property under an assumption that unfortunately does not hold for all graphs.

[2] While a generic simulation of IIM on the CREW PRAM model will result in a $\mathcal{O}\big(\log^2 n\big)$ time algorithm, the simulation incurs $\mathcal{O}(n \log n)$ work due to sorting.

2 Parallel Local Max

Our central observation is:

Lemma 1. *Each iteration of the local max algorithm for the unit weight case removes at least half of the edges in expectation.*

Proof. Consider the graph remaining in the currently considered iteration where $d(v)$ denotes the degree of a node and m the remaining number of edges. Consider the end point at node v of an edge $\{u, v\}$ as *marked* if and only if some edge incident to v becomes matched. Note that an edge is removed if and only if at least one of its end points becomes marked. Now consider a particular edge $e = \{u, v\}$. Since any of the $d(u) + d(v) - 1$ edges incident to u and v is equally likely to be locally maximal, e becomes matched with probability $1/(d(u) + d(v) - 1)$.[3] If e is matched, this event is responsible for setting $d(u) + d(v)$ marks, i.e., the expected number of marks caused by an edge is $(d(u)+d(v))/(d(u)+d(v)-1) \geq 1$. By linearity of expectation, the total expected number of marks is at least m. Since no edge can have more than two marks, at least $m/2$ edges have at least one mark and are thus deleted.[4] ∎

Métivier et al. [22] uses a similar proof technique to define "preemptive removal" of nodes for distributed maximal independent set problem.

Assume now that each iteration can be implemented to run with work linear in the number of surviving edges (independent of the number of nodes). Working naively with the expectations, this gives us a logarithmic number of rounds and a geometric sum leading to linear total work for computing a maximal matching. This can be made rigorous by distinguishing *good* rounds with at least $m/4$ matched edges and bad rounds with less matched edges. By Markov's inequality, we have a good round with constant probability. This is already sufficient to show expected linear work and a logarithmic number of expected rounds. We skip the details since this is a standard proof technique and since the resulting constant factors are unrealistically conservative. An analogous calculation for median selection can be found in [20, Theorem 5.8]. One could attempt to show a shrinking factor close to $1/2$ rigorously by showing that large deviations (in the wrong direction) from the expectation are unlikely (e.g., using Martingale tail bounds). However this would still be a factor two away from the more heuristic argument in Footnote 4 and thus we stick to the simple argument.

There are many ways to implement an iteration which of course depend on the considered model of computation.

Sequential Model. For each node v maintain a candidate edge $C[v]$, originally initialized to a dummy edge with zero weight. In an iteration go through all

[3] For this to be true, the random noise added for tie breaking needs to be renewed in every iteration. However, in our experiments this had no noticeable effect.

[4] This is a conservative estimate. Indeed, if we make the (over)simplified assumption that m marks are assigned randomly and independently to $2m$ end points, then only one fourth of the edges survives in expectation. Interestingly, this is the amount of reduction we observe in practice – even for the weighted case.

remaining edges $e = \{u, v\}$ three times. In the first pass, if $w(e) > w(C[u])$ set $C[u] := e$ (add random perturbation to $w(e)$ in case of a tie). If $w(e) > w(C[v])$ set $C[v] := e$. In the second pass, if $C[u] = C[v] = e$ put e into the matching M. In the third pass, if u or v is matched, remove e from the graph. Otherwise, reset the candidate edge of u and v to the dummy edge. Note that except for the initialization of C which happens only once before the first iteration, this algorithm has no component depending on the number of nodes and thus leads to linear running time in total if Lemma 1 is applied.

CRCW PRAM Model. In the most powerful variant of the *Combining CRCW PRAM* that allows concurrent writes with a maximum reduction for resolving write conflicts, the sequential algorithm can be parallelized directly running in constant time per iteration using m processors.

MapReduce Model. The CRCW PRAM result together with the simulation result of Goodrich et al. [11] immediately implies that each iteration of local max can be implemented in $\mathcal{O}(\log_M n)$ rounds and $\mathcal{O}(m \log_M n)$ communication complexity in the MapReduce model, where M is the size of memory of each compute node. Since typical compute nodes in MapReduce have at least $\Omega(m^\epsilon)$ memory [15], for some constant $\epsilon > 0$, each iteration of local max can be performed in MapReduce in constant rounds and linear communication complexity.

External Memory Models. Using the PRAM emulation techniques for algorithms with geometrically decreasing input size from [5, Theorem 3.2] the above algorithm can be implemented in the external memory [1] and cache-oblivious [9] models in $\mathcal{O}(sort(m))$ I/O complexity, which seems to be optimal.

2.1 $\mathcal{O}(\log^2 n)$ work-optimal CREW solution

In this section, we present a $\mathcal{O}(\log^2 n)$ CREW PRAM algorithm, which incurs only $\mathcal{O}(m)$ work.

Consider the following representation of the graph $G = (V, E)$. Let V be a totally ordered set, i.e., given two vertices $u, v \in V$ we can uniquely determine whether $u < v$ or not. Let $\mathtt{E}$ be an array of undirected edges with each entry $\mathtt{E}[k]$ storing all the information of a single edge $\{u, v\} \in E$, i.e., vertex endpoints u and v, its weight or any other auxiliary data. Let $\mathtt{A}$ be an array of tuples (v, e_k), where $v \in V$ and e_k is the *pointer* to $\mathtt{E}[k]$ representing the edge $\{u, v\}$. Let $\mathtt{A}$ be sorted by the first entry, i.e. all tuples (v, e_k) pointing to the edges incident on the same vertex v are in contiguous space in $\mathtt{A}$.

Note that any edge $\mathtt{E}[k] = \{u, v\}$ contains two corresponding entries in $\mathtt{A}$ pointing to it: (u, e_k) and (v, e_k). During our algorithm, a processor responsible for (u, e_k) might need to find and update entry (v, e_k) (and vice versa). The following lemma describes how to compute for each entry (u, e_k) the index of the corresponding entry (v, e_k) in $\mathtt{A}$.

Lemma 2. *For every edge $\mathtt{E}[k] = \{u, v\}$ entries $\mathtt{A}[i] = (u, e_k)$ and $\mathtt{A}[j] = (v, e_k)$ of $\mathtt{A}$ can compute each other's index in $\mathtt{A}$ in $\mathcal{O}(1)$ time and $\mathcal{O}(|\mathtt{A}|)$ work in the CREW PRAM model.*

Proof. For every $\mathtt{E}[k] = \{u, v\}$ we show how $\mathtt{A}[j] = (v, e_k)$ can compute the index of the corresponding entry $\mathtt{A}[i] = (u, e_k)$ in $\mathtt{A}$ for $u < v$. The indices for the other half of the entries are computed symmetrically.

The algorithm proceeds in two phases. In the first phase, each entry $\mathtt{A}[i] = (u, e_k)$, stores the value i in $\mathtt{E}[k] = \{u, v\}$ iff $u < v$. In the second phase, each entry $\mathtt{A}[j] = (v, e_k)$ reads the stored value i from $\mathtt{E}[k] = \{u, v\}$ iff $v > u$.

If we assign a separate processor to each entry of $\mathtt{A}$, each processor performs only $\mathcal{O}(1)$ steps. Moreover, there are no concurrent writes because, at each step only one of the two vertices of the edge e_k writes to $\mathtt{E}[k]$. Note, we need a concurrent read to $\mathtt{E}[k] = \{u, v\}$ to determine the relative order of u and v. ∎

Lemma 3. *Using our graph representation, each node v in the graph can apply an associative operator $\oplus$ to all edges incident on v in $\mathcal{O}(\log |\mathtt{A}|)$ time and $\mathcal{O}(|\mathtt{A}|)$ work on the CREW PRAM model.*

Proof. First, we read for each entry $(v, e_k) \in \mathtt{A}$ the value from $\mathtt{E}[k]$ on which to apply the operator. Next, we run segmented prefix sums with $\oplus$ operator on these values, where segments are the portions of $\mathtt{A}$ representing the neighbors of a single node and are easily identified from the definition of $\mathtt{A}$. Finally, each entry of $(v, e_k) \in \mathtt{A}$ applies its result of segmented prefix sums to the edge $\mathtt{E}[k]$, while using the technique of Lemma 2 to avoid write conflicts. Each step of the algorithm can be implemented in $\mathcal{O}(\log |\mathtt{A}|)$ time using $\mathcal{O}(|\mathtt{A}|)$ work. ∎

Now we are ready to describe the solution to the matching problem. We perform the following in each phase of the local max algorithm.

1. Each edge $\mathtt{E}[k]$ picks a random weight w_k.
2. Using Lemma 3, each vertex v identifies k' such that $\mathtt{E}[k']$ is the heaviest edge incident on v by applying the associative operator MAX to the edge weights picked in the previous step.
3. Using Lemma 2, each entry $(v, e_{k'})$ checks if $\mathtt{E}[k'] = \{u, v\}$ is also the heaviest incident edge on u. If so, the smaller of u and v adds $e_{k'}$ to the matching and sets the deletion flag $f = 1$ on $\mathtt{E}[k']$.
4. Using Lemma 3, each entry $(v, e_{k'})$ spreads the deletion flag over all edges incident on v by applying MAX associative operator on the deletion flags of incident edges on v. Thus, if at least one edge incident on v was added to the matching, all edges incident on v will be marked for deletion.
5. Now we must prepare the graph representation for the next phase by removing all entries of $\mathtt{E}$ and $\mathtt{A}$ marked for deletion, compacting $\mathtt{E}$ and $\mathtt{A}$ and updating the pointers of $\mathtt{A}$ to point to the compacted entries of $\mathtt{E}$. To perform the compaction, we compute for each entry $\mathtt{E}[k]$, how many entries $\mathtt{E}[i]$ and $\mathtt{A}[i], i \leq k$ must be deleted. This can be accomplished using parallel prefix sums on the deletion flags of each entry in $\mathtt{E}$ and $\mathtt{A}$. Let the result of prefix sums for edge $\mathtt{E}[k]$ be d_k and for entry $\mathtt{A}[i]$ be r_i. Then $k - d_k$ is the new address of the entry $\mathtt{E}[k]$ and $i - r_i$ is the new address of $\mathtt{A}[i]$ once all edges marked for deletion are removed.

6. Each entry $\mathtt{E}[k]$ that is not marked for deletion copies itself to $\mathtt{E}[k - d_k]$. The corresponding entry $(v, e_k) \in \mathtt{A}$ updates itself to point to the new entry $\mathtt{E}[k - d_k]$, i.e., (v, e_k) becomes (v, e_{k-d_k}), and copies itself to $\mathtt{A}[i - r_i]$.

The algorithm defines a single phase of the local max algorithm. Each step of the phase takes at most $\mathcal{O}(|\mathtt{A}|) = \mathcal{O}(m)$ work and $\mathcal{O}(\log |\mathtt{A}|) = \mathcal{O}(\log m) = \mathcal{O}(\log n)$ time in the CREW PRAM model. Over $\mathcal{O}(\log m)$ phases, each with geometrically decreasing number of edges, the local max algorithm takes overall $\mathcal{O}\big(\log^2 n\big)$ time and $\mathcal{O}(m)$ work in the CREW PRAM model.

3 Implementations and Experiments

We now report experiments focusing on computing approximate maximum weight matchings. We consider the following families of inputs, where the first two classes allow comparison with the experiments from [19].

Delaunay Instances are created by randomly choosing $n = 2^x$ points in the unit square and computing their Delaunay triangulation. Edge weights are Euclidean distances.

Random graphs with $n := 2^x$ nodes, αn edges for $\alpha = \{4, 16, 64\}$, and random edge weight chosen uniformly from $[0, 1]$.

Random geometric graphs with 2^x nodes (rgg$x$). Each vertex is a random point in the unit square and edges connect vertices whose Euclidean distance is below $0.55 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost connected.

Florida Sparse Matrix. Following [8] we use 126 symmetric non-0/1 matrices from [6] using absolute values of their entries as edge weights, see [3] for the full list. The number of edges of the resulting graphs $m \in (0.5 \ldots 16) \times 10^6$. See [3] for a detailed list.

Graph Contraction. We use the graphs considered by KaFFPa for partitioning graphs from the 10'th DIMACS Implementation Challenge [25].

We compare implementations of local max, the red-blue algorithm from [8] (RBM) (their implementation), heavy edge matching (HEM) [10], greedy, and the global path algorithm (GPA) [19]. HEM iterates through the nodes (optionally in random order) and matches the heaviest incident edge that is nonadjacent to a previously matched edge. The greedy algorithm sorts the edges by decreasing weights, scans them and inserts edges connecting unmatched nodes into the matching. GPA refines greedy. It greedily inserts edges into a graph G_2 with maximum degree two and no odd cycles. Using dynamic programming on the resulting paths and even cycles, a maximum weight matching of G_2 is computed.

Algorithms involving sorting use standard STL Visual Studio 2010 sort routine.

Sequential and shared-memory parallel experiments were performed on an Intel i7 920 2.67 GHz quad-core machine with 6 GB of memory. We used a commodity NVidia Fermi GTX 480 featuring 15 multiprocessors, each containing

32 scalar processors, for a total of 480 CUDA cores on chip. The GPU RAM is 1.5 GB. We compiled all implementations using CUDA 4.2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

3.1 Sequential Speed and Quality

We compare solution quality of the algorithms relative to GPA. Via the experiments in [19] this also allows some comparison with optimal solutions which are only a few percent better there. Figure 1 shows the quality for Delaunay graphs (where GPA is about 5 % from optimal [19]). We see that local max achieves almost the same quality as greedy which is only about 2 % worse than GPA. HEM, possibly the fastest nontrivial sequential algorithm is about 13 % away while RBM is 14 % worse than GPA, i.e., HEM and RBM almost double the gap to optimality of local max. Looking at the running times, we see that HEM is the fastest (with a surprisingly large cost for actually randomizing node orders) followed by local max, greedy, GPA, and RBM. From this it looks like HEM, local max, and GPA are the winners in the sense that none of them is dominated by another algorithm with respect to both quality and running time. Greedy has similar quality as local max but takes somewhat longer and is not so easy to parallelize. RBM as a sequential algorithm is dominated by all other algorithms. Perhaps the most surprising thing is that RBM is fairly slow. This has to be taken into account when evaluating reported speedups. We suspect that a more efficient implementation is possible but do not expect that this changes the overall conclusion. In [3] we report similar results for the rgg instances and random graphs.

Looking at the wide range of instances in the Florida Sparse Matrix collection leads to similar but more complicated conclusions. Figure 2 shows the solution qualities for greedy, local max, RBM and HEM relative to GPA. RBM and even more so HEM shows erratic behavior with respect to solution quality. Greedy

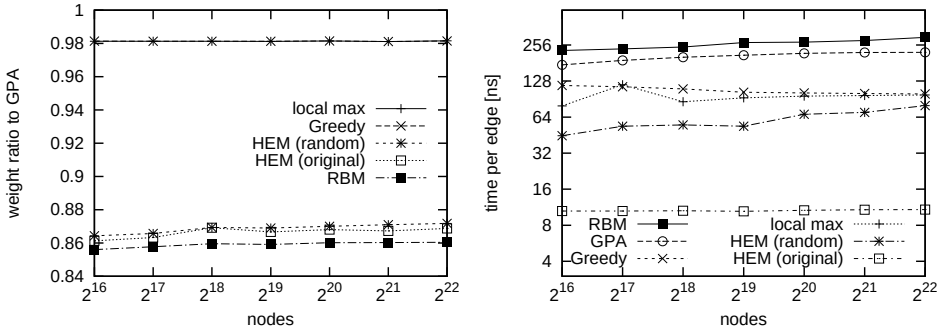

Fig. 1. Ratio of the weights computed by GPA and other algorithms for Delaunay instances and running times.

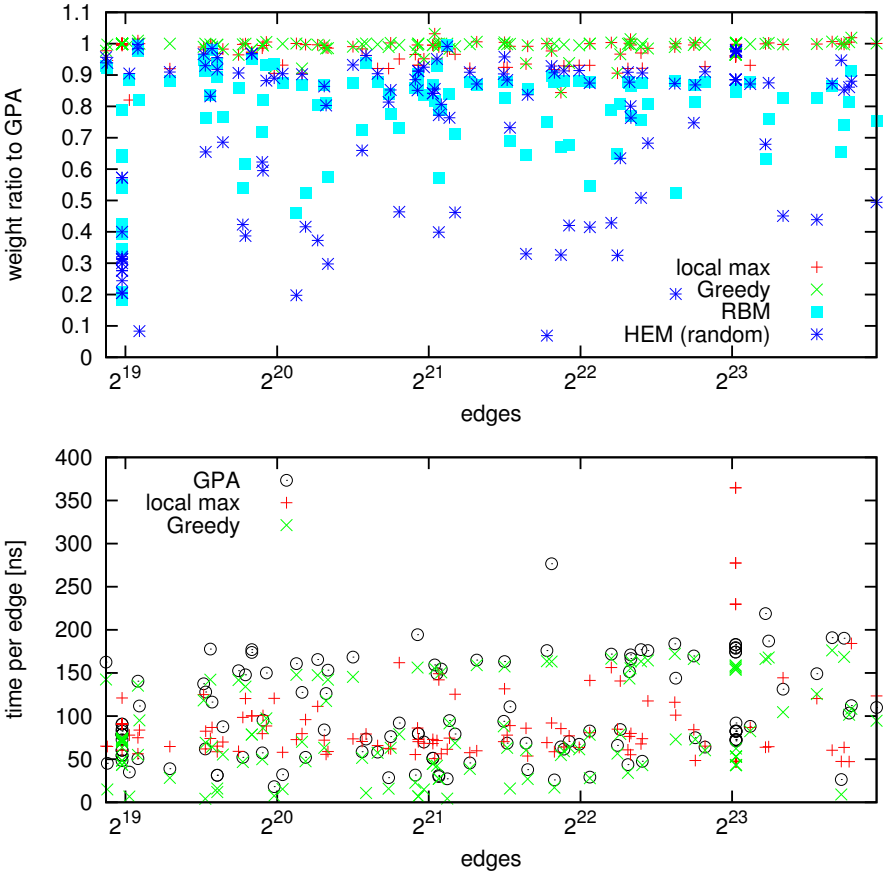

Fig. 2. Ratio of the weights computed by GPA and other sequential algorithms for sparse matrix instances and running time.

and local max are again very close to GPA and even closer to each other although there is a sizable minority of instances where greedy is somewhat better than local max. Looking at the corresponding running times one gets a surprisingly diverse picture. HEM which is again fastest and RBM which is again dominated by local max are not shown. There are instances where local max is considerably faster then greedy and vice versa. A possible explanation is that greedy becomes quite fast when there is only a small number of different edge weights since then sorting is quite an easy problem.

Experiments on the graph contraction instances in [2] show local max about 1 % away from GPA. For these instances the average fraction of remaining edges after an iteration is well below 25 %. Notable exceptions are the graphs *add20*

and *memplus* which both represent VLSI circuits. Nevertheless, none of the instances considered required more than 10 iterations.

3.2 Distributed Memory Implementation

Our distributed memory parallelization (using MPI) on p processing elements (PEs or MPI processes) assigns nodes to PEs and stores all edges incident to a node locally. This can be done in a load balanced way if no node has degree exceeding m/p. The second pass of the basic algorithm from Section 2 has to exchange information on candidate edges that cross a PE boundary. In the worst case, this can involve all edges handled by a PE, i.e., we can expect better performance if we manage to keep most edges locally. In our experiments, one PE owns nodes whose numbers are a consecutive range of the input numbers. Thus, depending on how much locality the input numbering contains we have a highly local or a highly non-local situation. We have not considered more sophisticated ways of node assignment so far since our motivating application is graph partitioning/clustering where almost by definition we initially do *not* know which nodes form clusters – this is the intended *output*. Since Lemma 1 also applies to the subgraph relevant for a particular PE, we can expect that the graph shrinks fairly uniformly over the entire network.

We performed experiments on two different clusters at the KIT computing center both using compute-nodes with two quad-core processors each. Refer to [2] for details. We ran experiments with up 128 compute-nodes corresponding to 1024 cores with one MPI process per core.

Figure 3 illustrates how our distributed local max implementation scales for the random geometric graphs *rgg23* and *rgg24* (using random edge weights)

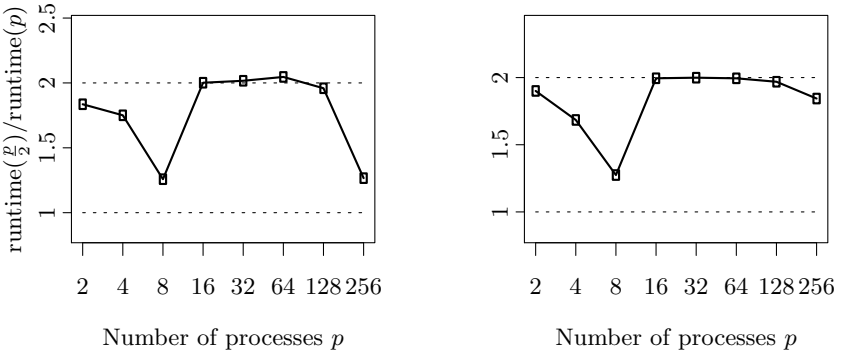

Fig. 3. Scaling results of the parallel local max algorithm on random geometric graphs with random edge weights. Left: rgg23 (≈63 million edges). Right: rgg24 (≈ 132 million edges).

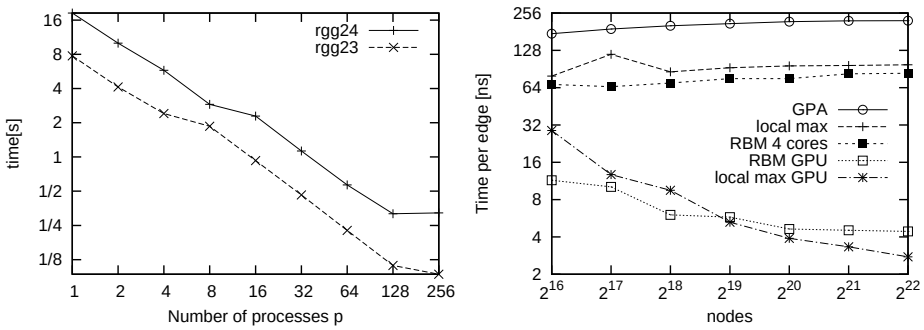

Fig. 4. Running time for distributed memory implementation on rgg23 and rgg24 (left). Time per edge of sequential and GPU algorithms for Delaunay instances (right).

which have fairly good locality. We plot the decrease in running time for successive doubling of p, i.e., a value of two stands for perfect relative speedup for this step and a value below one means that parallelization no longer helps. We see values slightly below two for the steps $1 \to 2$ and $2 \to 4$ which is typical behavior of multicore algorithms when cores compete for resources like memory bandwidth. For $p = 8$ we start to use two compute-nodes (with 4 active cores each) and consequently we see the largest dip in efficiency. Beyond that, we have almost perfect scaling until the problem instance becomes too small. We have similar behavior for other graphs with good locality. For graphs with poor locality, efficiency is not very good. However the ratios stay above one for a very long time, i.e., it pays to use parallelism when it is available anyway. This is the situation we have when partitioning large graphs for use on massively parallel machines. Considering that the matching step in graph partitioning is often the least work intensive one in multi-level graph partitioning algorithms we conclude that local max might be a way to remove a sequential bottleneck from massively parallel graph partitioning. See Figure 4 (left) for the absolute timing and refer to [2] for additional data.

3.3 GPU Implementation

Our GPU algorithm is a fairly direct implementation of the CRCW algorithm. We reduce the algorithm to the basic primitives such as segmented prefix sum, prefix sum and random gather/scatter from/to GPU memory. As a basis for our implementation we use back40computing library by Merrill [21].

Figure 4 (right) compares the running time of our implementation with GPA, sequential local max, the RBM algorithm parallelized for 4 cores, and its GPU parallelization from [8]. While the CPU implementation has troubles recovering from its sequential inefficiency and is only slightly faster than even sequential local max, the GPU implementation is impressively fast in particular for small graphs. For large graphs, the GPU implementation of local max is faster. Since

local max has better solution quality, we consider this a good result. Our GPU code is up to 35 times faster than sequential local max. We may also be able to learn from the implementation techniques of RBM GPU for small inputs in future work.

For random geometric graphs and random graphs, we get similar behavior (see [3] for details). The results for rgg are slightly worse for GPU local max – speedup is up to 24 over sequential local max and a speed advantage over GPU RBM only for the very largest inputs. As for random graphs, the denser the graph the larger is our speedup over the sequential and GPU RBM implementations. Thus, for $\alpha = 64$ our implementation is faster than GPU RBM already for $n = 2^{15}$. For $n = 2^{18}$ it is 65% faster than GPU RBM and 30 times faster than the sequential local max.

4 Conclusions And Future Work

The local max algorithm is a good choice for parallel or external computation of maximal and approximate maximum weight matchings. On the theoretical side it is provably efficient for computing maximal matchings and guarantees a 1/2-approximation. On the practical side it yields better quality at faster speed than several competitors including the greedy algorithm and RBM. Somewhat surprisingly it is even attractive as a sequential algorithm, outperforming HEM with respect to solution quality and other algorithms with respect to speed.

We have learned about the linear work algorithm by Blelloch et al. [4] from an anonymous reviewer during the review process. While our algorithm guarantees better expected asymptotic runtime, the practical results in [4] seem to be quite promising. However, lack of optimized shared memory implementation of our algorithm for multicores, use of different compilers and operating systems, and different set of test cases makes a thorough and fair comparison of the two algorithms unfeasible in the short period of time and is left for future work.

Many interesting questions remain. Can we omit re-randomization of edge weights when computing maximal matchings? The result of Blelloch et al. [4] partially answers this question by performing randomization only once at the expense of the performance guaratee. Is there a linear work parallel algorithm with polylogarithmic execution time that computes 1/2-approximations (or any other constant factor approximation). Can we even do 2/3-approximations with linear work in parallel [7, 23]?

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. Marcel Birn. Engineering fast parallel matching algorithms. Diploma Thesis, Karlsruhe Institute of Technology, 2012.
3. Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. *CoRR*, abs/1302.4587, 2013.

4. G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, pages 308–317, 2012.
5. Y.-J.n Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.
6. T. Davis. The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, 2008.
7. D. Drake and S. Hougardy. Improved linear time approximation algorithms for weighted matchings. In *APPROX, LNCS 2764*, pages 14–23, 2003.
8. B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the Multicore - Challenge II*, pages 108–119. 2012.
9. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.
10. V. Kumar G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
11. M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching and simulation in the mapreduce framework. In *ISAAC*, pages 374–383, 2011.
12. Jaap-Henk Hoepman. Simple distributed weighted matchings. *CoRR*, cs.DC/0410047, 2004.
13. M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. pages 1–12, 2010.
14. Amos Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.
15. Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.
16. M. Karpinski and W. Rytter. *Fast parallel algorithms for graph matching problems*, volume 98. Clarendon Press, 1998.
17. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.
18. F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *PPAM*, volume 4967, pages 708–717, 2007.
19. J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Exp. Algorithms (WEA)*, 2007.
20. K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.
21. Duane Merrill. back40computing: fast and efficient software primitives for GPU computing. http://code.google.com/p/back40computing/.
22. Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari. An optimal bit complexity randomized distributed MIS algorithm. In *SIROCCO*, pages 323–337, 2009.
23. S. Pettie and P. Sanders. A simpler linear time $2/3-\epsilon$ approximation for maximum weight matching. Technical Report MPI-I-2004-1-002, MPII, 2004.
24. R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS*, pages 259–269, 1999.
25. P. Sanders and C. Schulz. High Quality Graph Partitioning. In *Proceedings of the 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*, pages 1–17. AMS, 2013.