Computer Science Technical Report

Approximating Weighted Matchings Using the Partitioned Global Address Space Model

Alicia Thorsen, Phillip Merkey

Michigan Technological University
Computer Science Technical Report
CS-TR-10-02
April, 2010

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Contents

List of Algorithms

Abstract

Even though the maximum weighted matching problem can be solved in polynomial time, there are many real world problems for which this is still too costly. As a result, there is a need for fast approximation algorithms for this problem. Manne and Bisseling presented a parallel greedy algorithm which is well suited to distributed memory computers. This paper discusses the implementation of the Manne-Bisseling algorithm using the partitioned global address space (PGAS) model. The PGAS model is a union of the shared address space and distributed models which aims to provide the programming conveniences and performance benefits of these models. Four variations of the algorithm are presented which explore the design constructs available in the PGAS model. The algorithms are expressed using a generic PGAS description, however the experiments and empirical performance numbers were obtained from an implementation written in Unified Parallel C with atomic memory operation extensions.

1 Introduction

Let $G = (V, E)$ be an undirected graph with vertices V and edges E. A *matching* M in a graph G is a subset of edges such that no two edges in M are incident to the same vertex. If G is a weighted graph with edge weights $w : E \to \mathbb{R}^+$, the *weight of the matching* is defined as $w(M) := \sum_{e \in M} w(e)$.

The goal of the MWM problem is to find a matching which maximizes the total weight of the edges in M. Edges in M are considered *matched* while the remainder are *free* or *unmatched*. Vertices incident to matched edges are also considered matched and vertices incident to free edges are considered free.

In 1965, Edmonds presented the first known exact polynomial time algorithm for the MWM problem. It runs in $O(n^2 m)$ [5], where n represents the number of vertices in a graph and m is the number of edges. Since then, Gabow has developed the fastest known implementation which runs in $O(nm + n^2 \log n)$ [6]. Although considerable work has been done on the serial MWM problem, there is much left in the area of parallel algorithms. It is still an open problem to find a MWM using a polylogarithmic time parallel algorithm with polynomially many processors [9].

Even though the serial exact algorithm for MWM is polynomial, there is still a need for fast and simple approximation algorithms. The quality of an approximation algorithm is measured by its *approximation ratio* which is a lower bound on the quality of the solutions the algorithm produces. The greedy algorithm is by far the simplest approximation algorithm for the MWM problem. First, the edges in the graph are sorted by weight, then the heaviest edge e is removed and added to the matching. All edges adjacent to e are then discarded since they are no longer candidates for the matching. This process continues until the graph is empty. The algorithm is guaranteed to find a matching with a weight at least $\frac{1}{2}$ of the optimal solution. The running time is $O(m \log n)$ since the edges of G need to be sorted [4].

Since the running time of the greedy algorithm is dominated by sorting, Preis [13] developed a variant called LAM which eliminates sorting the edges. Instead of choosing the heaviest edge, a *locally heaviest edge* is chosen and added to the matching. A *locally heaviest edge* or *dominating edge* is one which has a weight greater than all of its neighbors. Just like the greedy algorithm this process continues until there are no more edges left. The approximation ratio for this algorithm is also $\frac{1}{2}$ and the running time is $O(m)$ since the locally heaviest edge can be found in amortized constant time [4].

Hoepman [8] developed a linear distributed protocol derived from the LAM algorithm. It assigns one processor to each vertex of the graph and by passing messages to neighboring processors, determines which two processors are adjacent to a dominating edge. Manne and Bisseling [12] showed how this protocol could be used in an efficient parallel matching algorithm that is suitable for distributed memory computers. They developed an MPI implementation which scales well on both complete and sparse graphs. This paper presents four implementation variants of the Manne-Bisseling algorithm using the partitioned global address space (PGAS) model. It explores the distributed and shared address space constructs offered by the PGAS model and compares the usability and performance implications of them.

2 Parallel Greedy Matching

The parallel version of the greedy algorithm uses the concept of *locally dominating edges* to concurrently add edges to the matching. A locally dominating edge is an edge which is heavier than all of its neighbors. At the beginning of each iteration, the sequential greedy matching algorithm selects the heaviest available edge, so each edge added to the matching is locally dominant in the iteration it is chosen. Since locally dominant edges are non-adjacent, they can be easily found in parallel, added to the matching, then removed from the graph along with their neighbors. This continues until all edges have been removed from the graph. Algorithm 1 gives an outline of this procedure.

Algorithm 1: Parallel Greedy Matching

Input: $G = (V, E, w)$
Result: M contains a matching

$M \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    let D be the set of locally dominating edges in E
    add D to M
    let N be the edges adjacent to D
    remove D and N from E
end

3 Luby's MIS Algorithm

The parallel greedy matching algorithm given is similar to Luby's maximal independent set algorithm (MIS) [11] run on the edges of the graph instead of the vertices. Luby's MIS algorithm begins by assigning a random value to each vertex in the graph then selects those vertices which have a value larger than all of its neighbors. The chosen vertices are added to the MIS, then removed along with its neighbors from the graph. This process continues until there are no more vertices remaining in the graph. On average, Luby's algorithm converges after $O(\log n)$ rounds if the weights on the vertices are randomly generated and assigned [7, 11]. Algorithm 2 gives an overview of this procedure.

Algorithm 2: Luby's MIS Algorithm

Input: $G = (V, E)$
Result: I contains a maximal independent set

$I \leftarrow \emptyset$
while $V \neq \emptyset$ **do**
  assign a random value to each $u \in V$
  barrier
  let D be the set of locally dominating vertices in V
  add D to I
  let N be the vertices adjacent to D
  remove D and N from V
  barrier
end

To find an approximated weighted matching of a graph G, we can run Luby's MIS algorithm on the *line graph* of G [12]. A line graph $L(G)$ is a graph which describes the adjacencies of the edges in G. Each vertex in $L(G)$ represents an edge in G and the vertex is labeled using the weight of the edge in G. Vertices in $L(G)$ are adjacent if their corresponding edges in G are also adjacent. The edges of $L(G)$ are therefore unweighted.

The main difference between the parallel greedy matching algorithm and Luby's MIS algorithm is that Luby's algorithm regenerates random values for the vertices at the beginning of each iteration [7]. In $L(G)$, the vertex labels do not change since they represent the weights of the edges of G.

4 Partitioned Global Address Space Model

Implementing the parallel greedy matching algorithm on a distributed or shared memory platform is not a trivial task. Some of the details which need to be considered are communication, synchronization and data consistency. In order to exploit the advantages of both the shared memory and distributed platforms, the implementation details are presented using the partitioned global address space (PGAS) model.

The PGAS model is a union of the shared memory and distributed models which aims to provide the programming conveniences of the shared memory model alongside the performance benefits of the distributed model. Like the shared address space model, PGAS languages have a global space to which all processors can read and write. This space is also logically partitioned so that a portion of it is local to each process. This allows a programmer to exploit memory locality by placing data close to the processes that manipulate it.

Each process has a private address space in addition to *affinity* to a portion of the shared address space. Data objects in the shared address space are visible to all processes, however latency is reduced for objects in a process's portion. Current PGAS languages follow the single program multiple data (SPMD) execution model. Examples of these languages include Unified Parallel C (UPC) [14], Co-Array Fortran [15] and X10 [2]. Figure 1 shows an example of a shared array in the PGAS model.

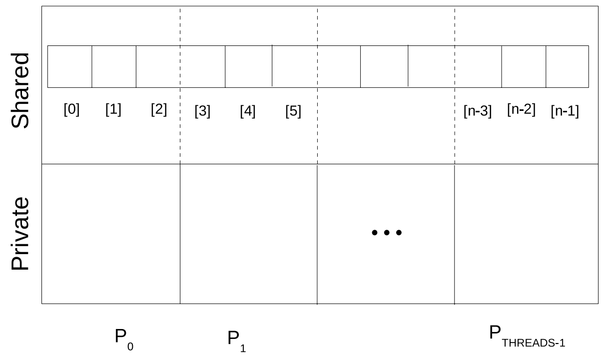

Figure 1: **PGAS Memory View.** Example of a shared array of n elements distributed across *THREADS* processes with a blocking factor of 3.

4.1 PGAS Notation

PGAS languages, such as UPC, provide programming constructs for denoting shared and private variables, data partitioning and affinity. In the pseudocode presented, all variables are declared as *shared*, *local* or *private*. Shared variables are visible to all processes, local variables are shared variables with affinity to a particular process, and private variables are visible only to the owning process. Data partitioning is also accomplished easily in the PGAS model. Given a shared array A, the elements with affinity to the specified process will be denoted as myA. If the affinity of a data element is not known, the function $owner(x)$ returns the id of the owning process.

5 Parallel Implementations

We will now present four different ways to implement the parallel greedy matching algorithm given in Algorithm 1 using the PGAS model. The main differences in these implementations are *synchronous* vs. *asynchronous* execution, and *notification* vs. *polling* based work scheduling.

Here is a brief overview of the notations used in the pseudocode. Given a weighted graph $G = (V, E, w)$, let N_u be the set of vertices that are neighbors of u. Let C_u, the set of candidate vertices of u, be the unmatched vertices in N_u. Let u's *mate* be the endpoint of the heaviest unmatched edge incident to u. This endpoint is denoted as $mate_u$ and is computed by the function $h(C_u)$. If u has chosen v as its mate then u wants to *match* with v. If two neighboring edges have the same weight, ties are broken using the vertex id, i.e. if $w(u, v) = w(v, x)$, the edge (u, v) would be considered heavier if $u > x$. Tie-breaking is only needed when two edges share a common endpoint.

5.1 sync-poll

In *sync-poll* each process begins by computing the mate values for its vertices then synchronizes with the other processes. Next each process revisits the vertices in its subgraph to find the vertices that are incident to dominating edges. For each vertex u, if its mate v chose u as a mate, then u and v are endpoints of a dominating edge.

The dominating edges found are added to the matching and their vertices are removed from the graph. Any neighboring edges are also removed from the graph. Since the graph changes at this point, edges which were previously dominated are now locally dominant. To locate these edges, the processes revisit the remaining vertices in their subgraphs. Once the new dominating edges are found they are added to the matching and the procedure repeats until the graph is empty. Algorithm 3 provides a detailed description of this procedure. Of the four variants presented, *sync-poll* is closest in structure to the traditional shared memory implementation of Luby's MIS algorithm.

5.2 async-poll

Since the weights of the edges do not change like the vertex values in Luby's MIS algorithm, it is not necessary to synchronize the processes before they revisit their subgraphs to search for dominating edges. Once a vertex has chosen a mate, it does not change its selection until the mate is matched with another vertex. Therefore, a process can continuously poll the mates of it vertices to check their statuses and update its vertices accordingly. After the initialization step, this procedure has no dependencies and can therefore be completely asynchronous.

In the *async-poll* variation, a process stays active until all of its vertices are either matched or have no candidates left. As processes make updates to their vertices, these changes allow other processes to make progress. In a non-empty graph, it is guaranteed that at least one edge will be dominating, which prevents this algorithm from deadlocking. Algorithm 4 describes *async-poll* in further detail.

Algorithm 3: sync-poll

Input: $G = (V, E, w)$
Result: Vertices marked as "matched" constitute a matching

shared G, $mate_v$, $matched_v$, S
local myV, C_u, N_u, $mate_u$, $matched_u$
private u, v

foreach $u \in myV$ **do**
    $C_u \leftarrow N_u$
    $matched_u \leftarrow false$
    $mate_u \leftarrow h(C_u)$
barrier
repeat
    $S \leftarrow \emptyset$
    foreach $u \in myV$ **do**
        $v \leftarrow mate_u$
        if $v = null$ **then**
            $myV \leftarrow myV \setminus \{u\}$
        else
            if $mate_v = u$ **then**
                $matched_u \leftarrow true$
                $myV \leftarrow myV \setminus \{u\}$
                if $owner(v) = myID$ **then**
                    $matched_v \leftarrow true$
                    $myV \leftarrow myV \setminus \{v\}$
    barrier
    foreach $u \in myV$ **do**
        $mate_u \leftarrow h(C_u)$
    $S \leftarrow S \cup myV$
    barrier
until $S = \emptyset$

5.3 sync-notify

In both of the poll variations presented, a process needs to repeatedly check the graph to find work. Another approach would be to notify a process when it has work to do. If a vertex u is matched, then any neighbor x which wants to match with it needs to be updated. If u and x are assigned to different processes then the process that owns x needs to be notified to update it. Similarly, if a vertex u changes its mate and is matched with a remote neighbor v, then v needs to be notified of the match. In the synchronous version of this

Algorithm 4: async-poll

Input: $G = (V, E, w)$
Result: Vertices marked as "matched" constitute a matching

shared G, $mate_v$, $matched_v$
local myV, C_u, N_u, $mate_u$, $matched_u$
private u, v

```
foreach u ∈ myV do
    C_u ← N_u
    matched_u ← false
    mate_u ← h(C_u)
barrier
while myV ≠ ∅ do
    foreach u ∈ myV do
        mate_u ← h(C_u)
        v ← mate_u
        if v = null then
            myV ← myV \ {u}
        else
            if mate_v = u then
                matched_u ← true
                myV ← myV \ {u}
                if owner(v) = myID then
                    matched_v ← true
                    myV ← myV \ {v}
```

algorithm, notifications are aggregated locally then written to shared memory in one bulk step. Algorithm 5 describes this procedure in further detail.

Notifications for a process are posted to a shared array which acts like a queue. The entries in the queue are vertices which the process needs to update. Each process maintains a shared variable *count* that reflects the number of items currently in its queue. To post notifications to a process' queue, *count* is first incremented using the atomic memory operation *fetch-and-add*, then the values are written in the reserved space. Since the processes synchronize between the reading and writing steps, a process is guaranteed that no other process is posting while it is reading.

5.4 async-notify

In this version, notifications are not aggregated locally, but posted right away using atomic memory operations. This means that processes read and write notifications concurrently. After the initialization step, this variation is completely asynchronous. As in *sync-notify*, processes post notifications by first increasing *count* for the queue. The owning process reads items at the front of the queue and marks them as -1 to prevent re-reading later. The owner also needs to keep track of the last location that was read because this marks the front of the queue. Since posting processes increase *count* before actually writing their values, it is possible for the reading process to consume values which have not already been written. To account for this, the queue is initialized to -1 so the reader can detect unwritten values within the bounds of *count*.

Since the front of the queue is constantly incremented, and processes add items at the end, it is possible for the queue to run out of space. To prevent this, the owning process periodically resets the front of the queue. This reset operation cannot be performed while another process is writing. To accomplish this, the owning process resets the queue after reading when the value of *count* is the same as before the read. This indicates that no process has altered the queue while the owner was reading. The reset is done using the atomic memory operation *compare-and-swap*. Algorithm 6 describes this procedure in further detail.

6 Experiments

The four variations discussed were implemented using Unified Parallel C (UPC) [14] on a Cray X1 and Cray XT4. The Cray X1 is a shared memory vector processor supercomputer and the Cray XT4 is a distributed memory massively parallel supercomputer. UPC is a parallel extension of ANSI C for partitioned global address space programming and is supported on almost every parallel architecture available today. The algorithms presented rely on atomic memory operations (AMO), however UPC does not offer AMOs as a language feature. They are available on the Cray X1 as intrinsic procedures, and on the Cray XT4 as part of the Berkeley implementation of UPC [1].

Each variation of the algorithm was run on 3 graphs. The first graph is a grid of size 256x256. It has $65,536$ vertices and $130,560$ edges. The grid graph was chosen because it represents a graph with high locality. Each vertex can have at most 4 neighbors which are most likely within the same subgraph after partitioning. This means that remote references are less likely to occur.

The second graph is a complete bipartite graph with $2,048$ vertices and $130,560$ edges. The graph is partitioned so that each edge is a *crossing edge*. A crossing edge is one whose endpoints do not belong to the same subgraph. The bipartite graph was chosen because it contains no locality, which means that whenever a process examines the desired mate of one of its vertices, it will perform a remote reference.

The third graph is a sparse graph called *crankseg* that was generated from real-world data. The graph has 52,804 vertices and 5,280,703 edges and can be found in the University of Florida Sparse Matrix Collection [3]. This graph represents the middle-ground test case since it will produce both remote and local references. Since the structure of the grid and bipartite graphs are known beforehand, they are partitioned by hand. The sparse graph, however, is partitioned using the Metis Graph Partitioning library [10]. Figures 2 and 3 show the results generated from both machines.

7 Conclusion

7.1 Machine Comparison

Overall all four variations of the algorithm perform better on the Cray X1 than the Cray XT4, which is not surprising. Each variation is fine-grained with poor locality and is better suited to a supercomputer like the X1. The X1 is a real shared memory machine which uses loads and stores and therefore has finer granularity and lower latency than the XT4. The XT4 favors algorithms that require conventional serial performance and generate more coarse-grained memory traffic.

Comparing the runtime of each variation on both machines is not as straightforward since it involves considering the latency, bandwidth and granularity of remote references on each machine. However it is fair to compare how each variation scales on both machines.

The grid graph represents the best case scenario for the algorithm since there is a lot of locality. There is some speedup on both machines but each variation scales better on the X1. This test case shows that even with a low number of remote references, the X1 is still more amenable to this type of algorithm.

The bipartite graph represents the worst case since all edges are crossing edges. Even though there is some speedup on the X1 it is not ideal. On the XT4 each variation degrades in performance as more processes are added. This graph demonstrates how the XT4 suffers from excessive fine-grained remote references since it cannot take advantage of local memory latency.

The sparse graph, like the grid graph, scales well on the X1. However, like the bipartite graph, it is also degrades on the XT4. This is to be expected as the sparse graph produces more remote references than the grid graph.

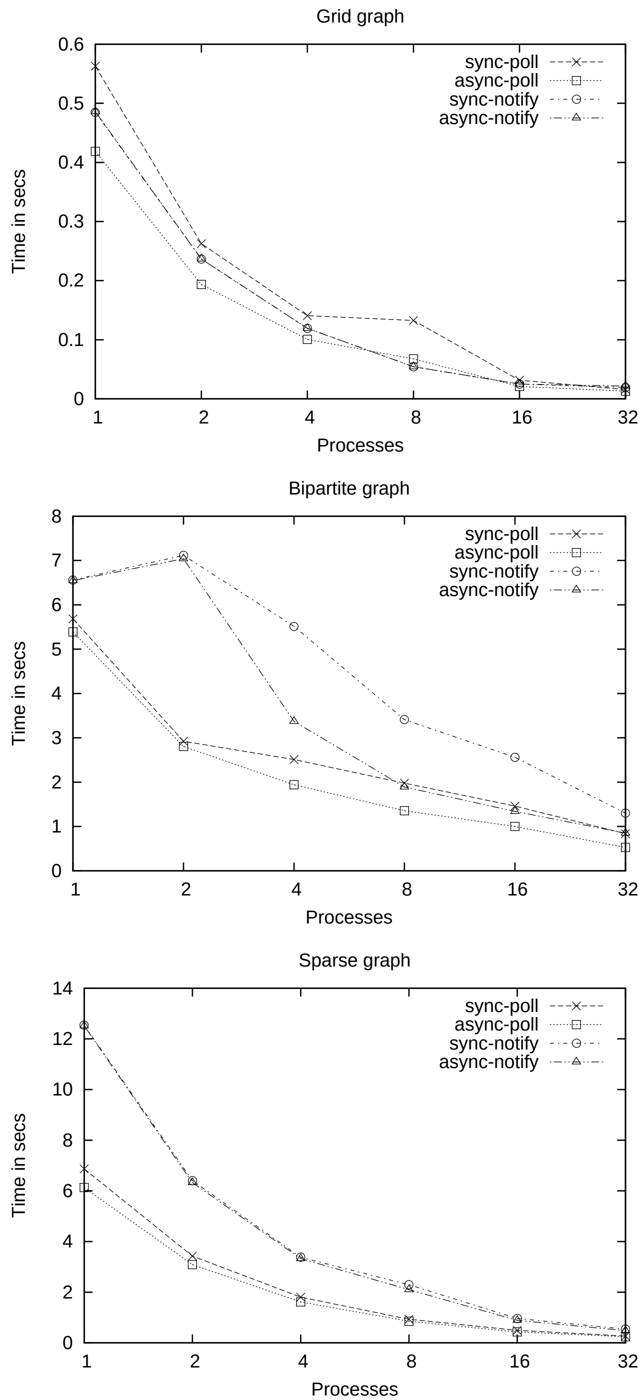

Figure 2: Running time on the Cray X1

7.2 Algorithm Comparison

Overall, the poll versions perform better than the notify versions on both architectures. The poll versions are implemented in a straightforward manner in UPC and the resulting code

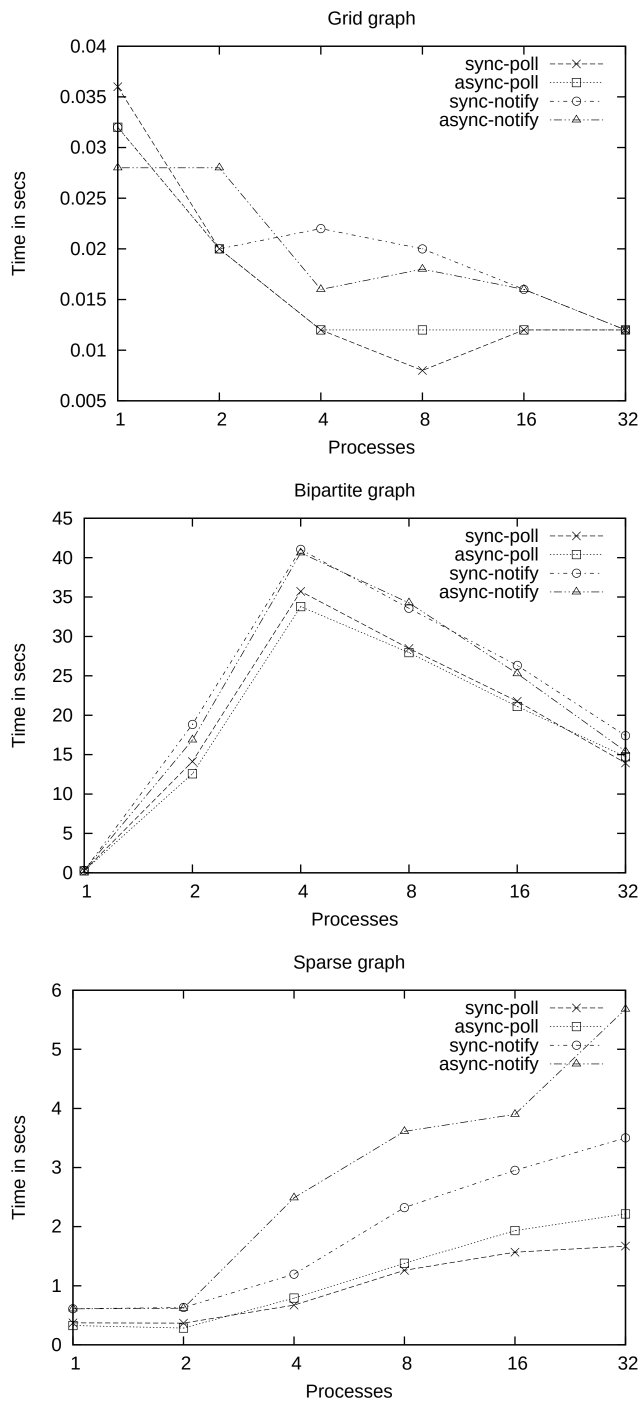

Figure 3: Running time on the Cray XT4

looks very similar to the pseudocode presented in this paper. The notify versions involve more parallel overhead to prepare notifications, communicate, and maintain the queues. In the poll versions, the polling process bears the burden of finding work. However, in the notify versions, this work is transferred to the notifying thread. From the results presented, it seems that the polling versions are more efficient on these architectures when using the PGAS model.

8 Future Work

In the future, an investigation will be conducted to compare the notify versions to equivalent MPI implementations, to compare performance and determine what optimizations each version takes advantage of. The algorithms will also be tested on a wider array of graphs to observe how different characteristics affect running time.

Acknowledgment

We would like to thank Brad Chamberlain at Cray, Inc. for providing access to the Cray X1 for testing, and Dr. Fredrik Manne at the University of Bergen, Norway for access to the Cray XT4.

References

[1] Berkeley UPC Compiler. http://upc.lbl.gov.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538, 2005.

[3] T. Davis. The University of Florida Sparse Matrix Collection. In *ACM Transactions on Mathematical Software*. http://www.cise.ufl.edu/research/sparse/matrices/.

[4] D. E. Drake and S. Hougardy. Linear time local improvements for weighted matchings in graphs. In *Experimental and Efficient Algorithms: 2nd Int. Workshop on Efficient Algorithms*, volume 2647 of *Lecture Notes in Computer Science*, page 622, 2003.

[5] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[6] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, 1976.

[7] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, 2nd ed.* Addison-Wesley, 2003.

[8] J.-H. Hoepman. Simple distributed weighted matchings, 2004. eprint cs.DC/0410047.

[9] S. Hougardy and D. E. D. Vinkemeier. Approximating weighted matchings in parallel. *Information Processing Letters*, 99(3):119–123, 2006.

[10] G. Karypis and V. Kumar. Metis, a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0, 1998.

[11] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, 1985.

[12] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *7th Int. Conf. on Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, 2007.

[13] R. Preis. Linear time $\frac{1}{2}$ - approximation algorithm for maximum weighted matching in general graphs. In *Symp. on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 259–269, 1999.

[14] UPC Consortium. UPC Language Specifications, v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.

[15] A. Wallcraft. Official Co-Array Fortran website. http://www.co-array.org.

Algorithm 5: sync-notify

```
Input: G = (V, E, w)
Result: Vertices marked as "matched" constitute a matching

shared G, mate_v, matched_v, S
local myV, C_u, N_u, mate_u, matched_u
private u, v, D
foreach u ∈ myV do
    C_u ← N_u
    mate_u ← h(C_u)
    matched_u ← false
barrier
repeat
    S ← ∅
    while myV ≠ ∅ do
        let u ∈ myV
        mate_u ← h(C_u)
        v ← mate_u
        if v ≠ null and mate_v = u then
            matched_u ← true
            myV ← myV ∪ update(u, C_u \ {v})
            if owner(v) = myID then
                matched_v ← true
                myV ← myV ∪ update(v, C_v \ {u})
            else
                if matched_v = false then
                    prepare notification for owner(v) about v
        myV ← myV \ {u}
    post all prepared notifications
    barrier
    read all notifications and append to myV
    barrier
    S ← S ∪ myV
until S = ∅

function update(u, C_u) :
begin
    D ← ∅
    foreach x ∈ C_u s.t. mate_x = u do
        if owner(x) = myID then
            D ← D ∪ x
        else
            prepare notification for owner(x) about x
    return D
```

Algorithm 6: async-notify

```
Input: G = (V, E, w)
Result: Vertices marked as "matched" constitute a matching

shared G, mate_v, matched_v
local myV, C_u, N_u, mate_u, matched_u
private u, v, D
foreach u ∈ myV do
    C_u ← N_u
    mate_u ← h(C_u)
    matched_u ← false
barrier
repeat
    while myV ≠ ∅ do
        let u ∈ myV
        mate_u ← h(C_u)
        v ← mate_u
        if v = null then
            V ← V \ u
        else
            if mate_v = u then
                matched_u ← true
                myV ← myV ∪ update(u, C_u \ {v})
                V ← V \ u
                if owner(v) = myID then
                    matched_v ← true
                    myV ← myV ∪ update(v, C_v \ {u})
                    V ← V \ v
                else
                    if matched_v = false then
                        notify owner(v) about v
        myV ← myV \ {u}
    read all notifications and append to myV
until V = ∅

function update(u, C_u) :
begin
    D ← ∅
    foreach x ∈ C_u s.t. mate_x = u do
        if owner(x) = myID then
            D ← D ∪ x
        else
            notify owner(x) about x
    return D
```