Maximum Weighted Matching Using the Partitioned Global Address Space Model

Alicia Thorsen, Phillip Merkey
Computer Science Department
Michigan Technological University, USA
{athorsen, merk}@mtu.edu

Fredrik Manne
Department of Informatics
University of Bergen, Norway
fredrik.manne@ii.uib.no

Keywords: Maximum weighted matching, partitioned global address space, PGAS, Unified Parallel C, UPC

Abstract

Efficient parallel algorithms for problems such as maximum weighted matching are central to many areas of combinatorial scientific computing. Manne and Bisseling [13] presented a parallel approximation algorithm which is well suited to distributed memory computers. This algorithm is based on a distributed protocol due to Hoepman [9]. In the current paper, a partitioned global address space (PGAS) implementation is presented.

PGAS programmers have the conveniences of using a shared memory model, which provides implicit communication between processes using normal loads and stores. Since the shared memory is partitioned according to the affinity of a process, one is also able to exploit data locality.

This paper addresses the main differences between the PGAS and MPI implementations of the Manne-Bisseling algorithm. It highlights some advantages of using the PGAS model such as shorter, simpler code, similarity to the sequential algorithm, and options for fine-grained and coarse-grained communication.

1. INTRODUCTION

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

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

The first exact polynomial time algorithm for this problem was given by Edmonds in 1965 and runs in $O(n^2m)$ [7], where n and m are the number of vertices and edges in a graph, respectively. Since then, much work has been done to improve the worst case running time of this algorithm. The fastest known result is due to Gabow who reduced it to $O(nm + n^2 \log n)$ [8]. In the area of parallel algorithms, it is still an open problem to find a maximum weighted matching using an *NC* algorithm [10]. *NC* is the class of problems that are computable in polylogarithmic time with polynomially many processors.

One of the simplest approximation algorithms for the weighted matching problem is the greedy algorithm. It works by iteratively removing the heaviest edge e and adding it to the matching. All edges adjacent to e are then discarded. This process continues until there are no more edges left. The approximation ratio for this algorithm is $\frac{1}{2}$ and the runtime is $O(m \log n)$ since the edges of G need to be sorted [6].

Preis [14] developed an algorithm called LAM which is based on the greedy approach, but avoids sorting the edges of the graph. At each step the *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. The approximation ratio for this algorithm is also $\frac{1}{2}$, but the runtime is $O(m)$ since the locally heaviest edge can be found in amortized constant time [6].

Hoepman [9] developed a linear distributed protocol which builds on the LAM algorithm. It assigns one processor to each vertex of the graph and after a set of communication rounds, determines which two processors share a dominating edge. Manne and Bisseling [13] showed how this protocol could be used in an efficient parallel matching algorithm which is suitable for distributed memory computers. They developed an MPI implementation which scales well on both complete and sparse graphs.

An implementation of the Manne-Bisseling algorithm is presented using the partitioned global address space (PGAS) model. This paper addresses the main differences between the PGAS and MPI implementations and discusses how the two paradigms affect their respective implementations.

2. PROGRAMMING PARADIGMS

Currently the two most popular parallel computing paradigms are shared address space and message passing. In the shared address space or shared memory paradigm, all processors can read and write to a global space and therefore communicate implicitly. In the message passing paradigm, each processor has its own private memory and exchange of data and synchronization information is done explicitly through messages.

The message passing paradigm works best for problems in which the data can be easily partitioned and the computation is well structured. Graph computations on sparse graphs are usually "data-driven" in the sense that the structure of the graph dictates the order of the computations [12]. This structure is highly irregular and therefore difficult to express in code. As a result, many graph problems are difficult to implement using message passing because the explicit communication required to model these access patterns is non-intuitive [12].

The shared address space paradigm, however, is ideal for irregular computations due to the implicit communication. Programming is also simplified as the programmer can focus more on the algorithm than the communication. Unfortunately, all large scale shared address space platforms have non-uniform memory access (NUMA) which can lead to poor performance from remote memory accesses.

2.1. PGAS Paradigm

The partitioned global address space paradigm aims to address the problems associated with the shared memory model while still maintaining its conveniences [5]. 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 partition. Current PGAS languages follow the single program multiple data (SPMD) execution model. Examples of these languages include Unified Parallel C (UPC) [2], Co-Array Fortran [15] and X10 [4].

PGAS languages, such as UPC, provide programming constructs for denoting shared and private variables, data partitioning, affinity and synchronization. Instead of focusing on the syntax of one particular language, some conventions are adopted for representing these constructs in the pseudocode.

2.1.1. Shared and Private Variables

Shared variables reside in the shared address space and are visible to all processes. There is only one copy of each shared variable. In this paper, *local variables* are shared variables with affinity to a particular process. Private variables reside in a process's private memory and are only visible to the owning process. Each process has its own copy of each private variable. In order to clarify locality, all variables are declared as shared, local or private in the pseudocode presented.

2.1.2. Data Partitioning

Shared arrays are distributed round-robin across all processes one block of elements at a time. Given a shared array $A[n]$ with a block size of 1 distributed over k processes, each process p has affinity to elements $A[p]$, $A[k+p]$, $A[2k+p]$, $A[3k+p]$ and so on. If the block size is b then process p has affinity to b contiguous elements starting at element $A[pb]$. The size of the block is specified in the declaration of the array. Unlike message passing, the array indices do not change after partitioning because the global ordering is still maintained. Figure 1 shows an example of this.

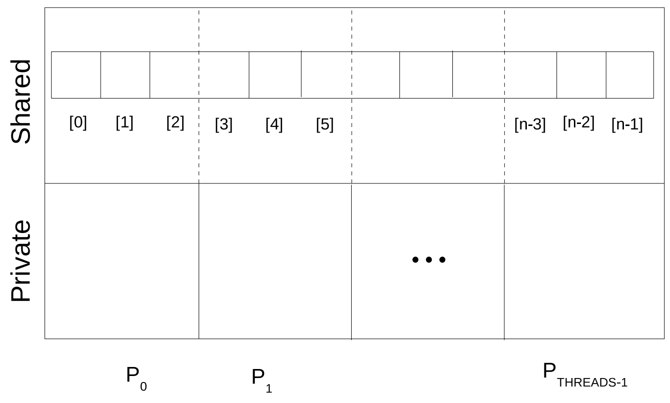

Figure 1. PGAS Memory View. Example of a shared array of n elements distributed across `THREADS` processes with a blocking factor $b = 3$.

2.1.3. Affinity

Once an array has been partitioned, it is easy to calculate the affinity of an element based on the block size. Given a shared array A, the set of elements with affinity to a process is denoted as myA. If the block size is unknown, for example when dereferencing a pointer, the programmer can call a function which returns the id of the owning process. This function is denoted as $owner(x)$ in the pseudocode.

2.1.4. Synchronization

Synchronization constructs such as barriers ensure that all processes reach a certain point in the code before any of them continue execution. This is denoted by the **barrier** statement in the pseudocode.

3. SEQUENTIAL ALGORITHM

The sequential matching algorithm given by Manne and Bisseling [13] will be presented first. Given a weighted graph $G = (V, E, w)$, let N_v be the set of vertices that are neighbors of v. Let C_v, the set of candidate vertices of v, be the unmatched vertices in N_v. Let v's *mate* be the endpoint of the heaviest unmatched edge incident to v. This endpoint is denoted as $mate_v$ and is computed by the function $h(C_v)$. If v has chosen u as its mate then v wants to *match* with u. 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.

The sequential algorithm is composed of two stages. In the first stage, the initial set of dominating edges is found and added to the matching. For each vertex v, the algorithm computes its mate u and checks to see if u has chosen v as a mate. A match is made if the two vertices mutually choose each other. The edge (u,v) is added to the matching M and the vertices $\{u,v\}$ are added to the set D. This set D contains matched vertices which are used later on to increase the matching.

Since each vertex v in D is no longer a candidate for matching, any vertex which wants to match with v needs a new mate. In the second stage, the algorithm removes each vertex v from D one at a time and looks at each unmatched neighbor x. If x wants to match with v, the algorithm updates the candidate list C_x and finds the next best available mate, y. If y wants to match with x then (x,y) is added to M and the vertices $\{x,y\}$ are added to D. The algorithm terminates when $D = \emptyset$. If there is an edge which can extend the current matching it will be found before D is exhausted [13]. Algorithm 1 provides further details.

Algorithm 1 Sequential Matching Algorithm [13]

```
Require: $G = (V,E,w)$
Ensure: $M$ contains the edges in the computed matching
 1: for all $v \in V$ do
 2:     $mate_v \leftarrow null$
 3:     $C_v \leftarrow N_v$
 4: end for
 5: $D \leftarrow \emptyset$
 6: $M \leftarrow \emptyset$
 7: for all $v \in V$ do
 8:     $mate_v \leftarrow h(C_v)$
 9:     $u \leftarrow mate_v$
10:     if $mate_u = v$ then
11:         $D \leftarrow D \cup \{u,v\}$
12:         $M \leftarrow M \cup \{(u,v)\}$
13:     end if
14: end for
15: while $D \neq \emptyset$ do
16:     $v \leftarrow$ some vertex from $D$
17:     $D \leftarrow D \setminus \{v\}$
18:     for all $x \in C_v$ where $mate_x = v$ and $(v,x) \notin M$ do
19:         $C_x \leftarrow C_x \setminus \{v\}$
20:         $mate_x \leftarrow h(C_x)$
21:         $y \leftarrow mate_x$
22:         if $mate_y = x$ then
23:             $D \leftarrow D \cup \{x,y\}$
24:             $M \leftarrow M \cup \{(x,y)\}$
25:         end if
26:     end for
27: end while
28: return $M$
```

4. PARALLEL ALGORITHM

The *owner-computes* model is used in the parallel version of this algorithm. The vertices of the graph are partitioned and each process runs the sequential algorithm on its allocated vertices. The graph can be partitioned naïvely using the relative numbering, or with a graph partitioning library such as Metis [11] which minimizes the number of crossing edges. In the parallel version, the matching is defined by the mate value of the vertices marked as matched. The algorithm is composed of two stages corresponding to the sequential algorithm.

In the first stage, each process finds the dominating edges incident to its allocated vertices. It iterates over the set of vertices in its subgraph and computes their desired mates. Next all processes synchronize to ensure all mates are precomputed before matches are made.

Then each process iterates over each vertex v in its subgraph and marks it as matched if its desired mate u also chose v. This match is made even if u belongs to another process. In this case the owner of v marks v as matched and the owner of u marks u as matched. Each vertex that a process marks as matched is added to its private set D. Since processes only match the vertices in their subgraphs, the vertices in D are always local.

The second stage proceeds as in the sequential algorithm with the limitation that a process cannot update the candidate list and mate value of a remote vertex. A process p removes a vertex v in its D, by considering each unmatched neighbor x of v where x wants to match with v. If x is owned by $q \neq p$, p instructs q to find a new mate for x. However, if x is local to p, p updates the candidate list C_x and finds the next best mate, y. If y wants to match with x and y is also local to p, the match is made as in the serial case. If however, y is owned by $q \neq p$, p instructs q to complete the match. Whenever a process marks a vertex as matched the vertex is added to its D. The algorithm terminates when $D = \emptyset$ for all processes.

4.1. MPI Implementation

In the MPI implementation of this algorithm, each process maintains a set of ghost-vertices to handle crossing edges. If a vertex v resides on process p and has a neighbor u which resides on process q, where $p \neq q$, a ghost-vertex v' and the corresponding edge (u,v') will be created on process q.

When the initial mates are computed for each vertex v, this value is propagated to all ghost copies of v. Whenever v chooses a new mate, all ghost copies of v need to be updated. This requires a message from p to all processes that have ghost copies of v. Since all copies of v will always have the same mate value, it is impossible for v to be matched with more than one remote vertex. If two remote vertices x and y, which reside on different processes, want to match with two different ghost copies of v, the mate value of v will either be x,

y or neither. Since it cannot be both x and y at the same time, two conflicting matches cannot be made. For more details on the MPI implementation please refer to [13].

4.2. PGAS Implementation

In the PGAS implementation, the vertices are stored in shared memory in an array. The array is distributed based on a partitioning computed by the Metis graph partitioning library, so that each process has affinity to the vertices in its subgraph. If Metis returns a partitioning with unequal partition sizes, the array is padded with dummy vertices to ensure a uniform block size. Each vertex is represented by a struct which contains a linked list of its neighbors, its desired mate and a flag signaling whether or not it is matched. The data structure is an augmented adjacency list with repeated edges.

The first stage of the PGAS implementation is very similar to the general parallel matching algorithm outlined in section 4. Algorithm 2 gives a detailed description of this stage.

Algorithm 2 PGAS Matching Algorithm Stage 1

```
Require: $G = (V, E, w)$
Ensure: Vertices marked as "matched" constitute a matching
 1: shared $G$, $mate_u$, $matched_u$
 2: local $myV$, $C_v$, $N_v$, $mate_v$, $matched_v$
 3: private $D$, $v$, $u$
 4: $D \leftarrow \emptyset$
 5: for all $v \in myV$ do
 6:     $C_v \leftarrow N_v$
 7:     $mate_v \leftarrow h(C_v)$
 8:     $matched_v \leftarrow false$
 9: end for
10: barrier
11: for all $v \in myV$ do
12:     $u \leftarrow mate_v$
13:     if $mate_u = v$ then
14:         $matched_v \leftarrow true$
15:         $D \leftarrow D \cup \{v\}$
16:         if $u \in myV$ then
17:             $matched_u \leftarrow true$
18:             $D \leftarrow D \cup \{u\}$
19:         end if
20:     end if
21: end for
22: barrier
```

The second stage of the PGAS implementation can be done in either a synchronous or asynchronous manner. In the asynchronous version messages are sent individually using atomic memory operations, while in the synchronous version they are sent in bulk during a communication step. The synchronous version is very much like the MPI implementation. Receiving messages works similarly. In the asynchronous version messages are retrieved when a process exhausts its work, however in the synchronous version messages are received in bulk

Messages for a process p are written to p's portion of a shared array S. Each element in S is a vertex which the owning process needs to update. S is distributed evenly among all processes and $|S| = |V|$. Each portion of S that belongs to a process p acts as a stack for p since vertices are added and removed from the end. Each process p also maintains a shared value $size_p$ that reflects the number of items currently in its stack. To send a message to a process p, $size_p$ is incremented using an atomic memory operation to reserve the space, then the values are written. When p removes an element from the stack, it performs the same actions in reverse. First it copies the item to be removed then decrements the stack counter using an atomic operation. Algorithm 3 describes the synchronous version of stage 2.

4.3. Comparison

This algorithm was implemented in both MPI and UPC, however they differ in several areas.

4.3.1. Similarity to the Sequential Algorithm

One of the main advantages of using the PGAS model is the similarity between the sequential algorithm and its corresponding parallel version. Maintaining this similarity makes it easier for a programmer to develop a parallel implementation, since the programming overhead in parallelizing a sequential algorithm can often be significant. In this paper, the pseudocode for the serial algorithm and the PGAS version is very similar.

One of the drawbacks of the MPI implementation is the need for ghost-vertices and the arithmetic overhead that goes with using them. In the PGAS model, there is no need for ghost-vertices since the entire graph can be placed in shared memory. Of course this means that reading a vertex which is not in a process's subgraph incurs a cost. However from a programmability standpoint, it is preferable to read and write to shared memory in the same manner that private memory is accessed.

4.3.2. Code Length

The length of the UPC implementation is much shorter than the MPI version which is mainly due to the amount of parallel overhead that the MPI programmer must implement. Code length is a software metric that is often used to predict a program's reliability and ease of maintenance [3].

In MPI, distributing the graph, setting up the ghost-vertices, calculating the locations of off-processor vertices and edges, and sending and receiving messages are among the extra work that is left to the programmer. PGAS languages like UPC make it easy to distribute the graph by using a blocking factor when the array is created. There is also very little arithmetic needed to locate vertices since the global

Algorithm 3 PGAS Matching Algorithm Stage 2

```
Require: $G = (V, E, w)$, $D$
Ensure: Vertices marked as "matched" constitute a matching
 1: shared $G$, $S$, $mate_y$, $matched_y$
 2: local $myV$, $C_v$, $C_x$, $mate_v$, $mate_x$, $matched_x$
 3: private $D$, $Q$, $v$, $x$, $y$, $workLeft$
 4: $S \leftarrow \emptyset$
 5: $workLeft \leftarrow$sum of $|D|$ for all processes
 6: while $workLeft > 0$ do
 7:   while $D \neq \emptyset$ do
 8:     $v \leftarrow$ some vertex from $D$
 9:     $D \leftarrow D \setminus \{v\}$
10:     for all $x \in C_v$ such that $mate_x = v$ and $mate_v \neq x$ do
11:       if $x \notin myV$ then
12:         prepare msg for $owner(x)$ to update $x$
13:       else
14:         $C_x \leftarrow C_x \setminus \{v\}$
15:         $mate_x \leftarrow h(C_x)$
16:         $y \leftarrow mate_x$
17:         if $mate_y = x$ then
18:           $matched_x \leftarrow true$
19:           $D \leftarrow D \cup \{x\}$
20:           if $y \in myV$ then
21:             $matched_y \leftarrow true$
22:             $D \leftarrow D \cup \{y\}$
23:           else
24:             if $matched_y = false$ then
25:               prepare msg for $owner(y)$ to update $y$
26:             end if
27:           end if
28:         end if
29:       end if
30:     end for
31:   end while
32:   write all msgs to $S$
33:   barrier
34:   copy my msgs to $Q$
35:   update vertices in $Q$ and add matched vertices to $D$
36:   barrier
37:   $workLeft \leftarrow$sum of $|D|$ for all processes
38: end while
```

numbering of vertices is maintained. It is also easier for a process to determine which vertices are allocated to it since an $owner()$ function is provided.

4.3.3. Implicit Communication

In an MPI program, communication is done through sends and receives, while in a PGAS program it is done through reads and writes to shared memory. This implicit form of communication makes it easier to selectively broadcast values such as when a vertex v has chosen a new mate. When this occurs in the MPI version, a message must be explicitly sent to all processes which have a ghost-copy of v. In the PGAS implementation, this is not necessary since any process can read v's updated information when necessary. On the other hand, since the communication is implicit it is harder to account for non-local memory references which are often equivalent to messages.

4.3.4. Coarse and Fine-Grained Communication

In the synchronous version of the PGAS implementation we use coarse-grained communication, since all messages are aggregated and sent in one step. However in the asynchronous version, we use atomic memory operations to write notifications as soon as they are discovered. This is an example of fine-grained communication. In MPI, programmers typically make their programs as coarse-grained as possible as in the Bulk-Synchronous-Processing (BSP) model [12]. BSP type algorithms separate computation and communication into distinct supersteps to reduce the overall cost of latency. This model is appropriate for well-structured problems, however for graph algorithms this limits the amount of fine-grained parallelism that can be exploited [12].

4.3.5. Atomic Memory Operations

As mentioned earlier, atomic memory operations were used in the UPC implementation to send and receive messages. Adding to the stack involves writing to remote shared memory which incurs a cost. Atomic operations were used to provide concurrent, conflict-free access to the stack. If atomic operations are not available on a particular platform, locks may be used as a substitute. However, locks are not as scalable as atomic operations and were generally avoided.

5. PERFORMANCE

The PGAS implementation was developed in UPC and run on an in-house UPC reference implementation to verify correctness. We are seeking access to a machine with the characteristics necessary for this type of algorithm to make performance measurements. This algorithm requires support for fine-grained memory references with little or no locality.

The performance of this algorithm is largely dictated by the number of remote memory references that are made. A process performs a remote read whenever it updates one of its candidate lists involving non-local vertices, and when it checks the mate value of a remote vertex. Remote writes are done when sending messages to other processes. Both of these situations are affected by the partitioning of the graph, as the number of remote neighbors in a vertex's candidate list determines the amount of remote memory references.

Given a sparse graph which has been partitioned into k sections, the number of crossing edges c can be easily computed. In the worst case, a process will have c remote neighbors in its candidate lists since edges are repeated. The probability that a vertex chooses a remote mate is dependent on c and the average degree of the graph. Manne and Bisseling [13] showed that if the edge weights are assigned randomly, the Hoepman protocol [9] is expected to terminate in $O(log|E|)$ rounds. In the parallel algorithm this corresponds to a process exhausting all the vertices in its D and processing all messages it received.

Given these factors, some performance predictions can be made using a performance model for PGAS languages. This model from Zhang and Seidel [16], uses micro-benchmarks to determine the cost of a remote memory reference and the compiler and runtime optimizations available on a platform. Since both of the DARPA High Productivity Computing Systems (HPCS) [1] petascale machines are being designed with PGAS languages in mind, it is appropriate to evaluate this implementation in a way that is consistent with the scale and capabilities of expected platforms.

6. FUTURE WORK

A tighter upper bound will be determined on the number of remote memory references a process makes in each round. Using this and performance characteristics of anticipated PGAS platforms, a probabilistic performance model of this algorithm will be created. If we are able to secure access to a tightly coupled system, the algorithm will also be evaluated using a wider array of graphs to determine how various graph structures affect its behavior.

REFERENCES

[1] DARPA High Productivity Computing Systems website. http://www.highproductivity.org.

[2] Official UPC website. George Washington University. http://www.gwu.edu/ upc.

[3] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the UPC language. In *Proc. of Parallel and Distributed Processing Symp.*, page 254, 2004.

[4] 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.

[5] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Proc. of the Symp. on Principles and practice of parallel programming*, pages 36–47, 2005.

[6] 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.

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

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

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

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

[11] 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.

[12] A. Lumsdaine, D. Gregor, J. Berry, and B. Hendrickson. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[13] 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.

[14] 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.

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

[16] Z. Zhang and S. Seidel. A performance model for fine-grain accesses in UPC. In *Proc. of the Int. Parallel and Distributed Processing Symp.*, 2006.