A multithreaded algorithm for network alignment via approximate matching

Arif M. Khan, David F. Gleich, Alex Pothen
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907-2107
Email: {khan58,dgleich,apothen}@purdue.edu

Mahantesh Halappanavar
Pacific Northwest National Laboratory
Email: Mahantesh.Halappanavar@pnnl.gov

***Abstract*—Network alignment is an optimization problem to find the best one-to-one map between the vertices of a pair of graphs that overlaps as many edges as possible. It is a relaxation of the graph isomorphism problem and is closely related to the subgraph isomorphism problem. The best current approaches are entirely heuristic and iterative in nature. They generate real-valued heuristic weights that must be rounded to find integer solutions. This rounding requires solving a bipartite maximum weight matching problem at each iteration in order to avoid missing high quality solutions. We investigate substituting a parallel, half-approximation for maximum weight matching instead of an exact computation. Our experiments show that the resulting difference in solution quality is negligible. We demonstrate almost a 20-fold speedup using 40 threads on an 8 processor Intel Xeon E7-8870 system and now solve real-world problems in 36 seconds instead of 10 minutes.**

I. Introduction

The network alignment problem addresses the question: given two graphs, what's the best way of matching their vertices in order to make the graphs overlap in as many edges as possible? See Figure 1 for a guiding figure that illustrates the problem setup – we'll formally define the problem in Section II. This problem is not new. It's a generalization of the edge-subgraph isomorphism problem. However, with the growth of network and graph datasets, applications of network alignment have exploded over the past decade.

One of the first uses of network alignment was in pattern recognition and computer vision. See Conte et al. for a comprehensive survey of applications in that domain [1]. Network alignment methods have also been used extensively in ontology matching [2] and database schema matching applications [3]. The largest problems tend to come from ontology alignment applications.

An important new application is in bioinformatics where the problem is how to identify similarities between protein-protein interaction networks from different species [4], [5]. Indeed, this applications has sparked a range of novel methods and software for the problem [6]–[11]. These contributions were recently profiled by Atias and Sharan [12]. Because network alignment solutions usually need to be interpreted, verified, and refined by a human, algorithms for the problem need to be run, ideally, within a few seconds.

The underlying network alignment problem is NP-hard, and there is no known approximation algorithm. Current methods are entirely heuristic, albeit principled heuristics. They have been experimentally tested and shown to perform well. Some methods, such as Klau's [7] (see Section III for more details), do come with an *a posteriori* approximation bound. In a recent study by one of the authors [13], the best two methods were a belief-propagation method [14] and Klau's method [7]. Both of these methods are iterative in nature. Some of the work in an iteration has the flavor of a parallel matrix computation, while the remaining work involves solving a maximum-weight bipartite matching problem. (All matching problems in this paper are maximum edge-weight matchings.) This combination makes network alignment procedures a fascinating mixture of matrix methods and combinatorial algorithms and provides a unique performance challenge. In particular, solving the matching problem usually dominates the runtime of the methods. Given the lack of good parallel algorithms for the problem, this aspect has limited the ability of network alignment methods to capitalize on modern multicore processors.

In this paper, we investigate using an approximate maximum-weight bipartite matching procedure to develop parallel methods for network alignment on multicore NUMA architectures. As we shall see, these methods scale to roughly 40 cores with only a small change in solution quality. The remainder of the paper proceeds as follows. First, in Section II, we formally introduce the network alignment problem as

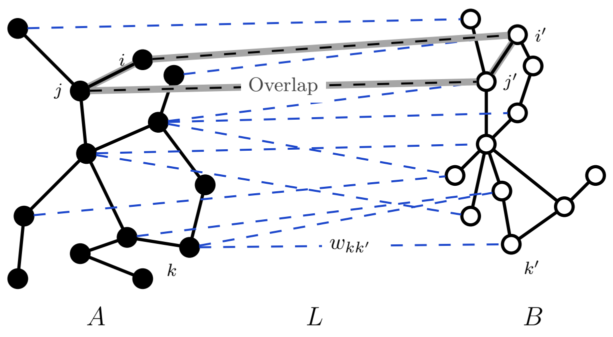


Fig. 1. From [14]. In the network alignment problem, we want to find a subset of edges from L that form a matching between A and B with as many overlaps as possible.

an integer quadratic program. We next describe the belief-propagation method and the Klau's Lagrangian relaxation method in Section III. For both methods, we discuss implementation and parallelization strategies for everything except the bipartite matching in Section IV. In Section V, we explain the multi-core parallel approximation algorithm for bipartite maximum-weight matching from [15].

In the next portion of the paper, we experimentally investigate the resulting procedures. The issue we address in Section VII is how much of the solution quality is lost when using the approximation algorithms. We find only a marginal change in the solution quality using the approximation algorithms. The second issue we address in Section VIII is how well resulting algorithms scale on multi-core processors. We find that these procedures have good scaling up to 40-cores on an Intel Xeon E7-8870.

In the spirit of reproducible research, we make our codes and most of the data available:

http://www.cs.purdue.edu/~dgleich/codes/netalignmc/

II. The Network Alignment Problem

The formal statement of the network alignment problem is:

Network Alignment. Let $A = (V_A, E_A)$ and $B = (V_B, E_B)$ be two undirected graphs and their vertex and edge sets, respectively. Let L be a weighted bipartite graph between the vertex sets: $L = (V_A \cup V_B, E_L, w)$. Let α and β be two positive constants. All of A, B, L and α, β are problem inputs, which we assume are given. A matching M in L is a subset of E_L with at most one edge incident on each vertex. We say that an edge (i, j) in E_A is *overlapped* with an edge (i', j') in E_B if both (i, i') and (j, j') are in a matching M. The network alignment problem is to find a matching M in L that maximizes:

$$\begin{aligned}&\alpha \cdot \text{weight of matching subset } M\ + \\ &\quad \beta \cdot \text{number of overlapped edges in } A \text{ and } B.\end{aligned}$$

This problem generalizes the maximum common edge subgraph problem, which corresponds to setting L to the full bipartite graph and $\alpha = 0, \beta = 1$.

The computational methods we utilize are based on an integer quadratic program for this problem. Related integer programs often arise in assignment or matching problems [16]. Let $\mathbf{w}$ be a weight vector for an arbitrary ordering of the edges in L. Note that $\mathbf{w}$ has length $|E_L|$. We use $w_{i,i'}$ to indicate an element of the vector, which is a weight on the edge matching $i \in V_A$ to $i' \in V_B$. Let $\mathbf{x}$ be a length $|E_L|$ indicator vector over the edges of L in that same ordering: $x_{i,i'} = 1$ if the edge is in the matching and 0 otherwise. The weight of the matching subset is then given by the inner-product:

$$\mathbf{x}^T\mathbf{w} = \sum_{(i,i') \in E_L} x_{i,i'} w_{i,i'}.$$

To enforce that $\mathbf{x}$ corresponds to a matching, we use the following linear matching constraints:

$$\begin{aligned}\sum_{i' \in V_B \text{ where } (i,i') \in E_L} x_{i,i'} \le 1 \quad &\text{for all } i \in V_A \quad \text{and} \\ \sum_{i \in V_A \text{ where } (i,i') \in E_L} x_{i,i'} \le 1 \quad &\text{for all } i' \in V_B.\end{aligned}$$

In the rest of the paper, we write these linear constraints as $\mathbf{Cx} \le \mathbf{e}$, where $\mathbf{C}$ is the $(|V_A| + |V_B|)$-by-$|E_L|$ node-edge incidence matrix of the graph L and $\mathbf{e}$ is the vector of all ones. The integer program for a max-weight bipartite matching problem is then:

$$\begin{aligned}&\text{maximize} && \mathbf{x}^T\mathbf{w} \\ &\text{subject to} && \mathbf{Cx} \le \mathbf{e}, x_{i,i'} \in \{0,1\}.\end{aligned}$$

In order to compute the number of overlapped edges in A and B, we introduce the matrix $\mathbf{S}$. The rows and columns of $\mathbf{S}$ correspond to edges in E_L, thus $\mathbf{S}$ is $|E_L|$-by-$|E_L|$, and $S_{(i,i'),(j,j')}$ denotes a particular entry in the matrix. If we set $S_{(i,i'),(j,j')} = 1$ if (i, j) and (i', j') are both edges in graphs A and B respectively, and otherwise set $S_{(i,i'),(j,j')} = 0$, then the number of overlapped edges in A and B is $\mathbf{x}^T\mathbf{Sx}/2$. And hence, a quadratic integer program for network alignment is:

$$\begin{aligned}&\text{maximize} && \alpha\mathbf{x}^T\mathbf{w} + \tfrac{\beta}{2}\mathbf{x}^T\mathbf{Sx} \\ &\text{subject to} && \mathbf{Cx} \le \mathbf{e}, x_{i,i'} \in \{0,1\}.\end{aligned} \tag{NAQP}$$

In general, the matrix $\mathbf{S}$ is indefinite. Thus, the complexity of the problem does not change if we relax the integrality constraint to a bounded real-valued variable because solving a nonconvex quadratic program is still NP-hard. However, an equivalent mixed-integer linear program exists. Let $\mathbf{Y}$ be a matrix with the same sparsity pattern as $\mathbf{S}$ such that $Y_{(i,i'),(j,j')} \le x_{i,i'}$ and $Y_{(i,i'),(j,j')} \le x_{j,j'}$. The mixed integer program:

$$\begin{aligned}&\text{maximize} && \alpha\mathbf{x}^T\mathbf{w} + \tfrac{\beta}{2}\mathbf{e}^T\mathbf{Ye} \\ &\text{subject to} && \mathbf{Cx} \le \mathbf{e}, \quad x_{i,i'} \in \{0,1\} \\ &&& Y_{(i,i'),(j,j')} \le x_{i,i'}, Y_{(i,i'),(j,j')} \le x_{j,j'} \\ &&& \quad \text{for all } (i,i'),(j,j') \text{ where } \mathbf{S} \text{ is non-zero}\end{aligned} \tag{MILP}$$

encodes the same objective. This type of modification was first used for quadratic assignment problems by Lawler [17], and it is the basis of Klau's method described next.

III. Network Alignment Methods

In the previous section, we formally defined the network alignment problem and then stated it as both an integer quadratic program and a mixed integer linear program (MILP). Both of these are just as hard to solve. In this paper, we study two fast heuristics from a recent study by one of the authors [13].

A straightforward heuristic to generate a solution is to relax the integrality constraint in the MILP form. Then, solving the resulting linear program will compute a real-valued score for each edge in L, but won't necessarily result in a matching. We

TABLE I
NOTATION FOR THE ALGORITHMS

$\text{bound}_{l,u}(x) = \begin{cases} l & x \leq l \\ x & l < x < u \\ u & x \geq u \end{cases}$	**tril**$(\mathbf{M})$, **triu**$(\mathbf{M})$: the lower and upper triangular portions of $\mathbf{M}$
bipartite_match$(\mathbf{w})$	returns the matching indicator vector $\mathbf{x}$ for a max-weight bipartite matching problem on graph L with weight $\mathbf{w}$; in the experiments, we utilize the scalable, parallel approximation algorithm described in Section V instead of an exact routine
round_heuristic$(\mathbf{g})$	for a real-valued heuristic approximation to the solution of a network alignment problem, compute $\mathbf{x}$ = bipartite_match$(\mathbf{g})$, and then evaluate the objective function $\alpha\mathbf{w}^T\mathbf{x} + \frac{\beta}{2}\mathbf{x}^T\mathbf{S}\mathbf{x}$; also keep track of which $\mathbf{g}$ produced the largest objective

can explicitly round the solution to a matching by using these scores as weights for a max-weight bipartite matching problem. Both of the algorithms below outperform this procedure, and have relatively good parallelization potential in contrast to solving large, sparse linear programs.

In the following pseudocodes, we'll utilize the notation from Table I for convenience.

A. Klau's Matching Relaxation

Klau's method for network alignment builds on a different type of relaxation of the linear program. We refer readers to Klau's paper [7] and [13] for the precise details. We present the pseudo-code in Listing 1. The essence of the method is as follows. A simple way to compute an upper bound on the objective of the network alignment problem is to ignore the matching constraints and compute $\alpha\mathbf{e}^T\mathbf{w} + \frac{\beta}{2}\mathbf{e}^T\mathbf{S}\mathbf{e}$. This bound, predictably, is terrible. The problem is that $\mathbf{e}$ is rather far from a matching. Klau improves it by decomposing the objective as $\mathbf{x}^T(\alpha\mathbf{w} + \frac{\beta}{2}\mathbf{S}\mathbf{x})$ where $\mathbf{x}$ is a matching. To get an upper bound on $\frac{\beta}{2}\mathbf{S}\mathbf{x}$, we treat the elements of each row of $\mathbf{S}$ as weights on the edges of the bipartite graph L and we pick an optimal matching in L for each row (Step 1). We store the weight in the corresponding element of the vector $\mathbf{d}$. We then choose a heaviest matching of $\alpha\mathbf{w} + \mathbf{d}$ (Step 3), which is an upper bound on $\mathbf{x}^T(\alpha\mathbf{w}+\frac{\beta}{2}\mathbf{S}\mathbf{x})$. The problem with this approach is that the matchings picked for each row of $\mathbf{S}$ (whose indicators are stored in the matrix $\mathbf{S_L}$) and the final matching of all rows $\mathbf{x}^{(k)}$ may not agree. The fix for this makes the algorithm iterative. Klau's method uses a sub-gradient type approach to update a weight matrix $\mathbf{U}^{(k)}$ (actually, a matrix of Lagrange multipliers) in order to force the matchings picked in each row to agree (Step 5). The parameter γ is like a step-size parameter on these updates. At each step of this method, we generate an upper-bound on objective function, and a lower-bound from the current best objective (Step 4). Thus, this method can actually detect when it has reached the optimal point, although that will not always occur. In the subgradient method, we check to see if we have found a better upper-bound recently (within `mstep` iterations), and if not, decrease the step-size γ by a factor of 2.

Listing 1. Klau's iterative procedure for network alignment.

```
$\mathbf{U}^{(0)} = 0, \gamma$ is given, mstep is given
**for** $k = 1$ to n$_{\text{iter}}$
  **for** each row $i$ in $\mathbf{S}$,    *Step 1: row match*
    **set** row $i$ of $\mathbf{S_L}$
      $=$ bipartite_match$(\mathbf{e}_i^T(\frac{\beta}{2}\mathbf{S}+\mathbf{U}^{(k)}-{\mathbf{U}^{(k)}}^T))$
    and **set** $d_i$ to be the value of the matching
  $\bar{\mathbf{w}}^{(k)} = \alpha\mathbf{w} + \mathbf{d}$    *Step 2: daxpy*
  $\mathbf{x}^{(k)}$ = bipartite_match$(\bar{\mathbf{w}}^{(k)})$    *Step 3: match*
  obj$^{(k)} = \alpha{\mathbf{x}^{(k)}}^T\mathbf{w} + \frac{\beta}{2}{\mathbf{x}^{(k)}}^T\mathbf{S}\mathbf{x}^{(k)}$    *Step 4: objective*
  upper$^{(k)} = {\bar{\mathbf{w}}^{(k)}}^T\mathbf{x}^{(k)}$
  $\mathbf{X}^{(k)} = \text{diag}(\mathbf{x}^{(k)})$    *Step 5: update U*
  $\mathbf{F} = \mathbf{U}^{(k-1)} - \gamma\mathbf{X}^{(k)}\textbf{triu}(\mathbf{S_L}) + \gamma\textbf{tril}(\mathbf{S_L})^T\mathbf{X}^{(k)}$
  $\mathbf{U}^{(k)} = \underset{-0.5,0.5}{\text{bound}}\mathbf{F}$
  **if** upper$^{(k)}$ has not changed in mstep iterations,
    **set** $\gamma = \gamma/2$
  **end**
**end**
**return** $\mathbf{x}^{(k)}$ with the largest value of obj$^{(k)}$
```

B. Belief Propagation for Network Alignment

Another heuristic approach for network alignment is to use a message passing procedure known as belief propagation [13], [14], [18]. In order to do so, we treat maximizing the network alignment quadratic program (NAQP) as finding a maximum a posteriori estimate of the following probability distribution:

$$\mathbb{P}(\mathbf{x}) = \tfrac{1}{Z}\exp(\alpha\mathbf{x}^T\mathbf{w} + \tfrac{\beta}{2}\mathbf{x}^T\mathbf{S}\mathbf{x})\,\text{Ind}[\mathbf{C}\mathbf{x} \leq \mathbf{e}],$$

where Z is an unknown normalization constant. Belief propagation is a standard approximation for this task when the probability distribution can be written as a product of exponentials (as it can in this case). This requires introducing a factor graph form of the problem and we omit that in the interest of space. Its use here was also inspired by the success of a belief propagation method for solving a maximum weight matching problem in a distributed setting [19].

A full derivation of the algorithm, along with many simplifications to the classical belief propagation procedure, is presented by Bayati, Gleich et al. [13], [14]. We only provide a short, intuitive description here. The algorithm works by iteratively updating two weight vectors $\mathbf{y}^{(k)}, \mathbf{z}^{(k)}$ and a weight matrix $\mathbf{S}^{(k)}$. The first represents the log-likelihood of each edge in L occurring in the final matching given that we want each vertex in graph A matched to at most one vertex in graph B. The second represents the same thing, but given that we want each vertex in graph B to match to at most one vertex in graph A. The final vector represents the log-likelihood of overlapped edges appearing in the solution. These weight vectors correspond to edges in L (1st and 2nd) and non-zeros in $\mathbf{S}$ (3rd). The update rules encode the logic of a local, greedy agent that attempts to determine its own likelihood, given the likelihoods in its neighborhood. Because of this

Listing 2. A belief-propagation message passing procedure for network alignment. See the text for a description of othermax and round_heuristic.

```
$\mathbf{y}^{(0)} = 0, \mathbf{z}^{(0)} = 0, \mathbf{d}^{(0)} = 0, \mathbf{S}^{(k)} = 0$
for $k = 1$ to $n_{\text{iter}}$
    $\mathbf{F} = \text{bound}_{0,\beta}[\beta\mathbf{S} + \mathbf{S}^{(k)^T}]$   Step 1: compute F
    $\mathbf{d} = \alpha\mathbf{w} + \mathbf{Fe}$   Step 2: compute d
    $\mathbf{y}^{(k)} = \mathbf{d} - \text{othermaxcol}(\mathbf{z}^{(k-1)})$   Step 3: othermax
    $\mathbf{z}^{(k)} = \mathbf{d} - \text{othermaxrow}(\mathbf{y}^{(k-1)})$
    $\mathbf{S}^{(k)} = \text{diag}(\mathbf{y}^{(k)} + \mathbf{z}^{(k)} - \mathbf{d})\mathbf{S} - \mathbf{F}$   Step 4: update S
    $(\mathbf{y}^{(k)}, \mathbf{z}^{(k)}, \mathbf{S}^{(k)}) \leftarrow \gamma^k(\mathbf{y}^{(k)}, \mathbf{z}^{(k)}, \mathbf{S}^{(k)}) +$
      $(1 - \gamma^k)(\mathbf{y}^{(k-1)}, \mathbf{z}^{(k-1)}, \mathbf{S}^{(k-1)})$   Step 5: damping
    round_heuristic $(\mathbf{y}^{(k)})$   Step 6: matching
    round_heuristic $(\mathbf{z}^{(k)})$
end
return $\mathbf{y}^{(k)}$ or $\mathbf{z}^{(k)}$ with the largest objective value
```

interpretation, the weight vectors are usually called messages as they communicate the "beliefs" of each "agent." In this particular problem, the neighborhood of an agent represents all of the other edges in graph L incident on the same vertex in graph A (1st vector), all edges in L incident on the same vertex in graph B (2nd vector), or the edges in L that are part of an overlap. The message vectors do not generally converge, and thus, the iteration is artificially damped to enforce convergence. We only describe one type of damping. See [13] for other variations.

After each update to the messages, we round the messages to a matching using a bipartite maximum weight matching procedure, and then evaluate the objective function.

We present a pseudo-code for the method in Figure 2. This code uses the mildly curious function *othermaxrow*. Suppose that $\mathbf{g}$ is a weight vector on the edges of a bipartite graph L. This means we can index $\mathbf{g}$ with the edges of L such that $g_{i,i'}$ is the weight on the edge $(i, i') \in E_L$. The othermaxrow function then computes a new weight for each edge in L:

$$[\text{othermaxrow}(\mathbf{g})]_{i,i'} = \underset{0,\infty}{\text{bound}}[\max_{(i,k') \in E_L, k' \neq i'} g_{i,k'}].$$

This function computes something rather simple. Given a row, replace all non-zeros in that row with the maximum value for the row; except, for the element that *is* the maximum value, replace it with the second largest value. The othermaxcol function works on columns instead of rows.

C. Stopping Criteria

Both algorithms generate a sequence of heuristic weight vectors whose solution quality varies continually. There is no monotonicity in the solution quality, which can vary greatly between iterations. Thus, no simple stopping criteria is possible. Due to the shrinking step length in Klau's method and the artificial damping in BP, there is also no point in running for more than 500-1000 iterations with reasonable choices of these two parameters.

D. Complexity

The complexity of each iteration of Klau's method and the BP method is $O(\text{nnz}(S) + |E_L| + \text{matching})$, where $O(\text{matching})$ is the complexity of the bipartite matching in step 5. Let $N = (|V_A| + |V_B|)$. Currently, the best known algorithm for computing an optimal edge-weighted matching has complexity $O(|E_L|N + N^2 \log N)$ [20]. Practical implementations have complexity $O(|E_L|N \log N)$ [21]. The half-approximate matching discussed below has complexity $O(|E_L|)$. Thus, when we replace the exact matching step with approximate matching for our experiments, the complexity of each iteration will be $O(\text{nnz}(S) + |E_L|)$.

IV. Parallel Network Alignment Implementations

We now consider a shared-memory multi-core implementation of these procedure with OpenMP. All required memory is pre-allocated before the first iteration and there are no dynamic memory allocations. We avoid computing intermediate results whenever possible, see the online codes for details.

A. Matrix computations in both methods

All matrices are sparse and are stored as compressed sparse row arrays. All non-zero patterns and structures remain fixed throughout iterations. We found using simple OpenMP "parallel for" loops faster than using a matrix library such as Intel's Math Kernel Library. This result is due to the simplicity of the matrix computations. For instance, because $\mathbf{S}$ and $\mathbf{U}$ are structurally symmetric with the same structure, the transposes have the same row pointer and the column index arrays. But the value array is permuted. So we compute the permutation and whenever we need to transpose one of these matrices, we just permute the values array according to the permutation. Since these matrices do not change structure during the algorithm, we can compute the permutation once. Sometimes – such as line 5 of Klau's method or line 3 of BP – we simply use the permutation array to pull elements from appropriate memory locations without any intermediate write.

The matrix $\mathbf{S}$ can be highly imbalanced (some rows are empty and others have many non-zeros) and so we found that using a dynamic schedule in OpenMP's "parallel for" construction yielded better performance than a static schedule. After some experimentation, we found that a chunk-size of 1000 seemed to produce the best performance for these operations. Indeed, we found this observation to be the case for all operations involving the matrix $\mathbf{S}$. Synchronization only occurs at the end of each "parallel for" loop.

B. Specifics about Klau's method

In the first step of the iteration, we need to solve a bipartite matching problem for each row of the matrix $\mathbf{S}$ with weights that change based on $\mathbf{U}^{(k)}$. We compute $\frac{\beta}{2}\mathbf{S} + \mathbf{U}^{(k)} - \mathbf{U}^{(k)^T}$ using the permutation trick, and then we parallelize the operation over rows. Each of these matching problems is small because there are only a few non-zeros in each row of $\mathbf{S}$ and so we do not consider using the parallel approximation here. We precompute the maximum memory required for p threads to run matching problems on the rows of $\mathbf{S}$ and preallocate this memory outside of the iteration. The output of this procedure is a sparse matrix $\mathbf{S}_L$, which is conceptually an indicator vector for the non-zeros in $\mathbf{S}$ – we store this as an integer valued array for the compressed sparse row order of $\mathbf{S}$.

The remaining steps of the algorithm – except for the bipartite matching – can all be parallelized using standard parallel matrix constructs. In step 5, the functions $\mathbf{X}^{(k)}\mathbf{triu}(\mathbf{S}_L)$ and $\mathbf{tril}(\mathbf{S}_L)^T\mathbf{X}^{(k)}$ involve rescaling rows and columns of a portion of $\mathbf{S}$ and there is no need to form the diagonal matrix $\mathbf{X}^{(k)}$. Again, we use the same permutation trick for the transpose.

C. Specifics about the BP method

The othermax computations in Step 3 are parallelized over columns and rows, respectively. We again found that using a dynamic schedule with a chunk-size of 1000 produced the best performance here. The update to the non-zeros in $\mathbf{S}^{(k)}$ in step 4 corresponds to rescaling the matrix $\mathbf{S}$ by rows, and then subtracting the values from $\mathbf{F}$.

There is an additional parallelization opportunity for this procedure. Unlike Klau's method, moving from the kth to $(k+1)$th iteration does not depend on solving a bipartite matching problem. Instead, we need to solve the matching problem because the quality of the iterates $\mathbf{y}$ and $\mathbf{z}$ changes as the iteration progresses. Solutions at the final iterations are often inferior to those at intermediate iterations. This flexibility enables us to store the last few iterates $\mathbf{y}^{(k)}, \mathbf{z}^{(k)}, \ldots, \mathbf{y}^{(k-r+1)}, \mathbf{z}^{(k-r+1)}$, and then evaluate their quality simultaneously. We found that this batched rounding procedure markedly improved the scalability of the code. We refer to the parameter r as the batch size, which refers to the number of iterates we store before rounding. We call the resulting method BP(batch=r), e.g. BP(batch=1) is the method where we immediately round. The use of r for the batch size is slightly different from storing the history for r iterations as previously described, although they only differ by a factor of two. Given a batch size of r, we store r sets of weights. After $r/2$ iterations, we run r matching routines each as OpenMP "tasks". We then use OpenMP's nested parallelism within each task based on available threads. If the batch size is 4 and 4 threads are available, all matching problems will be solved simultaneously, and each problem is solved serially. But if 8 threads are available, each matching problem runs in parallel with 2 threads.

V. Parallel approximation algorithms for maximum weight bipartite matching

Algorithms for computing maximum weight matchings in bipartite and general graphs are either inherently serial, or have limited concurrency. Hence, we consider approximation algorithms to enhance concurrency while obtaining high quality matchings. We describe one such algorithm to compute half-approximate weight matchings. It also computes a *maximal* matching, which guarantees an approximation ratio of half for the cardinality as well. We call this algorithm the *locally-dominant* algorithm, since it repeatedly finds edges which are the heaviest in their local neighborhood in the graph.

The locally-dominant algorithm was first presented by Preis [22], and adapted by Manne and Bisseling [23] for parallel computers. Distributed-memory implementations of this algorithm were presented in [23] and [24]. Implementations suitable for multicores, manycores, and massively-multithreaded architectures were recently investigated by Halappanavar *et al.* [15]. We use their multicore implementation in this paper. The locally-dominant algorithm can compute matchings in general graphs. Therefore, we provide a bipartite graph as a general graph to the algorithm by not making a distinction between the two sets of vertices.

The details of the algorithm are presented in Algorithm ParallelMatch. For simplicity of presentation, we divide the algorithm into two phases.

Phase-1 of the algorithm (Lines 4 to 7) starts by calling the procedure FindMate for each vertex in parallel. This procedure scans the neighborhood of a vertex to identify the heaviest unmatched neighbor. If the neighbor list is maintained in a sorted order, this step can be done in constant time. Unique vertex ids are used to break ties consistently. Subsequently, locally-dominant edges are matched via a call to MatchVertex. This procedure checks if two vertices point to each other, in which case they are locally dominant. If true, the procedure will set corresponding entries in mate, and add the two end-points to the queue for processing their neighbors.

Updates to the queue are made using atomic memory operations. For efficiency we use the Intel x86 hardware intrinsic operator `__sync_fetch_and_add()` to increment the number of elements in the queue. This operator atomically increments the number and returns the original value, which can then be used to find a unique position in the queue by each thread to add a new element. To enable efficient reading and writing to queues, we use two independent queues Q_C and Q_N. While we process vertices matched in the previous iteration from Q_C, we add vertices matched in the current iteration to Q_N.

When every vertex has been processed once via calls to FindMate and MatchVertex, there will be some matched vertices queued in Q_C. Phase-2 begins at the end of this step and through a **while** loop (Line 9) iterates until Q_C becomes empty, at which point no more edges can be matched. In each iteration, we examine a matched vertex u, and the candidate mate of an unmatched neighbor v is reset to the next heaviest unmatched vertex in the adjacency list of v. Locally dominant edges are matched again, and the newly matched vertices are added to Q_N to process their neighbors at the next iteration. The threads synchronize at the end of each iteration of this loop, and the contents of Q_N are assigned to Q_C via a pointer swap (Line 15). The queue Q_N is then cleared. Each vertex in the queue can be processed in parallel. Thus, the size of Q_C determines the amount of work that can be done in parallel. This size varies as the algorithm progresses. Based on the numbers presented in [15], the size decreases roughly by half after each iteration for different synthetic graphs. The parallel time complexity of our implementation is determined by the number of iterations of the **while** loop (expected to be $O(\log |V|)$ if the size decreases by a constant in each iteration).

Algorithm 1 Parallel $\frac{1}{2}$-Approx Matching. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a queue, Q_C, with vertices to be processed in the current step, and a queue, Q_N, with vertices to be processed in the next step – only matched vertices are queued; a vector candidate that maintains the id of current heaviest neighbor of a vertex.

```
1:  procedure ParallelMatch($G(V, E)$, mate)
2:      $Q_C \leftarrow \emptyset$;  $Q_N \leftarrow \emptyset$
3:      mate$\leftarrow \emptyset$
4:      for each $v \in V$ in parallel do                     ▷ Phase-1
5:          candidate[$v$] $\leftarrow$ FindMate($v$)
6:      for each $v \in V$ in parallel do
7:          MatchVertex($v$, $Q_C$)
8:
9:      while $Q_C \neq \emptyset$ do                          ▷ Phase-2
10:         for each $u \in Q_C$ in parallel do
11:             for each $v \in adj(u)$ do
12:                 if candidate[$v$] = $u$ then
13:                     candidate[$v$] $\leftarrow$ FindMate($v$, $Q_N$)
14:                     MatchVertex($v$, $Q_N$)
15:         $Q_C \leftarrow Q_N$
16:         $Q_N \leftarrow \emptyset$
```

Algorithm 2 Find and return the current-heaviest candidate for a vertex.

```
1:  procedure FindMate($s$)
2:      $max\_wt \leftarrow -\infty$
3:      $max\_wt\_id \leftarrow \emptyset$
4:      for each $t \in adj(s)$ do
5:          if (mate[$t$] = $\emptyset$) AND ($max\_wt < w(e_{s,t})$) then
6:                                              ▷ Use vertex id to break ties
7:              $max\_wt \leftarrow w(e_{s,t})$
8:              $max\_wt\_id \leftarrow t$
9:      return ($max\_wt\_id$)
```

Algorithm 3 Match an edge if locally dominant. Add the two end-points to Q.

```
1:  procedure MatchVertex($s$, $Q$)
2:      if candidate[candidate[$s$]] = $s$ then
3:                                              ▷ Found a locally-dominant edge
4:          mate[$s$] $\leftarrow$ candidate[$s$]
5:          mate[candidate[$s$]] $\leftarrow s$
6:          $Q \leftarrow Q \cup \{s, \text{candidate}[s]\}$
7:                  ▷ Add both of the end-points; atomic updates to $Q$
```

The performance of matching algorithms critically depends on the initialization [25], [26]. The approximate weighted matching algorithm we are using was developed for non-bipartite graphs, and we have not adapted it to exploit the bipartiteness of the graph L. Hence the approximate matching algorithm spawns threads from both vertex sets V_A and V_B in order to match vertices. We experimented with an initialization algorithm tailored for bipartite graphs by spawning threads only from one of the vertex sets V_A or V_B to identify locally dominant edges. If the thread is responsible for matching a vertex in V_A, then it has to check the adjacency sets of the vertices in V_B that are adjacent to it in order to determine if the edge is locally dominant. The vertices in V_A (or V_B) can be matched in parallel by using `__sync_fetch_and_add()` operations to avoid conflicts. We found that this initialization noticably improved the speed of the algorithm. In future work, we will investigate approximation algorithms for weighted matching in bipartite graphs.

TABLE II
FOR EACH PROBLEM IN OUR BIOINFORMATICS AND ONTOLOGY SETS, WE REPORT THE NUMBER OF VERTICES IN GRAPH A AND B, THE NUMBER OF EDGES IN THE GRAPH L, AND THE NUMBER OF NONZEROS IN $\mathbf{S}$.

Problem	$\lvert V_A\rvert$	$\lvert V_B\rvert$	$\lvert E_L\rvert$	$\lvert S\rvert$
dmela-scere	9,459	5,696	34,582	6,860
homo-musm	3,247	9,695	15,810	12,180
lcsh-wiki	297,266	205,948	4,971,629	1,785,310
lcsh-rameau	154,974	342,684	20,883,500	4,929,272

VI. SAMPLE PROBLEMS

In order to evaluate the solution quality and parallel performance of network alignment with approximate matching, we consider three types of problems: synthetic, protein-protein network alignment, and ontology alignment. See Table II for summary statistics about the bioinformatics and ontology alignment problems. For all problems, the degree distribution in L is fairly regular, whereas the non-zero distribution in $\mathbf{S}$ is highly irregular and imbalanced.

A. Synthetic power-law problems

We use synthetic problems similar to those described in [13]. Each problem instance is small, and it is primarily used to evaluate the solution quality of the network alignment heuristics. We first generate an initial graph G and then randomly add edges with probability 0.02 to form the graphs A and B. We let G be a 400 node random power-law graph to approximate the structure of most modern information networks [27]. To produce G, we first sampled a power-law degree distribution and then generated a random graph with that prescribed degree distribution. Because we started with the same underlying graph G, we known the identity matching between the graphs and utilize this matching as a reference point. To generate L, we start with the identity matching and then randomly sample all possible edges in L with probability p in order to globally diversify the matches. We find it more meaningful to describe the probability p in terms of the expected degree in the graph: $\bar{d} = p \cdot |V_A|$.

B. Bioinformatics

Aligning between protein-protein interaction networks from different species is an important problem in bioinformatics [5]. We use a problem from Singh et al. [5] (dmela-scere) and a problem from Klau [7] (homo-musm). The first is an alignment between protein interactions in a fly (*D. melanogaster*) and yeast (*S. cerevisiae*). The second is an alignment between humans (*H. sapiens*) and mice (*M. musculus*). We utilize these problems solely for the instances of a network alignment problem and do not focus on the biological insights suggested. The graph L and associated weights are from the original papers.

C. Ontology alignment

We consider two problems in ontology alignment from [13]. The first is an alignment between the Library of Congress subject headings and Wikipedia categories (lcsh-wiki). While both ontologies have a core hierarchical tree, they also have many cross edges for other types of relationships. Thus we can think of them as general graphs. The second problem is an alignment between the Library of Congress subject headings and its counterpart in the French National Library: Rameau. In both cases, the edges and weights in L are computed via a text-matching of the subject heading strings (and via translated strings in the case of Rameau). These problems are larger than the bioinformatics ones.

VII. Network alignment with approximate matching

In this section we address the question: how does the behavior of Klau's method and the BP method change when we substitute the approximate matching procedure from Section V for the bipartite matching step in each algorithm? Note that we always use exact matching in the first step of Klau's method (Step 1: row match) because the problems in each row tend to be small and we parallelize over rows. Note also that the bipartite matching is much more integral to Klau's method than the BP procedure. For the BP procedure, we only solve a bipartite matching problem to *evaluate* the quality of an iterate, whereas in Klau's method, the results of the matching determine the update to the Lagrange multipliers $\mathbf{U}$. Put another way, the set of iterates from the BP method is independent of the choice of matching algorithm. At the end of the iteration, each of the methods returns the best heuristic it computed, and we perform one final step of exact maximum weight matching to convert this into the returned matching.

We begin by evaluating the solution quality on synthetic power-law problems. We use $\alpha = 1, \beta = 2$ and 1000 iterations of each method. We evaluate each solution by comparison with the identity alignment. Note that the identity alignment – which assigns each vertex in graph A to its mirror image in graph B based on the original graph G – may not be the optimal alignment because the perturbations to the graph could introduce a better solution. This seems to occur because we compute objective values larger than the identity alignment for $\bar{d} > 10$. We also study how many of the correct matches each method generates with respect to the identity matching. The results are shown in Figure 2. In the top plot, we show the fraction of the objective from the identity matching achieved (y-axis) as the expected degree $\bar{d}$ of random edges in L

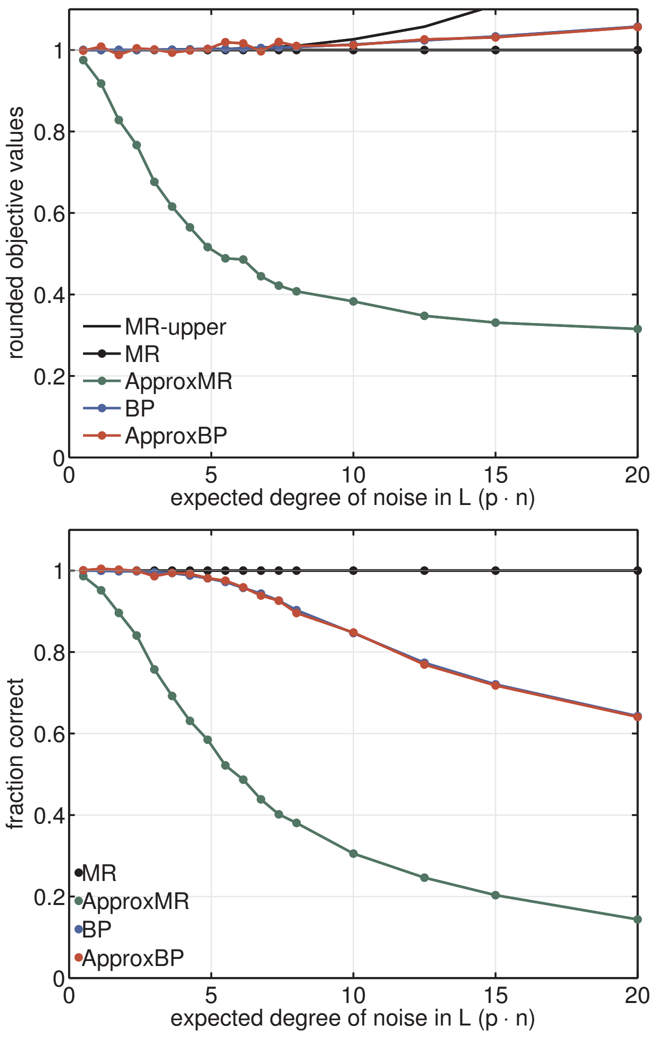
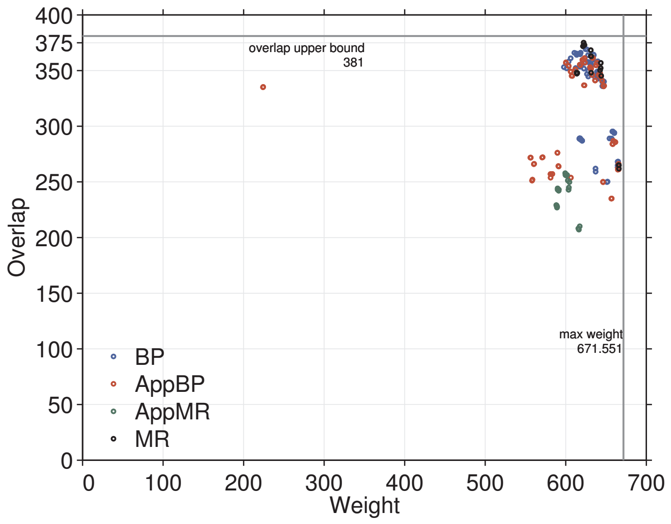


Fig. 2. Alignment with a power-law graph shows the large effect that approximate rounding can have on solutions from Klau's method (MR). With that method, using exact rounding will yield the identity matching for all problems (bottom figure), whereas using the approximation results in over a 50% error rate. The results from the BP method with and without approximate matching are indistinguishable. Small differences were randomly added to show both lines in the figure.

varies from 2 to 20 (x-axis). In the bottom plot, we show the fraction of correct matches (y-axis), again as the expected degree $\bar{d}$ varies. Problems with more random edges are more challenging. The figures demonstrate that Klau's method is sensitive to using an approximate matching routine, whereas the BP method with exact and approximate matching are nearly indistinguishable.

We also evaluate the how the matching weight ($\mathbf{w}^T\mathbf{x}$, plotted on the x-axis) and overlap ($\mathbf{x}^T\mathbf{S}\mathbf{x}/2$, plotted on the y-axis) change for a bioinformatics problem (dmela-scere) and an ontology problem (lcsh-wiki) in the upper and lower plots in Figure 3. Again, the BP results with and without approximate matching are virtually indistinguishable. Klau's method, however, produces results that are considerably worse.



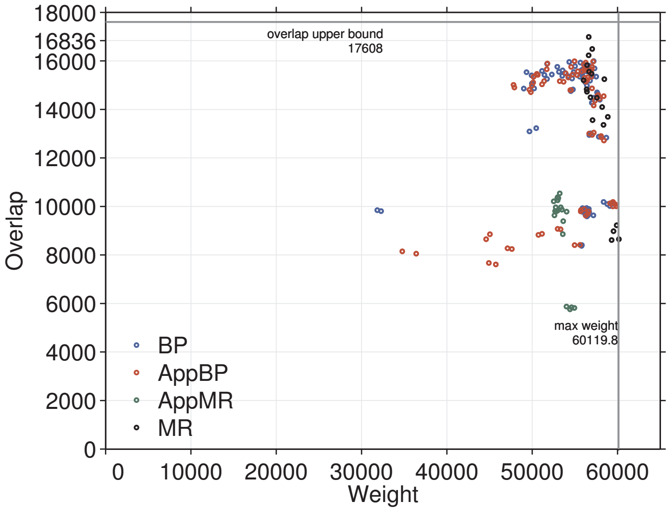


Fig. 3. (top) Results from the dmela-scere problem; (bottom) results from the lcsh-wiki problem. Each point represents the matching weight and overlap scores of a solution method for a wide variety of objective functions α, β, damping parameters and other input parameters (see [13] for the details). The goal is to evaluate the range of solutions produced by the methods with and without approximate matching. These show a relatively small change in solution quality for Klau's MR method and almost no change in the solution quality for the BP method.

VIII. Parallel scaling

In the previous section, we saw that the result quality using approximate matching in the BP method was almost indistinguishable from the results using an exact bipartite matching. This property suggests that using the parallel approximate matching will improve the runtime performance without altering its utility. Thus, in this section we address the question: how well does an OpenMP parallel implementation of network alignment scale? We further refine this question as: how does the layout of memory and computational threads impact scalability of the method (Section VIII-B)? Even though Klau's method did not perform as well with the approximation, we include it for comparison. We also investigate the bottleneck steps in our implementation in Section VIII-C. Note that we do not include the time required for the final exact bipartite matching step in these experiments. We begin by describing the system used to evaluate the scaling. The software was compiled with Intel's C++ Compiler version 12.0.4.

A. *Hardware architecture*

We use an 8-processor Intel Xeon 2.40 Ghz CPU E7-8870 server for our experiments. Each processor has 10 cores, and each core can run two threads. Each processor also has 30 MB of L3 cache. The server has 128GB of memory in a non-uniform memory access architecture with 16GB per processor.

B. *Strong scaling with changes in thread and memory layout*

The NUMA architecture of our test machine implies that the layout of threads to processors and memory to processors could have a large effect on the scalability of our parallel OpenMP codes. We explore two possibilities for memory layout: bound (memory is allocated to the relevant socket) and interleaved (memory is allocated in a round robin fashion between sockets). Specifically, we used the "numactl" library with "–membind=all" vs. "–interleave=all" options to control these behaviors. We also explore two possibilities for thread layout: compact (assign threads on the same processor) vs. scattered (distribute the threads across processors). We used the `KMP_AFFINITY` flag to control this setting.

In the following plots, we show the speedups as the thread count varies for four methods: Klau's MR method and the BP algorithm with batch sizes 1, 10 and 20. Recall that we can accumulate a small number of message vectors to round simultaneously in the BP method. A batch size of 1 indicates that the method will round immediately whereas a batch size of 20 means that we will round up to 20 vectors simultaneously. All speedups were computed relative to the fastest run we computed with one thread, which always happened using memory bound to a single processor. The time required to solve these problems was large enough that small fluctuations due to non-deterministic thread scheduling do not have any significant impact. We only show results up to 80 threads as we saw no additional speedup beyond that point.

The scaling results for lcsh-wiki are shown in Figure 4. We ran each code for 400 iterations using $\alpha = 1, \beta = 2$, and $\gamma = 0.99$. For Klau's method, we set `mstep` = 10. In our experiences, performance did not vary with these parameters. Regardless of the method, the best scalability arises from using interleaved memory. When using 80 threads, it tends not to matter if the threads are compactly assigned or scattered over processors, but scattering them shows additional speedup for fewer threads. Klau's MR method scales to 40 threads and yields a speedup of about 15-fold. The BP method scales to 40 threads with a 15-fold speedup. Increasing the batch-size does not yield any additional scaling, even though rounding takes approximate 50% to 75% of the total iteration time, see Figure 7. In Figure 5, we also show that we get approximately the same scaling behavior on the larger lcsh-rameau problem. In this case, using a batch size of 20 gave the best speedup.

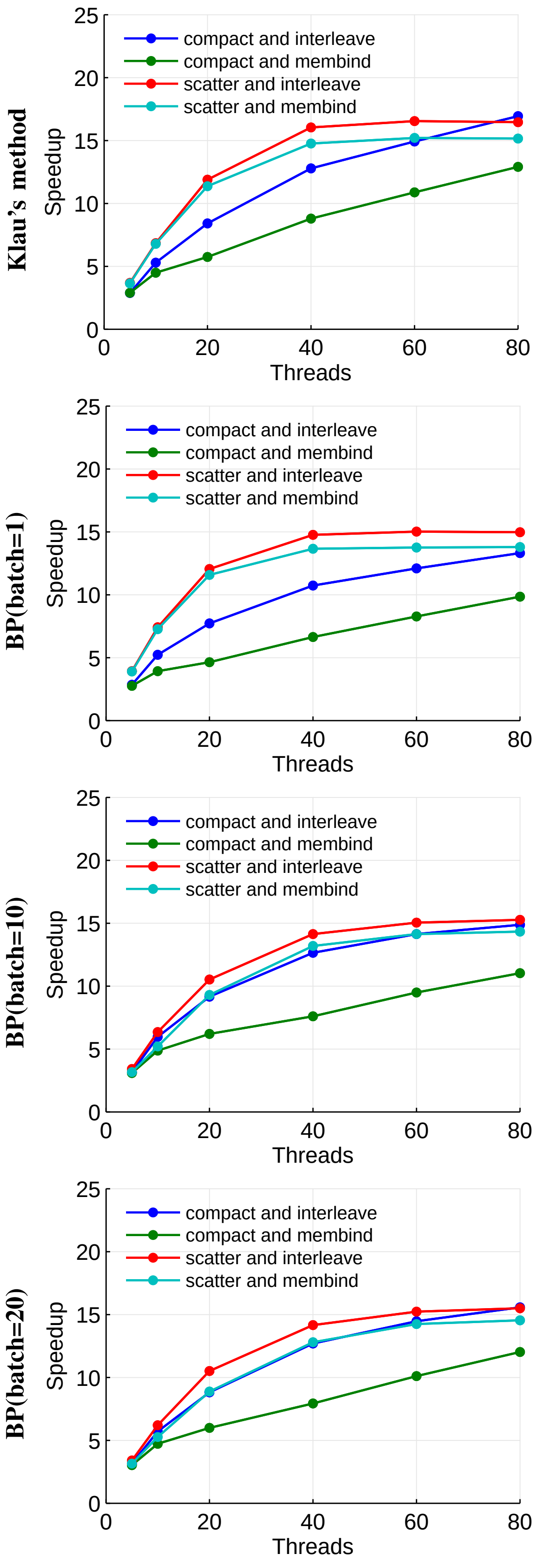


Fig. 4. Strong scaling results for the lcsh-wiki problem for the four methods: Klau's and BP with batch sizes 1, 10, and 20. See the text for a discussion.

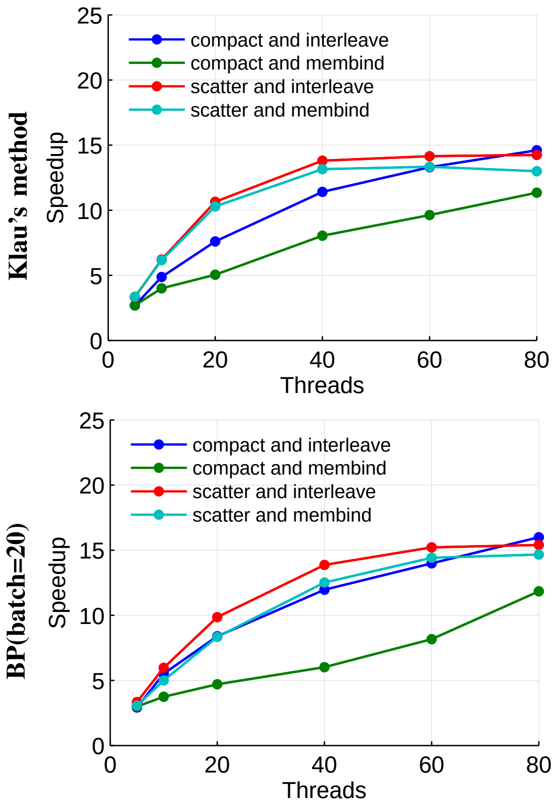


Fig. 5. Strong scaling results for the lcsh-rameau problem for Klau's method and BP with batch size 20. See the text for a discussion.

The scaling results for the two bioinformatics problems do not show any scaling beyond 10 threads, which is a single socket. This finding is expected given the small size of those problems would fit into the level 3 cache on the processor. To conserve space, we omit these results.

C. Scalability bottlenecks

In the last section we saw that the Klau's MR method and the BP method scale to roughly 40 threads with speedups around 15-fold. In this section, we investigate why the methods stop scaling. For each of the 5 steps identified in the pseudo-code for Klau's method, and for each of the 6 steps identified in the pseudo-code for the BP method, we show the strong scaling results. The results for lcsh-wiki using Klau's method and BP with a batch size of 20 are in Figure 6. The limiting step differs in each setting. The bipartite matching step seems to limit further scaling of the MR method. We are limited by the iterative portion of the approximate matching procedure. Whereas for the BP method, the damping step appears to be the limiting point on the lcsh-wiki problem. With a batch size of 20, we need to store and access the last 20 iterates, which stresses the memory bandwidth. If we add additional threads, then we use OpenMP nested parallelism. This mode does not consider memory layout when assigning threads, which causes many remote memory accesses and limits scalability.

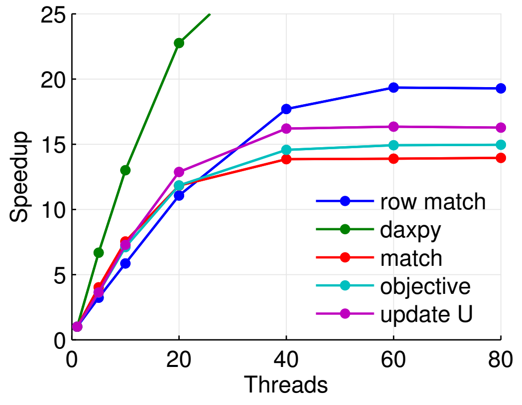


Fig. 6. Strong scaling of iteration steps for Klau's method on lcsh-wiki. For 40 threads, the row match step took 40% of the runtime and the matching took 40% of the runtime. Consequently, the matching limits the overall scalability.

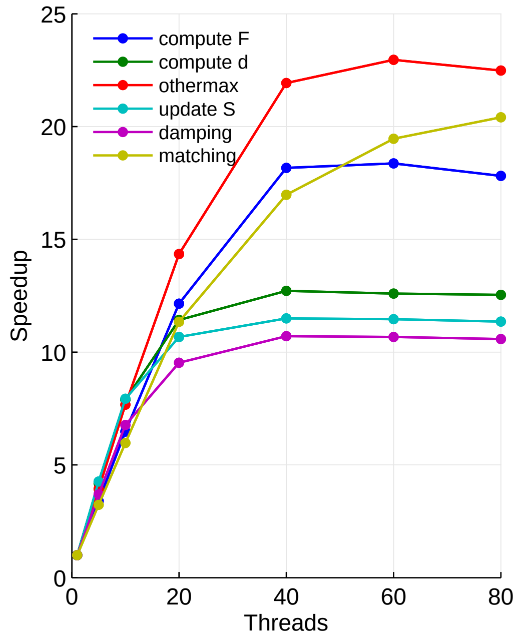


Fig. 7. Strong scaling of iteration steps for BP(batch=20) on lcsh-wiki. For 40 threads, the othermax step took 15% of the runtime and matching took 58% of the runtime. The damping step took 12% of the runtime and seems to be the limiting step.

IX. Discussion

Let us recap. We study the shared-memory, multi-threaded parallelization of two algorithms for the network alignment problem: Klau's method and a BP method. The key to the parallelization is to substitute a parallel, approximate matching routine for an exact bipartite matching routine. As we saw in Section VII, this substitution preserved the quality of the solutions generated the BP method on real world problems.

We were able to attain a 15-fold speedup using 40 cores for the lcsh-wiki problem with the BP method, which is a reasonably good strong scaling result for a complex blend of matrix computations and matching procedures. Practically, this speedup translates into finding a good solution in 36 seconds instead of 10 minutes. Indeed, at this speed, the methods are suitable for use in a computational steering or visual analytics setting where we can engage a human to evaluate solutions from the algorithm and adjust inputs accordingly. This setting is important because the network alignment problem is only an approximation for most users' true goal: find a matching that is "correct" and "useful." In this case, given the result of a network alignment problem, users may want to fix certain problematic alignments by removing potential matches from L and recompute.

We have also included an initial study of where the scalability falters for Klau's method and the BP method. These results indicate we could expect better scaling with improvement of a few steps of the method. Additionally, the structure of the BP method offers other avenues for parallelization. First, we could reorganize the computation of the BP messages in terms of independent tasks, e.g. the othermax functions could be computed independently. Some initial results we obtained using this idea are promising. Second, the algorithms could also be implemented in a distributed setting using primitives from the Combinatorial BLAS library [28] for the matrix computations and a distributed half-approximation matching algorithm [29]. Third, we could implement the methods using MPI and OpenMP for better control over the memory layout.

Acknowledgment

Our thanks to the Intel Corporation for providing the 8-processor Intel E7-8870 server as a gift to the CS department at Purdue for research on multi-core algorithms. This research was supported by the DOE CSCAPES Institute grant (DE-FC02-08ER25864), NSF CAREER grant 1149756-CCF, and the Center for Adaptive Super Computing Software— Multithreaded Architectures (CASS-MT) at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] D. Conte, P. P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 3, pp. 265–298, May 2004.

[2] W. Hu, N. Jian, Y. Qu, and Y. Wang, "GMO: A graph matching for ontologies," in *Proceedings of K-CAP Workshop on Integrating Ontologies*, 2005, pp. 41–48. [Online]. Available: ftp1.de.freebsd.org/Publications/CEUR-WS/Vol-156/paper7.pdf

[3] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceedings of the 18th International Conference on Data Engineering*. San Jose, CA: IEEE Computer Society, 2002, p. 117.

[4] B. P. Kelley, R. Sharan, R. M. Karp, T. Sittler, D. E. Root, B. R. Stockwell, and T. Ideker, "Conserved pathways within bacteria and yeast as revealed by global protein network alignment," *PNAS*, vol. 100, no. 20, pp. 11 394–11 399, September 2003. [Online]. Available: http://www.pnas.org/content/100/20/11394.abstract

[5] R. Singh, J. Xu, and B. Berger, "Pairwise global alignment of protein interaction networks by matching neighborhood topology," in *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, ser. Lecture Notes in Computer Science, vol. 4453. Oakland, CA: Springer Berlin / Heidelberg, 2007, pp. 16–31.

[6] ——, "Global alignment of multiple protein interaction networks with application to functional orthology detection," *Proceedings of the National Academy of Sciences*, vol. 105, no. 35, pp. 12 763–12 768, September 2008.

[7] G. Klau, "A new graph-based method for pairwise global network alignment," *BMC Bioinformatics*, vol. 10, no. Suppl 1, p. S59, January 2009.

[8] J. Flannick, A. Novak, B. S. Srinivasan, H. H. McAdams, and S. Batzoglou, "Græmlin: General and robust alignment of multiple large interaction networks," *Genome Research*, vol. 16, pp. 1169–1181, August 2006. [Online]. Available: http://ai.stanford.edu/~serafim/Publications/2006_Graemlin.pdf

[9] J. Flannick, A. Novak, C. B. Do, B. S. Srinivasan, and S. Batzoglou, "Automatic parameter learning for multiple network alignment," in *Proceedings of the 12th Annual International Conference on Computational Molecular Biology (RECOMB2008)*, 2008, pp. 214–231. [Online]. Available: http://ai.stanford.edu/~serafim/Publications/Graemlin2.0_submitted.pdf

[10] O. Kuchaiev, T. Milenkovic, V. Memisevic, W. Hayes, and N. Przulj, "Topological network alignment uncovers biological function and phylogeny," *arXiv*, vol. 0810.3280, no. Version 3, p. Online, 2009. [Online]. Available: http://arxiv.org/abs/0810.3280

[11] G. Kollias, S. Mohammadi, and A. Grama, "Network similarity decomposition (nsd): A fast and scalable approach to network alignment," *Knowledge and Data Engineering, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011.

[12] N. Atias and R. Sharan, "Comparative analysis of protein networks: hard problems, practical solutions," *Commun. ACM*, vol. 55, no. 5, pp. 88–97, May 2012.

[13] M. Bayati, D. F. Gleich, A. Saberi, and Y. Wang, "Message passing algorithms for sparse network alignment," Accepted, Transactions on Knowledge Discovery from Data, 2012. [Online]. Available: http://arxiv.org/abs/0907.3338

[14] M. Bayati, M. Gerritsen, D. F. Gleich, A. Saberi, and Y. Wang, "Algorithms for large, sparse network alignment problems," in *Proceedings of the 9th IEEE International Conference on Data Mining*, December 2009, pp. 705–710.

[15] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, "Approximate weighted matching on emerging manycore and multithreaded architectures," *International Journal of High Performance Computing Applications*, p. Accepted, 2012.

[16] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*, 2nd ed. SIAM, 2012.

[17] E. L. Lawler, "The quadratic assignment problem," *Management Science*, vol. 9, no. 4, pp. 586–599, July 1963. [Online]. Available: http://www.jstor.org/stable/2627364

[18] S. Bradde, A. Braunstein, H. Mahmoudi, F. Tria, M. Weigt, and R. Zecchina, "Aligning graphs and finding substructures by a cavity approach," *EPL (Europhysics Letters)*, vol. 89, no. 3, p. 37009, 2010.

[19] M. Bayati, D. Shah, and M. Sharma, "Max-product for maximum weight matching: Convergence, correctness, and LP duality," *Information Theory, IEEE Transactions on*, vol. 54, no. 3, pp. 1241–1251, March 2008.

[20] H. N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 434–443. [Online]. Available: http://dl.acm.org/citation.cfm?id=320176.320229

[21] K. Mehlhorn and G. Schäfer, "Implementation of $O(nm \log n)$ weighted matchings in general graphs: the power of data structures," *J. Exp. Algorithmics*, vol. 7, p. 4, December 2002. [Online]. Available: 10.1145/944618.944622

[22] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs," in *STACS 99*, ser. Lecture Notes in Computer Science, C. Meinel and S. Tison, Eds., vol. 1563. Springer Berlin / Heidelberg, 1999, pp. 259–269.

[23] F. Manne and R. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 4967. Springer Berlin / Heidelberg, 2008, pp. 708–717.

[24] M. Halappanavar, "Algorithms for vertex-weighted matching in graphs," Ph.D. dissertation, Old Dominion University, Norfolk, VA, 2009.

[25] J. Langguth, F. Manne, and P. Sanders, "Heuristic initialization for bipartite matching problems," *ACM Journal of Experimental Algorithmics*, vol. 15, 2010.

[26] K. Kaya, J. Langguth, F. Manne, and B. Uçar, "Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs," CERFACS, Toulouse, France, Technical Report TR/PA/11/33, 2011. [Online]. Available: http://www.cerfacs.fr/algor/reports/2011/TR_PA_11_33.pdf

[27] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[28] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, November 2011.

[29] U. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 1971 –1980.