# Designing Scalable $b$-MATCHING Algorithms on Distributed Memory Multiprocessors by Approximation

Arif Khan*, Alex Pothen*, Md. Mostofa Ali Patwary†,
Mahantesh Halappanavar‡, Nadathur Rajagopalan Satish†, Narayanan Sundaram† and Pradeep Dubey†
*Computer Science, Purdue University †Intel Labs ‡ Pacific Northwest National Lab
Email: *{khan58, apothen}@purdue.edu
†{mostofa.ali.patwary, nadathur.rajagopalan.satish, narayana.sundaram, pradeep.dubey}@intel.com
‡hala@pnnl.gov

***Abstract*—A $b$-MATCHING is a subset of edges $M$ such that at most $b(v)$ edges in $M$ are incident on each vertex $v$, where $b(v)$ is specified. We present a distributed-memory parallel algorithm, $b$-SUITOR, that computes a $b$-MATCHING with more than half the maximum weight in a graph with weights on the edges. The approximation algorithm is designed to have high concurrency and low time complexity. We organize the implementation of the algorithm in terms of asynchronous supersteps that combine computation and communication, and balance the computational work and frequency of communication to obtain high performance. Since the performance of the $b$-SUITOR algorithm is strongly influenced by communication, we present several strategies to reduce the communication volume. We implement the algorithm using a hybrid strategy where inter-node communication uses MPI and intra-node computation is done with OpenMP threads. We demonstrate strong and weak scaling of $b$-SUITOR up to $16K$ processors on two supercomputers at NERSC. We compute a $b$-MATCHING in a graph with $2$ billion edges in under $4$ seconds using $16K$ processors.**

## I. INTRODUCTION

For the problem of computing a maximum weighted $b$-MATCHING in a graph, we describe a distributed-memory parallel algorithm that scales to 16K cores of a multiprocessor. $b$-MATCHING is a generalization of the better known and studied MATCHING problem in graphs, and has applications to data privacy [1], semi-supervised learning and data clustering [2], finite element mesh refinement [3], preconditioning, etc. Our work on scalable algorithms for $b$-MATCHING is motivated by applications in data privacy and preconditioning.

We obtain a scalable parallel algorithm by careful algorithm design and choices in implementation. First, by employing approximation algorithms we avoid the polynomial, but still impractical, time complexity of an algorithm for computing the maximum weight, and obtain algorithms with near-linear time complexity. These approximation algorithms nevertheless guarantee that the $b$-MATCHING computed has at least half the maximum weight, although in practice, the computed weight is closer to $95\%$ or more of the optimal weight. Second, new approximation algorithms are designed to have high concurrency, so that they can scale to ten-thousand cores or more. The increase in concurrency is achieved by removing ordering constraints from the approximation algorithm, albeit at the cost of additional work. We make choices in the algorithm to reduce the quantum of this extra work, so that the gains from concurrency are not lost. Third, we choose variants of the algorithm to implement that have proven to have low time complexity and good practical performance on serial and shared-memory computers. Fourth, we organize the distributed-memory parallel computation in terms of supersteps that include computation and communication, and balance the granularity of computation and the frequency of communication in order to obtain high performance. Fifth, by the choice of our parallel algorithm, we can make use of asynchronous supersteps, so that a processor can continue to compute after it receives a message from any processor that holds information about the neighbors of the vertices it owns. We can also make use of asynchronous communications to hide the communication latency. By such algorithmic choices, we decrease the communication and synchronization costs, and achieve good scaling.

The rest of this paper is organized as follows. We provide background on MATCHINGS and $b$-MATCHINGS in Section II. A serial, recursive, version of the algorithm we consider in this paper, the $b$-SUITOR algorithm, is briefly discussed in Section III. This algorithm has each vertex $u$ proposing to match to its neighbors in decreasing order of weights, provided the neighbor already does not have a better offer than the weight $u$ offers to it. Next we discuss the distributed-memory parallel version of the $b$-SUITOR algorithm in Section IV. Strategies to reduce the communication overhead are considered in Section V. Our experiments on two leadership class distributed memory multiprocessors and results are described in Section VI. We provide a summary of our results and conclude in Section VII.

## II. BACKGROUND AND RELATED WORK

We consider an undirected, simple graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/31.00 ©2016 IEEE

We denote $n \equiv |V|$, and $m \equiv |E|$. Given a function $b$ that maps each vertex to a non-negative integer, a $b$-MATCHING is a set of edges $M$ such that *at most* $b(v)$ edges in $M$ are incident on each vertex $v \in V$. (This corresponds to the concept of a simple $b$-MATCHING in Schrijver [4].) An edge in $M$ is matched, and an edge not in $M$ is unmatched. Similarly, an endpoint of an edge in $M$ is a matched vertex, and other vertices are unmatched. If $M$ has exactly $b(v)$ edges incident on each vertex $v$, then the $b$-MATCHING is perfect. An important special case is when the $b(v)$ values are the same for every vertex, say equal to $b$. In this case, a perfect $b$-MATCHING $M$ is also called a $b$-factor. For future use, we define $\beta = \max_{v \in V} b(v)$, and $B = \sum_{v \in V} b(v)$. We also denote by $\delta(v)$ the degree of a vertex $v$, and by $\Delta$ the maximum degree of a vertex in a graph $G$.

Now consider the case when there are non-negative weights on the edges, given by a function $w : E \mapsto R_{\geq 0}$. The weight of a $b$-MATCHING is the sum of the weights of the matched edges. The objective we consider is maximizing the weight of a $b$-MATCHING, and it is not necessarily a $b$-MATCHING of maximum cardinality.

Edmonds [5] devised the first exact algorithm for $b$-MATCHING. Pulleyblank [6] gave a pseudo-polynomial time algorithm with time complexity $O(mnB)$. Several other algorithms for exact $b$-MATCHING were proposed in [7], [8], [9], [10], [11], [12], [13]. A survey of exact algorithms for $b$-MATCHING was provided by [3]. More recently, Huang and Jebara [14] proposed an exact $b$-MATCHING algorithm based on belief propagation which assumes that the solution is unique, and otherwise it does not guarantee convergence. Since algorithms for computing a $b$-MATCHING of maximum weight have high time complexities, they are not practical for massive graphs with billions of edges, and they do not have much concurrency either. Hence we consider approximation algorithms for $b$-MATCHING here.

Relatively little work has been done on approximate $b$-MATCHING. Mestre [15] showed that a $b$-MATCHING is a relaxation of a matroid called a $k$-extendible system with $k = 2$, and hence that the Greedy algorithm gives a $1/k = 1/2$-approximation for a maximum weighted $b$-MATCHING. He generalized the Path-Growing algorithm of Drake and Hougardy [16] to obtain an $O(m\beta)$ time 1/2-approximation algorithm. He also generalized a randomized algorithm for MATCHING to obtain a $(2/3-\epsilon)$-approximation algorithm with expected running time $O(m\beta \log \frac{1}{\epsilon})$ [15]. Morales et al. [17] have adapted the GREEDY algorithm and an integer linear program (ILP) based algorithm to the MapReduce environment to compute $b$-MATCHING in bipartite graphs. There have been several attempts at developing fast $b$-MATCHING algorithms using linear programming [18], [19], but these are slow relative to the approximation algorithms discussed here.

The fastest serial as well as shared memory multi-threaded algorithm, $b$-SUITOR, was proposed by us [20]. In this paper, we compared a number of approximate $b$-MATCHING algorithms, and identified key algorithmic issues for this problem in general. We also showed that the $b$-SUITOR algorithm is suitable for parallelization and demonstrated that the algorithm scales up to 240 cores in shared memory settings.

There are few papers on approximate $b$-MATCHING in distributed memory settings. Koufogiannakis and Young [21] showed a randomized algorithm which guarantees 1/2- approximation. However, the authors did not mention an implementation or any performance results in that paper. Manshadi et al. [19] proposed $(1-\epsilon)$-approximate algorithm for bipartite graphs. The algorithm uses a linear programming (LP) formulation and it is implemented on a MapReduce environment. Geogiadis and Papatriantafilou [22] implemented a distributed $1/2$-approximation for $b$-MATCHING, based on local dominating edges. To the best of our knowledge, this is the only implementation for approximate $b$-MATCHING for general graphs; however, the largest graph considered by these authors has $1K$ vertices with $500K$ edges, and they did not report the runtime performance of the algorithm.

In case of approximate MATCHING, ($b(v)$ is 1 for all $v$), there are several algorithms proposed in [23], [24], [25] for distributed memory settings. An algorithm based on *locally dominant edges* was first proposed for computing a $1/2$-approximate MATCHING by Preis [26]. (An edge is locally dominant if it is at least as heavy as all other edges incident on its endpoints.) Manne and Bisseling [27] have described a distributed-memory parallel implementation of the locally dominant edge algorithm. The SUITOR algorithm was proposed by Manne and Halappanavar [28], on which the $b$-SUITOR algorithm is based; the former algorithm was implemented on shared memory parallel architectures by these authors. Since we describe the $b$-SUITOR algorithm in the next Section, we do not discuss the SUITOR algorithm in detail here. Manne et al. [29] proposed a distributed memory MATCHING algorithm which has the so called *self-stabilizing* property. In a *self-stabilizing* algorithm in a distributed memory setting, every vertex has only local information, i.e., it has knowledge about itself and only its neighbors. A self-stabilizing algorithm neither requires any global knowledge nor any fixed initial ordering to reach a stable solution. Lotker et al. [23], [30] proposed another algorithm which can handle dynamic graphs, i.e., where vertices leave and join the graph.

Blelloch, Fineman and Shun [31] have shown that the maximal (not maximum) matching problem can be solved in parallel in $O(\log^2 n)$ rounds with high probability using the locally dominant edge algorithm. They prove that the maximal independent vertex set (MIS) problem can be solved in this many rounds with high probability, and the maximal matching problem can be reduced to the MIS problem on the line graph of the original graph.

## III. SEQUENTIAL $b$-SUITOR

We begin by justifying our choice of a specific variant of the $b$-SUITOR algorithm which we implement to be scalable on a distributed memory multiprocessor. The $b$-SUITOR algorithm has several attractive properties that make it a good choice to be implemented on a distributed memory multiprocessor. First, it is a half-approximation algorithm that computes the same

$b$-MATCHING as the one obtained by a GREEDY algorithm and a Locally Dominant Edge (LD) algorithm (see Theorem 1 in the next Section, and [20]). Second, it has increased concurrency over the GREEDY and the LD algorithms: unlike the GREEDY algorithm which considers edges to match in decreasing order of weights, and the LD algorithm which matches an edge only when it becomes locally dominant, in the $b$-SUITOR algorithm vertices can extend proposals in arbitrary order, thus increasing concurrency in the algorithm. A vertex has to extend proposals to its neighbors in decreasing order of weights, but since each vertex can propose independent of others, this exposes sufficient parallelism in the algorithm. Second, the $b$-SUITOR algorithm has low serial running times relative to the GREEDY and the LD algorithms, as we will report in the next paragraph. It can be proved that the expected number of proposals in the SUITOR algorithm is $O(n \log n)$ if the weights of the edges are chosen randomly, although in the worst-case it can be $O(n^2)$ [32], [33]. (This follows from the connection of the SUITOR algorithm to an algorithm for the stable matching problem.) Finally, it can be proved that the expected value of the ranks (the sum over the vertices of the position of the matched neighbor in the sorted adjacency list of each vertex) is $O(n \log n)$, again when the weights are assigned randomly [33]. This last result suggests that each vertex needs to examine on the average at most $O(\log n)$ neighbors in its sorted adjacency list to find a mate in the SUITOR algorithm. We believe that these results, which hold for the SUITOR algorithm, can be generalized to the $b$-SUITOR algorithm as well.

Now we compare the run times of the exact $b$-MATCHING algorithm, and the half-approximate GREEDY, LD, and $b$-SUITOR algorithms on serial and shared memory processors. Since exact $b$-MATCHING algorithms are challenging to implement, there are few implementations that are publicly available. We compare our sequential $b$-SUITOR algorithm with two exact algorithms: the first, an algorithm that solves the Integer Linear Programming (ILP) formulation of $b$-MATCHING and the second, an algorithm based on belief propagation (BP) [14]. The BP algorithm is not guaranteed to converge if the $b$-MATCHING is not unique. The serial $b$-SUITOR algorithm is 895× faster than ILP, and 287× faster than the BP algorithm on the test set used in [20]. When compared with other approximation algorithms, $b$-SUITOR is 17× faster than the GREEDY algorithm, and 3× faster than the Locally Dominating edge (LD) based algorithm [20] on a serial machine. On a shared memory multiprocessor with 16 threads, $b$-SUITOR is 14× faster than LD algorithms, and the former scales better than the latter with increasing numbers of threads. In summary, $b$-SUITOR algorithm is the fastest among these algorithms, and it is scalable in a multithreaded shared memory context. The $b$-SUITOR algorithm requires only local information, as described later in this section, which is advantageous in the distributed memory context.

We now describe a serial version of the $b$-SUITOR algorithm. This algorithm is a $1/2$-approximation algorithm for maximum edge weighted $b$-MATCHING, and was proposed in [20]. For each vertex $u$, we maintain a priority queue $S(u)$ that contains at most $b(u)$ elements from its adjacency list $N(u)$. The intent of this priority queue is to maintain a list of neighbors of $u$ that have proposed to $u$ and hence are Suitors of $u$. The priority queue enables us to update the lowest weight of a Suitor of $u$ in $\log b(u)$ time. If $u$ has fewer than $b(u)$ Suitors, then this lowest weight is defined to be zero. The operation $S(u).insert(v)$ adds the vertex $v$ to the priority queue of $u$ with the weight $w(u, v)$. If $S(u)$ has $b(u)$ vertices, then the vertex with the lowest weight in the priority queue is discarded on insertion of $v$. This lowest weight Suitor is stored in $S(u).last$; if the priority queue contained fewer than $b(u)$ vertices, then a value of $NULL$ is returned for $S(u).last$.

In what follows, we will need to break ties consistently when the priorities of two vertices are equal. Without loss of generality, we will say that $w(u, x) > w(v, x)$ if the weights are equal but vertex $u$ is numbered lower than $v$.

It is also conceptually helpful to consider an array $T(u)$ which contains the vertices that $u$ has proposed to. These are all the vertices $v$ such that $u$ is a Suitor of $v$. Again, there are at most $b(u)$ neighbors of $u$ in the set $T(u)$, and so this is a subset of $N(u)$. The operation $T(u).insert(v)$ inserts a vertex $v$ into the array $T(u)$, and $T(u).remove(v)$ removes the vertex $v$ from $T(u)$. Throughout the algorithm, we maintain the property that $v \in S(u)$ if and only if $u \in T(v)$. When the algorithm terminates, we satisfy the property that $v \in S(u)$ if and only if $u \in S(v)$, and then $(u, v)$ is an edge in the $b$-MATCHING.

Consider what happens when we attempt to find the $i$-th neighbor for a vertex $u$ to propose to. At this stage $u$ has made $i - 1$ outstanding proposals to vertices in the set $T_{i-1}(u)$, the index showing the number of proposals made by $u$. We must have $i \leq b(u)$, for $u$ can have at most $b(u)$ outstanding proposals. If a vertex $u$ has fewer than $b(u)$ outstanding proposals, then we say that it is *unsaturated*; if it has $b(u)$ outstanding proposals, then it is *saturated*. The $b$-SUITOR algorithm finds a partner for $u$, $p_i(u)$, according to the following equation:

$$p_i(u) = \underset{v \in N(u) \setminus T_{i-1}(u)}{\arg\max} \{w(u, v) \mid w(u, v) > w(v, S(v).last)\} \tag{1}$$

In words, the $i$-th vertex that $u$ proposes to is a neighbor $v$ that it it has not proposed to yet, such that the weight of the edge $(u, v)$ is maximum among such neighbors, and is also greater than the lowest weight offer $v$ has currently. We will call such a vertex $v$ an *eligible partner* for $u$ at this stage in the algorithm. Note that the vertex $p_i(u)$ belongs to $T_i(u)$ but not to $T_{i-1}(u)$.

We present the pseudo-code for the sequential $b$-SUITOR algorithm in Algorithm 1. A recursive version of the algorithm is described since it is easier to understand, although the versions we have implemented for both serial and parallel algorithms use iteration rather than recursion. The algorithm processes all of the vertices, and for each vertex $u$, it seeks to propose to $b(u)$ neighbors. In each iteration a vertex $u$

**Algorithm 1** $b$-SUITOR$(G, b)$

```
for all $u \in V$ do
    for $i = 1$ $to$ $b(u)$ do
        $x = \underset{v \in N(u) \setminus T(u)}{\arg\max} \{w(u,v) : w(u,v) > w(v, S(v).last)\}$
        if $x = NULL$ then
            break
        else
            MakeSuitor($u, x$)
```

**Algorithm 2** MakeSuitor$(u, x)$

```
$y = S(x).last$
$S(x).insert(u)$
$T(u).insert(x)$
if $y \neq NULL$ then
    $T(y).remove(x)$
    $z = \underset{v \in N(y) \setminus T(y)}{\arg\max} \{w(y,v) : w(y,v) > w(v, S(v).last)\}$
    if $z \neq NULL$ then
        MakeSuitor($y, z$)
```

proposes to a heaviest neighbor $v$ it has not proposed to yet, if the weight $w(u, v)$ is heavier than the weight offered by the last ($b(v)$-th) Suitor of $v$. If it fails to find such a vertex, then we break out of the loop. If it succeeds in finding an eligible vertex $x$ to propose to, then the algorithm calls the function MakeSuitor to make $u$ the Suitor of $x$. This function updates the priority queue $S(u)$ and the array $T(u)$. When $u$ becomes the Suitor of $x$, if it annuls the proposal of the previous Suitor of $x$, a vertex $y$, then the algorithm looks for an eligible partner $z$ for $y$, and calls MakeSuitor recursively to make $y$ a Suitor of $z$.

There are some modifications that can improve the performance of the basic $b$-SUITOR algorithm.

One modification is to sort the adjacency lists of the vertices to list edges in decreasing order of weights to reduce the time complexity of the algorithm. With sorting, the adjacency list of each vertex needs to be scanned from the highest to the lowest only once in the entire algorithm. This feature reduces the time complexity of the $b$-SUITOR algorithm from $O(m\Delta \log \beta)$ to $O(m \log(\beta\Delta)$. This could be further reduced by partially sorting the adjacency lists, since only some multiple of the $b(v)$ heaviest edges incident on $v$ are likely to be involved in an approximate $b$-MATCHING. The complexity of the partially sorted variant is $O(m(c + \log \beta)$, where $c$ is the maximum number of subsets in an adjacency list that is sorted; this is typically bounded by a constant.

The second modification concerns what to do when a vertex $u$ has one of its proposals annulled. Either we can process vertex $u$ immediately so that it proposes to its next eligible neighbor, or we can put $u$ into a queue for later processing after all vertices in the current iteration. The second, delayed processing option, leads to better cache accesses, and to fewer proposal annulments (since a vertex whose proposal is annulled is likely to have a low weight relative to other eligible vertices). It is this Delayed, Partial sorting (DP) variant that we consider in the distributed-memory setting.

## IV. MULTINODE $b$-SUITOR

We describe the distributed memory $b$-SUITOR algorithm in this section. If we set aside implementation details that arise due to the distributed-memory setting, the algorithm is conceptually simple. Referring to the Algorithms 1 and 2 discussed in the previous section, the heart of the $b$-SUITOR algorithm is that a vertex $u$ makes a proposal to another vertex $v$ by following a rule (corresponding to the invariant of the algorithm, Equation 1), and in doing so it may annul a proposal to $v$ which was made earlier by another vertex $w$. Also, when $u$ proposes to $v$, its current value of the best offer that $v$ has might not be correct, and thus $v$ might need to send a rejection message to $u$. In a distributed-memory algorithm, the three vertices $u$, $v$, and $w$ could be in three different compute nodes, and hence we need an inter-node message passing interface to coordinate the operations.

The input graph, $G(V, E, w)$ is distributed among the participating compute nodes. For each compute node, we denote the resident graph as $G_l = (V, E_l, w_l)$ where $V = V_l \cup V_r$; i.e., $V_l$ is the set of vertices local to that node, and $V_r$ is the set of vertices remote to that node. For a specific compute node, we do not need to consider all the remote vertices but the subset of the remote vertices which are neighbors of at least one local vertex. We call this subset of vertices as ghost vertices, and denote them by $V_g \subseteq V_r$. We call the subset of compute nodes that own the ghost vertices on a compute node as its neighboring nodes. The set $E_l$ denotes the edges induced by $V_l$, where at least one end point of an edge $e(u, v) \in E_l$ is local, and $w_l$ denotes the set of weights of such edges. The heap data structures for the local vertices $S(v_l)$ are exactly same as we have described in the previous section. However, for the ghost vertices, we keep only the *last Suitor* information.

We have three types of messages: i) PROP for *proposals*, ii) REJ for *rejections* and iii) ANL for *annulments*. In the sequential $b$-SUITOR algorithm, there was no notion of *rejections*, and this is one of the critical differences for the algorithm in the distributed setting. Processors do not send messages one at a time but in batches for obvious performance reasons. A processor maintains separate sending and receiving buffers for each neighboring compute node. If a compute node needs to communicate with a ghost vertex $v$, the algorithm first identifies the remote node $r$ which owns vertex $v$, and then creates an appropriate message for that vertex and adds it to the send buffer $SB_r$, assigned to that remote node.

Now we describe our multi-node $b$-SUITOR in Algorithms 3 and 4. The algorithm uses iteration rather than recursion. Since the Delayed Partial (DP) variant of the $b$-SUITOR algorithm performs the best in both sequential as well as multi-thread shared memory settings [20], our base algorithm in the distributed-memory setting is this variant. As our implementation employs a hybrid shared and distributed memory programming model, the intra-node computation is done in parallel with OpenMP, and the inter-node communication is done using the MPI library.

The algorithm maintains a queue of unsaturated vertices $Q$

for which it tries to find partners by extending proposals, and also a queue of vertices $Q'$ that become unsaturated during the current iteration (through annulments) to be processed again in the next iteration. When there is no more vertex to be processed, the algorithm terminates. The algorithm attempts to find $b(u)$ partners for each vertex $u$ in $Q$, (line 5) as long as its neighborhood has not been exhausted.

Consider the situation when a vertex $u$ has $i-1 < b(u)$ vertices outstanding proposals and the vertex $u$ finds an eligible partner $p$ by satisfying Equation 1 (line 8). There are two possible scenarios, i.e., $p$ is either a local or a ghost vertex. If it is a ghost vertex then the algorithm creates a Proposal message and adds it to the appropriate send buffer (line 21). If $p$ is a local vertex then the thread processing the vertex $u$ attempts to acquire the lock for the priority queue $S(p)$ so that other vertices do not simultaneously become Suitors of $p$. This attempt might take some time to succeed since another thread might have the lock for $p$, or could be competing for it. Once the thread processing $u$ succeeds in acquiring the lock, then it needs to check again if $p$ continues to be an eligible partner, since by this time another thread might have found another Suitor for $p$, and its lowest offer might have changed. If $p$ is still an eligible partner for $u$, then we make $u$ as a suitor of $p$. By making $u$ a suitor of $p$, we may dislodge the lowest weight Suitor $v$ of $p$, i.e., we annul the proposal that $v$ made to $p$ earlier (line 14). Again, what happens next depends on whether vertex $v$ is a local or ghost vertex. If it is a local vertex then we add $v$ to the queue of vertices $Q'$ to be processed in the next iteration (line 15) and the thread releases the lock on $S(p)$. If $v$ is a ghost vertex, then the algorithm creates an annulment message and adds it to the appropriate send buffer.

In the $Remote\_msg\_handle$ procedure, the algorithm sends all the messages gathered during the computation phase to the respective remote nodes using the asynchronous MPI send primitive. The reason for the asynchrony is that we do not know when the computation phases of other compute nodes will finish. Since $b$-SUITOR algorithm has the inherent *self stabilizing* property and also has separate queues for receiving messages, we do not need to wait for the outstanding send primitives to finish. Thus we hide much of the communication latency. Next, the compute node waits until it receives incoming messages from any of the remote nodes. As soon as it receives an incoming message from a remote node $r$, it starts to process the message. In doing so, we again hide the communication latency because by the time the algorithm is finished processing messages from $r$, it is likely that messages from other remote nodes are already in the respective receive buffers, $RB$s.

We process all the messages coming from a remote node $r$ in parallel (line 6). Consider a message $m$ received from remote node $r$, with $u$, $p$ and $w$ being the source (ghost) vertex, destination (local) vertex and the weight for the message $m$, respectively. There are two cases based on the message types.

1) **Proposal message:** If it is a Proposal message, then we need to check whether it is a valid proposal or not, i.e., if $w > S(p).last$. This validity checking is required because when $u$ decided that $p$ is an eligible partner to propose to (Algorithm 3 line 8), it made the decision based on the ghost information about $p$, and the information may not be current at that time. If the proposal is still valid, then we make $u$ a Suitor of $p$ with locking/unlocking mechanism for thread synchronization as before. If during this process, $u$ annuls a proposal of $v$ to $p$ then again we handle it as described in Algorithm 3.
   However, if the proposal is not a valid one, the algorithm replies to vertex $u$ with a Rejection message. An important point to note is that the proposal was rejected because $u$ made a decision based on the stale information about the last Suitor of $p$. So when the algorithm sends the Rejection message to $u$, it updates the current last Suitor information $S(p).last$ in the Rejection message (line 18).
2) **Rejection or Annulment message:** We treat both types of messages in the same manner because in effect, the algorithm requires us to find a new partner for $p$ for both cases. In case of rejection, the proposal to $u$ got rejected because of the stale information of $u$. In case of annulment, another vertex $v$ made an better offer than $p$. For both cases the worst suitor information of $u$ has been updated. So we update this information which is encoded in the message (line 20) and add vertex $p$ to $Q'$ to process it in the next iteration.

The distributed $b$-SUITOR algorithm is a $1/2-$ approximation algorithm. Its proof directly follows from the proof of the sequential $b$-SUITOR algorithm in [20]. We omit this proof and the proof for termination of the algorithm due to space limitations. We can prove a stronger result.

*Theorem 1:* All of the four algorithms: the distributed $b$-SUITOR, the serial $b$-SUITOR, the GREEDY, and the Locally Dominant edge algorithms, compute the same $b$-MATCHING, and hence are $1/2$-approximation algorithms for the maximum weighted $b$-MATCHING problem.

## V. STRATEGIES FOR REDUCING COMMUNICATION

The distributed memory $b$-SUITOR algorithm is communication-intensive. The communication volume depends on many factors such as the distribution of the edges and their weights, the values of $b(v)$, and the partitioning of the graph among the processors, etc. Except for the last factor, the other factors are input to the algorithm. We use a simple vertex partitioning for our experiments, i.e., vertices in the original order in the graph are equally distributed among the participating compute nodes. We show and discuss the effect on runtime for random permutation of vertices among the nodes in the experimental section. Clearly, a partition that minimizes the total number of cut edges should reduce the communication volume but partitioning itself is a heavy-weight task in distributed settings. Since we compute matchings in our test problem under 10 seconds, and a good partitioning algorithm requires much more time, partitioning

**Algorithm 3** Distributed memory algorithm for approximate $b$-MATCHING. **Input:** A graph $G_l = (V, E_l, w_l)$ where $V = V_l \cup V_r$, vectors $b_l$ and $s_l$. (Here $l$ denotes local values, and $r$ denotes remote values for this processor.) **Output:** A $1/2-$approximate edge weighted $b$-MATCHING $M$.

```
 1: procedure MULTINODE_b-SUITOR($G_l, b, s$)
 2:     $Q = V_l$; $Q' = \emptyset$;
 3:     All buffers are initially empty;
 4:     while $Q \neq \emptyset$ do
 5:         for all vertices $u \in Q$ in parallel do
 6:             $i = 1$;
 7:             while $i <= b(u)$ and $N(u) \neq exhausted$ do          ▷ Extend $i$-th proposal from $u$
 8:                 Let $p \in N(u)$ be an eligible partner of $u$;
 9:                 if $p \neq NULL$ then
10:                     if $p$ is local then          ▷ Adding and updating the heap of a local vertex
11:                         Lock $p$;
12:                         if $p$ is still eligible then
13:                             Make $u$ a Suitor of $p$;
14:                             if $u$ annuls the proposal of a vertex $v$ then
15:                                 if $v$ is local then
16:                                     Add $v$ to $Q'$; Update $db(v)$;
17:                                 else          ▷ Remote annulments are added to Message queues
18:                                     Let $r$ be the node that owns $v$;
19:                                     Add an Annulment message to the send buffer $SB_r(u, v, S(p).last, ANL)$
20:                             $i = i + 1$;
                        Unlock $p$;
21:                     else          ▷ Remote proposals are added to Message queues
22:                         Let $r$ be the node that owns $p$;
23:                         Add a Proposal message to the send buffer $SB_r(u, p, w(u, p), PRP)$;
24:                 else
25:                     $N(u) = exhausted$;
26:         /* Start of communication phase */
27:         Remote_msg_handle($SB, RB, Q', db$);
28:         Update $Q$ using $Q'$; Update $b$ using $db$;
```

as a preprocessing step will not improve the performance when the cost of partitioning is included.

Another source of redundant communications in Algorithm 3 is Proposal messages to vertices based on the ghost information. The ghost information on a processor might be stale at the time of a decision to extend a proposal, and this can generate many Rejection messages (we consider this in line 9 in Algorithm 4). If the information was correct and not stale, then we could have saved two messages (one Proposal message and the corresponding Rejection message) per decision. One extreme solution could be to send update messages as soon as the $S(u).last$ is updated for a vertex $u$, to all the compute nodes where $u$ is a ghost vertex. But that would mean: i) all compute nodes have to be in lockstep (synchronized communication) and ii) there would be many small messages across the interconnection network. Both of these lead to poor performance. So we consider three strategies to reduce the number of messages in the following.

### A. Subsetting the $b(v)$ values

We define $1 \leq b'(v) \leq b(v)$ for each node $v$. Instead of finding $b(v)$ partners for a vertex $v$ (Algorithm 3, line 7), we can find a batch of $b'(v)$ partners at each iteration until all partners are found. Since the adjacency lists are sorted, we find partners with higher edge weights first. The reason for this is that the weight of the last Suitor of a vertex can only increase in the course of the algorithm, so we let a vertex extend proposals to its heavier neighbors first and become their Suitor, and the corresponding ghost information about the last Suitor spreads in the graph first. Later on when the algorithm tries to find Suitors with lower edge weights, there is a reduced chance that the offers from lower weight neighbors can beat the offer of the last Suitor even if the ghost information is not fresh. We apply two different strategies for choosing the $b'$ values: i) constant $b'$, where $b'(v) \in \{1, 3, 5, \ldots, b(v)\}$, and ii) variable $b'$, where $b'(v) = 1/2b(v)$ or $(b(v)/d(v)) \times b(v)$.

### B. Subsetting the vertices on a compute node

Instead of subsetting the $b(v)$ values, we can process the vertices on a specific processor in subsets. Let us assume

**Algorithm 4** Procedure for remote message handling. **Input:** Send and Receive buffers, $SB$ and $RB$, for each remote node, a queue $Q'$ and a vector $db$. **Output:** All messages in Receive buffers are processed. The values of $SB$, $Q'$ and $db$ are updated.

```
procedure REMOTE_MSG_HANDLE($S, R, Q', db$)
    for all Neighboring Nodes $r$ do
        Send message $SB_r$;
    for all Neighboring nodes do                                    ▷ Wait for message from any remote node
        Let $r$ be the node from which data is received;
        for all messages $m \in RB_r$ in Parallel do
            Let $u$ be the remote vertex and $p$ the local vertex in $m$;
            if $m$ is a Proposal then                                ▷ Handling proposal messages
                if valid Proposal then
                    lock $p$; Make $u$ a Suitor of $p$; unlock $p$;
                    if $u$ annuls the proposal of a vertex $v$ then
                        if $v$ is local then
                            Add $v$ to $Q'$; Update $db(v)$;
                        else
                            Let $r'$ be the node that owns $v$;
                            Add an Annulment message to the send buffer $SB_{r'}(p, v, S(p).last, ANL)$
                else
                    Add a Reject message to the send buffer $SB_r(p, u, S(p).last, REJ)$
            else                                                     ▷ Handling annulment or reject messages
                Update Last Suitor of $u$ using $S(u).last \in m$;
                Add $p$ to $Q'$, update $db(p)$;
```

that a compute node in its $t^{th}$ iteration has to process $n_t$ unsaturated vertices. Instead of processing all $n_t$ at the same time in parallel, we can process a subset of $p$ vertices, and then do the communications. In the distributed memory setting this strategy has been called *super-stepping*, which has been shown to load-balance the communications among the compute nodes by making a trade-off between the freshness of the last Suitor information and the volume of communication. In our case, this strategy gives us an added benefit since by controlling the number of vertices being processed in a superstep (one round of computation and communication), we control the volume of last Suitor updates in $S(v)$. Since each $S(v)$ update ideally requires us to propagate the information, we reduce the number of such updates by processing fewer vertices at each step. We observe experimentally that the volume of stale updates decreases by applying this strategy.

| Problems | Vertices | Edges | Avg. deg |
|---|---|---|---|
| **ER_28** | 268,434,430 | 2,147,483,648 | 8 |
| **ER_27** | 134,217,028 | 1,073,741,824 | 8 |
| **ER_26** | 67,107,760 | 530,160,025 | 8 |
| **SSCA_28** | 268,435,154 | 2,136,323,325 | 8 |
| **SSCA_27** | 134,217,728 | 1,066,851,217 | 8 |
| **SSCA_26** | 67,107,987 | 534,179,576 | 8 |
| **G500_27** | 134,217,726 | 2,111,641,641 | 16 |
| **G500_26** | 67,108,089 | 1,073,058,343 | 16 |
| **G500_25** | 33,554,330 | 532,507,217 | 16 |
| **twitter** | 41,652,230 | 1,468,365,182 | 36 |
| **gsh-2015-host** | 68,680,142 | 1,802,747,600 | 27 |

TABLE I
TEST PROBLEMS.

### C. Ordering the vertices for extending proposals

Intuitively, edges with higher weights are more likely to be matched since our goal is to find a matching of maximum weight. Hence when a vertex makes a proposal with a higher edge weight, it is less likely to be annulled later by another vertex. So we sort the vertices according to the heaviest edge incident on them, and process them in that order. For example, if two vertices $u$ and $v$ have $w_u$ and $w_v$ respectively as their heaviest remaining edges incident on them, we process $u$ before $v$ if $w_u > w_v$. (This simple strategy does as well as more sophisticated ones that sort by estimating the value of the last Suitor of a vertex.) For the sequential algorithm, processing in this order indeed reduces the number of annulments by as much as 15% for problem with 500 million edges ([20]), In the distributed setting, message-based communication is costlier than the computation. As the number of compute nodes increases, for the same graph, each node owns fewer vertices, and the overhead cost of sorting the adjacency lists decreases. Therefore, for each compute node, we sort $Q$ according to the heaviest remaining edge before processing the vertices (before line 5 in Algorithm 3).

## VI. EXPERIMENTS AND RESULTS

We conducted our experiments on Edison and Cori, two leadership-class machines at NERSC, Berkeley. Edison is a Cray XC30 supercomputer, whose compute node consists of two 12-core 2.4 GHz Intel® Ivy Bridge processors with 64GB RAM. Each core in a compute node has its own

64KB L1 cache and 256KB L2 cache, as well as a 30MB shared L3 cache per socket. Edison nodes are interconnected with the Cray Aries network using a Dragonfly topology with 5.625TB/s global bandwidth. We used the Intel® MPI implementation for inter-node communication and OpenMP for intra-node multi-threading, and compiled the code with Intel® compiler [1] *mpiicpc-5.1.1* with -O3 -openmp flags. Our hybrid implementation used two MPI-openMP settings: $i$) 12 cores per MPI process, where each MPI process is placed in an Edison compute node, and $ii$) all 24 cores per MPI process and 1 MPI process per node. We find that the latter configuration gives better performance. Cori is a Cray XC40 supercomputer, with each compute node consisting of two 16-core 2.3 GHz Intel® Haswell processors with 128GB RAM. Each core in a node has its own 64KB L1 cache and 256KB L2 cache, as well as a 40MB shared L3 cache per socket. Cori nodes are also interconnected with the Cray Aries network using a Dragonfly topology.

Table I describes the set of problems for the experiments. We used synthetic datasets based on RMAT [34] as well as real world problems obtained from [35], [36]. We generated three classes of RMAT graphs: (a) G500 representing graphs with skewed degree distribution from the Graph 500 benchmark [37], (b) SSCA from HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark [38], and (c) Erdos-Renyi random graphs with uniform degree distributions. We used the following parameter settings:
(a) $a = 0.57$, $b = c = 0.19$, and $d = 0.05$ for G500,
(b) $a = 0.6$, and $b = c = d = 0.13\overline{3}$ for SSCA,
and (c) $a = b = c = d = 0.25$ for ER.
We have generated three scales for each class of graph where the scale $k$ determines the number of vertices, $n = 2^k$. For ER and SSCA graph the scales are $k \in \{26, 27, 28\}$ and for G500 graphs they are $k \in \{25, 26, 27\}$. We choose $b(v) = min\{10, \delta(v)\}$ for all the problems where $\delta(v)$ is the degree of vertex $v$. High $b(v)$ values relative to the average degree means that the algorithm has more work per vertex and also that the communication across the network is expected to be high. Since the $b$-SUITOR algorithm is communication-intensive, we investigate its performance under high communication loads.

At first, we investigate the best performing scheme for each communication reduction strategy. The number of combinations is large, i.e., we find the best performing scheme for $b(v)$ subsetting, where we have $b'(v) \in \{\{1, 3, \ldots, b(v)\}, 1/2b(v), b(v)/d(v) \times b(v)\}$, for a total of eight choices. Then for each of these choices, we apply the vertex subsetting policies, where we partition the vertices in each compute node into $\{1, 8, 16, 32, 64, 128\}$ subsets, for a total of six values. Then we repeat these $6 \times 8 = 48$ experiments with vertices sorted according to the heaviest edge weight incident on it. We run these experiments for the largest problems in each class and we report results from the best strategies in Figure 1.

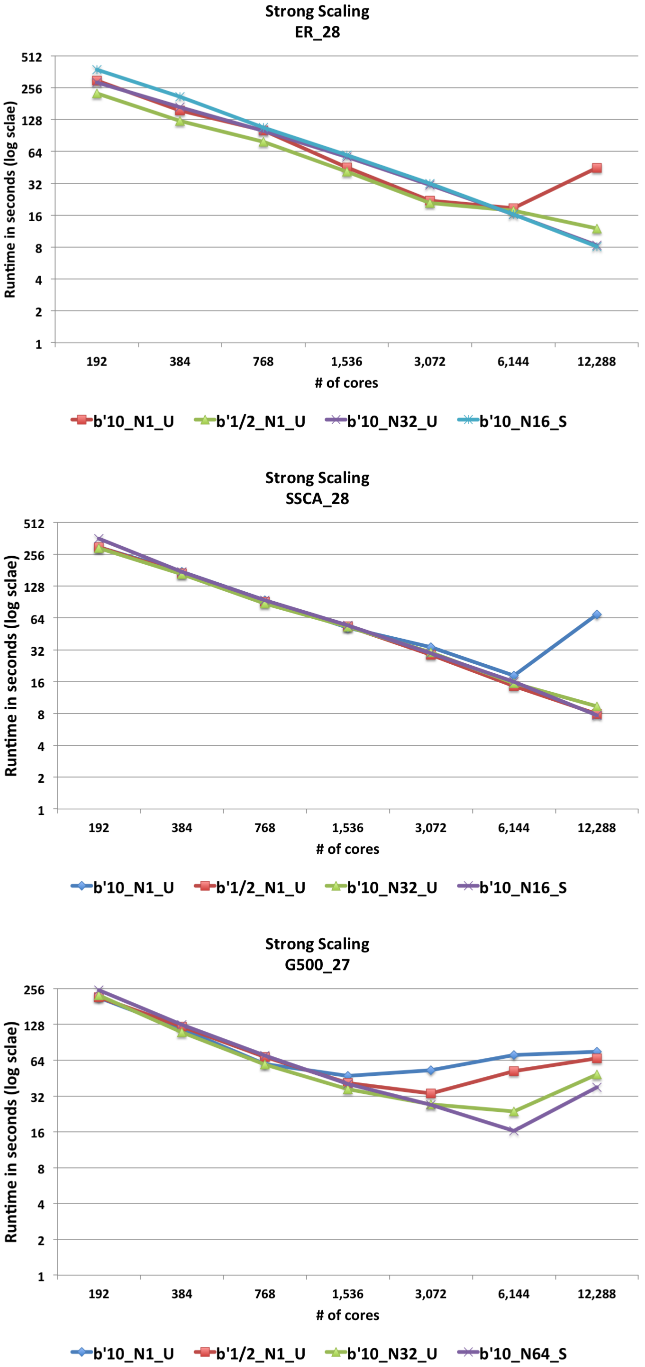

Fig. 1. Strong scaling results for different strategies for three classes of problems on Edison.

The strategy information is encoded in the legends of Figure 1. For example, $b'10$_N16_S in the ER_28 plot denotes that under the vertex sorting strategy: vertex subsetting with 16

[1] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel micro-architecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

subsets and $b' = 10$ (i.e., no $b(v)$ subsetting because $b(v) \leq 10$) gives the best performance. We plot the number of cores on the $x$-axis and logarithms of the runtimes in seconds on the $y$-axis.

We observe that for all three classes of graphs the basic strategy, ($b'$10_N1_U), is the worst performer. For ER and SSCA problems, the basic strategy scales up to 6K cores where as for G500_27, it scales up to 1.5K cores. All of the communication strategies make the algorithm scale better, and the combination of sorting the vertices with vertex subsetting but no $b(v)$ subsetting, is the best strategy for all problems. It is interesting to note that the behaviors of vertex subsetting and $b(v)$ subsetting are orthogonal to each other. When we use $b(v)$ subsetting, no vertex subsetting (i.e., N1) gives the best result, and vice versa. In summary, the $b$-SUITOR algorithm scales up to 12K cores with speed up of $47\times$ (ideally, $64\times$) for ER and SSCA problems. (Speedups are computed relative to 192 cores.) However for the G500 problem, it scales up only to 6K cores with $15\times$ (ideally, $32\times$). The under-performance of this class of graph is due to its skewed degree distribution, since the number of edges in each compute node can be highly imbalanced.

Next, we investigate why these strategies improve the performance by considering how it reduces the communication volume with respect to the basic strategy. By subsetting on $b(v)$ values and the vertices, we aim to reduce the stale information (last Suitor weights) to reduce rejections. Vertex ordering for extending proposals aims to reduce annulments. However, reducing annulments indirectly reduces rejections also, because the algorithm chooses the vertices with heavier edges to make proposals first. Hence we observe that sorting the vertices coupled with vertex subsetting performs the best. Table II verifies our claim and shows the percent reduction in the number of proposals, rejections, annulments and total number of messages with respect to the basic strategy for the largest problems in the three graph classes.

Since the basic strategy scales up to 256 nodes for ER_28 and SSCA_28, and 128 nodes for G500_27, we compare the number of different types of messages for the basic strategy with the other strategies with these node counts. The number of proposals and total number of messages are related to the numbers of rejections and annulments, so we focus on these two types. As claimed, subsetting $b(v)$ and vertices (second and third columns of Table II) reduces rejections for all the problems (as much as 20% for ER_28). However, these two strategies do not reduce the annulments that much. In fact for SSCA_28 and G500_27, the annulments increase by 2% with vertex subsetting. The last column of Table II shows the reduction with vertex sorting and vertex subsetting. We see that it reduces both rejections and annulments more than other strategies for all problems, and this is why it performs the best in terms of run times and scaling.

Next, we investigate the total number of proposal messages and proposal messages per node (in $\log_2$ scale) as a function of compute nodes in Figure 2. We observe that the number of proposal messages initially increases and then stabilizes for ER_28 and SSCA_28 graphs, but for G500_27 graph it keeps increasing with the number compute nodes. This is another indication of the relatively poor strong scaling for G500 graphs. We observe that the average number of proposal messages generated per node goes down for all the graphs, but the slope is more negative for ER and SSCA graphs.

| **ER_28 (256)** | $b'$**1/2_N1_U** | $b'$**10_N32_U** | $b'$**10_N16_S** |
|---|---|---|---|
| **Proposal** | 3.23% | 3.58% | **6.91%** |
| **Rejection** | 20.72% | 11.15% | **21.18%** |
| **Annulment** | 0.33% | 12.75% | **24.90%** |
| **Total** | 4.97% | 5.51% | **10.62%** |

| **SSCA_28 (256)** | $b'$**1/2_N1_U** | $b'$**10_N32_U** | $b'$**10_N16_S** |
|---|---|---|---|
| **Proposals** | 7.37% | 8.13% | **11.62%** |
| **Rejection** | 16.58% | 17.63% | **19.22%** |
| **Annulment** | 17.03% | *-1.55%* | **31.12%** |
| **Total** | 9.84% | 8.62% | **14.87%** |

| **G500_27 (128)** | $b'$**1/2_N1_U** | $b'$**10_N32_U** | $b'$**10_N64_S** |
|---|---|---|---|
| **Proposals** | 0.14% | 5.75% | **9.16%** |
| **Rejection** | 0.95% | 12.16% | **7.08%** |
| **Annulment** | 4.51% | *-2.19%* | **40.12%** |
| **Total** | 0.78% | 7.12% | **11.16%** |

TABLE II
REDUCTION IN NUMBER OF MESSAGES W.R.T THE BASIC STRATEGY FOR THE LARGEST INPUT SIZE IN EACH PROBLEM CLASS.

Now we investigate the sensitivity of the algorithm to different distributions of vertices among the compute nodes. In order to test this, we randomly permuted the vertices, and then mapped contiguous subsets of vertices to different compute nodes (keeping the number of local vertices the same). Then we performed three sets of experiments on the largest problems with different permutations for the basic and the best strategies. We report the mean runtimes in the histogram and the standard deviations as the error bars in Figure 3. The best strategy is mostly insensitive to the permutation except for G500_27. The result for the last graph class is expected since here with different permutations the edge distribution will be significantly changed.

Finally, we report results on Cori for the largest problems in each class with their best strategies. Figure 4 shows the strong scaling performance of $b$-SUITOR algorithm. We observe a speedup of 39 (ideally it would be 64) in going from 256 cores to $16K$ cores for RMAT. Run times are faster for the same number of compute nodes for Cori relative to Edison. E.g., the ER_28 graph takes 59 seconds with 128 nodes on Edison, whereas it takes 26 seconds on Cori, because a Cori node has more cores than an Edison node. The G500_27 graph scales up to 16K cores (512 nodes) on Cori, whereas it scaled only to 6K cores on Edison. We also experiment with the two real world problems on Cori. The smaller problem `twitter` has 1.5 billion edges, and scales up to 8K cores; the larger problem `gsh-2015-host` has 1.8 billion edges, and scales up to 16K cores. We observe nearly constant weak scaling performance on Cori in Figure 5.

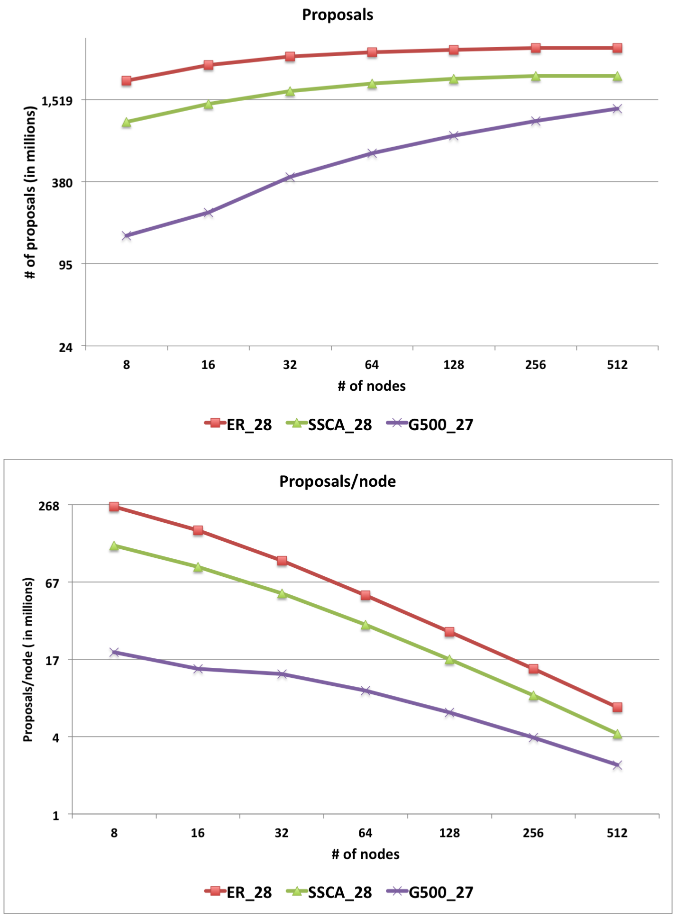

Fig. 2. The relationship between the number of Proposal messages and the number of compute nodes.

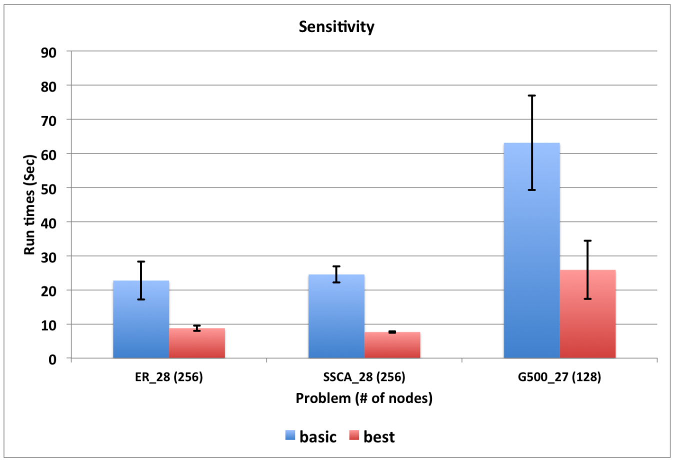

Fig. 3. Performance sensitivity under random partitioning.

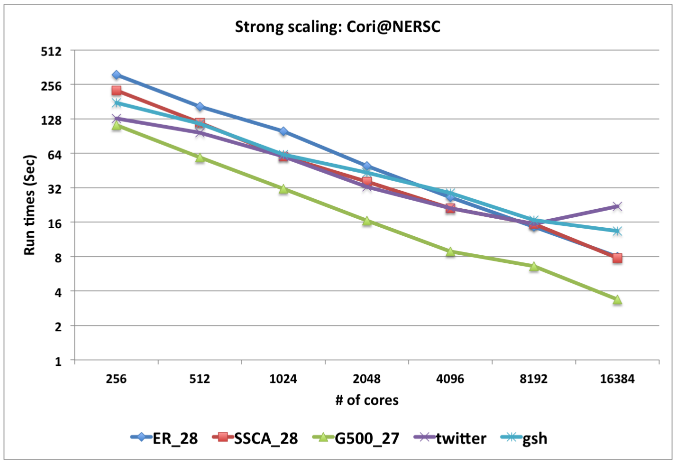

Fig. 4. Strong scaling with Cori.

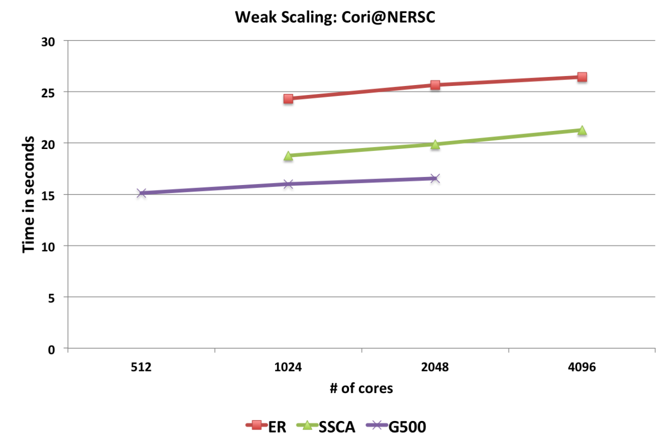

Fig. 5. Weak scaling results for different strategies with three classes of problems on Cori.

## VII. Conclusions

We have designed the first distributed-memory parallel algorithm for $b$-MATCHING that scales to 16K cores of a supercomputer. We demonstrated both strong scaling and weak scaling on three synthetic graphs as well as two graphs from applications. We computed a half-approximate maximum weighted $b$-MATCHING in a graph with 2 billion edges in less than $4$ seconds on 16K cores of Cori.

**Acknowledgments**. We thank Prabhat, Group Leader of the Data and Analytics Services, for access to Cori and Edison at NERSC, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Ths work was supported by grants from the U.S. National Science Foundation CCF-1552323 and the U.S. Department of Energy (DOE) DE-SC0010205, and the Applied Mathematics Program of the Office of Advanced Scientific Computing Research within the Office of Science of the DOE at the Pacific Northwest National Laboratory, which is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

# References

[1] K. M. Choromanski, T. Jebara, and K. Tang, "Adaptive anonymity via $b$-matching," in *Advances in Neural Information Processing Systems*, 2013, pp. 3192–3200.

[2] T. Jebara and V. Shchogolev, "$b$-matching for spectral clustering," in *European Conference on Machine Learning*. Springer, 2006, pp. 679–686.

[3] M. Müller-Hannemann and A. Schwartz, "Implementing weighted b-matching algorithms: Insights from a computational study," *J. Exp. Algorithmics*, vol. 5, Dec. 2000.

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

[5] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of Research of the National Bureau of Standards - B*, vol. 69B, pp. 125–130, 1965.

[6] W. R. Pulleyblank, "Faces of matching polyhedra," Ph.D. dissertation, Faculty of Mathematics, University of Waterloo, 1973.

[7] H. N. Gabow, "An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems," *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pp. 448–456, 1983.

[8] A. B. Marsh III, "Matching algorithms," Ph.D. dissertation, The John Hopkins University, Baltimore, 1979.

[9] R. P. Anstee, "A polynomial algorithm for $b$-matching: An alternative approach," *Information Processing Letters*, vol. 24, pp. 153–157, 1987.

[10] U. Derigs and A. Metz, "On the use of optimal fractional matchings for solving the (integer) matching problem," *Computing*, vol. 36, pp. 263–270, 1986.

[11] D. L. Miller and J. F. Pekny, "A staged primal-dual algorithm for perfect $b$-matching with edge capacities," *ORSA J. of Computing*, vol. 7, pp. 298–320, 1995.

[12] M. Padberg and M. R. Rao, "Odd minimum cut-sets and $b$-matchings," *Math. Oper. Res.*, vol. 7, pp. 67–80, 1982.

[13] M. Grötschel and O. Holland, "Solving matching problems with linear programming," *Math. Prog.*, vol. 33, pp. 243–259, 1985.

[14] B. C. Huang and T. Jebara, "Fast b-matching via sufficient selection belief propagation," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 361–369.

[15] J. Mestre, "Greedy in approximation algorithms," in *Algorithms–ESA 2006*. Springer, 2006, pp. 528–539.

[16] D. E. Drake and S. Hougardy, "A simple approximation algorithm for the weighted matching problem," *Information Processing Letters*, vol. 85, no. 4, pp. 211–213, 2003.

[17] G. De Francisci Morales, A. Gionis, and M. Sozio, "Social content matching in Mapreduce," *Proceedings of the VLDB Endowment*, vol. 4, no. 7, pp. 460–469, 2011.

[18] C. Koufogiannakis and N. E. Young, "Distributed algorithms for covering, packing and maximum weighted matching," *Distributed Computing*, vol. 24, no. 1, pp. 45–63, 2011.

[19] F. M. Manshadi, B. Awerbuch, R. Gemulla, R. Khandekar, J. Mestre, and M. Sozio, "A distributed algorithm for large-scale generalized matching," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 613–624, 2013.

[20] A. Khan, A. Pothen, M. Patwary, N. Satish, N. Sundaram, and P. Dubey, "Efficient approximation algorithms for weighted $b$-matching," *SIAM Journal on Scientific Computing*, p. 25, 2016, to appear.

[21] C. Koufogiannakis and N. E. Young, "Distributed fractional packing and maximum weighted b-matching via tail-recursive duality," in *Distributed Computing*. Springer, 2009, pp. 221–238.

[22] G. Georgiadis and M. Papatriantafilou, "Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists," *Algorithms*, vol. 6, no. 4, pp. 824–856, 2013.

[23] Z. Lotker, B. Patt-Shamir, and A. Rosén, "Distributed approximate matching," *SIAM Journal on Computing*, vol. 39, no. 2, pp. 445–460, 2009.

[24] M. Wattenhofer and R. Wattenhofer, "Distributed weighted matching," in *International Symposium on Distributed Computing*. Springer, 2004, pp. 335–348.

[25] J.-H. Hoepman, "Simple distributed weighted matchings," *arXiv preprint cs/0410047*, 2004.

[26] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs," in *Symposium on Theoretical Aspects of Computer Science (STACS)*. Springer, 1998, pp. 259–269.

[27] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *The Seventh International Conference on Parallel Processing and Applied Mathematics*, 2007, pp. 708–717.

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

[29] F. Manne and M. Mjelde, "A self-stabilizing weighted matching algorithm," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2007, pp. 383–393.

[30] Z. Lotker, B. Patt-Shamir, and S. Pettie, "Improved distributed approximate matching," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2008, pp. 129–136.

[31] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," in *Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 2012, p. 10.

[32] F. Manne, M. Naim, and M. Halappanavar, "On stable marriages and greedy matchings," in *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, 2016, p. 8 pp., to appear.

[33] B. Pittel, "On a random instance of a stable roommates problem: Likely behavior of the proposal algorithm," *Combinatorics, Probabiltiy and Computing*, vol. 2, no. 1, pp. 53–92, 1993.

[34] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[35] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[36] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web (WWW 2014)*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.

[37] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group*, 2010.

[38] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *CTWatch Quarterly*, vol. 2, no. 4B, pp. 41–51, 2006.