A High-Quality and Fast Maximal Independent Set Implementation for GPUs

MARTIN BURTSCHER, SINDHU DEVALE, SAHAR AZIMI, JAYADHARINI JAIGANESH, and EVAN POWERS, Department of Computer Science, Texas State University

Computing a maximal independent set is an important step in many parallel graph algorithms. This article introduces ECL-MIS, a maximal independent set implementation that works well on GPUs. It includes key optimizations to speed up computation, reduce the memory footprint, and increase the set size. Its CUDA implementation requires fewer than 30 kernel statements, runs asynchronously, and produces a deterministic result. It outperforms the maximal independent set implementations of Pannotia, CUSP, and IrGL on each of the 16 tested graphs of various types and sizes. On a Titan X GPU, ECL-MIS is between 3.9 and 100 times faster (11.5 times, on average). ECL-MIS running on the GPU is also faster than the parallel CPU codes Ligra, Ligra+, and PBBS running on 20 Xeon cores, which it outperforms by 4.1 times, on average. At the same time, ECL-MIS produces maximal independent sets that are up to 52% larger (over 10%, on average) compared to these preexisting CPU and GPU implementations. Whereas these codes produce maximal independent sets that are, on average, about 15% smaller than the largest possible such sets, ECL-MIS sets are less than 6% smaller than the maximum independent sets.

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**;

Additional Keywords and Phrases: Maximal independent set, code optimization, GPU, parallel programming

ACM Reference format:
Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. 2018. A High-Quality and Fast Maximal Independent Set Implementation for GPUs. *ACM Trans. Parallel Comput.* 5, 2, Article 8 (December 2018), 27 pages.
https://doi.org/10.1145/3291525

1 INTRODUCTION

An independent set of an undirected graph is a subset of its vertices such that none of the vertices in the set is adjacent to another. The set is maximal if all remaining vertices are adjacent to at least one vertex in the set. Maximal independent sets (MISs) are not necessarily unique. The largest possible such sets are called maximum independent sets and are also not necessarily unique. Finding a maximum independent set is NP-hard in general, which is why MISs are widely used in practice, for which good heuristics exist to compute them quickly. This article focuses on making parallel MIS computations faster and on producing larger sets.

This work was supported by the U.S. National Science Foundation under grant no. 1406304 as well as hardware donations from NVIDIA Corporation.

Authors' addresses: M. Burtscher, S. Devale, S. Azimi, J. Jaiganesh, and E. Powers, Department of Computer Science, Texas State University, 601 University Drive, San Marcos, TX 78666-4684; emails: burtscher@txstate.edu, {sindhu.devale, azimi.bm, jjdharini}@gmail.com, edp30@txstate.edu.

2329-4949/2018/12-ART8 $15.00
https://doi.org/10.1145/3291525

Determining an MIS is a basic building block of many parallel graph algorithms, including graph coloring [23], maximal matching, 2-satisfiability, maximal set packing [20], the odd set cover problem, and coarsening in algebraic multigrid [6]. Importantly, it can be employed to parallelize computations with arbitrary and dynamically changing conflicts using the following recipe. (1) Establish the current conflicts. (2) Record them in a conflict graph (aka interference graph), where the vertices represent the computations and the edges represent the conflicts. (3) Compute an MIS of this graph. (4) Execute the computations in parallel that correspond to the vertices in the MIS and remove those vertices and their edges from the graph. (5) Repeat these steps, collectively called a round, until all computations have been processed. By definition, this approach results in only independent work being executed concurrently. Moreover, it exposes a maximal number of computations to be executed in each round. Note that the alternative—parallelizing such codes using locks—can be expensive owing to the locking overhead whereas the above approach is lock free. Of course, the MIS-based approach is useful only if the MIS itself can be computed rapidly and in parallel.

Many regular codes are implicitly parallelized using this approach, for instance, the red and black ordering of the parallel Gauss-Seidel algorithm [10]. In contrast, irregular codes often require the explicit computation of an MIS at runtime. Some irregular parallel applications—for example, MIS-based graph coloring [23]—do not produce new conflicts during execution. In this case, only steps 3 through 5 need to be repeated in the later rounds. Other applications require the iteration of all five steps owing to dynamically changing conflicts. Delaunay mesh refinement (DMR) is an example [31]. In fact, one of the first successful parallelizations of DMR is based on this recipe of computing an MIS in each round [16].

This technique is often referred to as an inspector-executor approach. It allows the work identified in each round to be executed in an embarrassingly parallel manner. The conflict graph itself can often also be built in parallel and, importantly, before any of the computations start making updates, thus avoiding the need for buffering old values. This is possible for all cautious computations [24]. There are also computations for which a suitable graph already exists and no separate conflict graph needs to be built. The conventional MIS computation is an example.

In all of these cases, two aspects of the MIS computation are crucial: (1) the runtime should be as short as possible and (2) the MIS should be as large as possible. A high-quality and fast MIS implementation has the potential to substantially speed up and improve not only MIS code but also parallel computations such as the ones listed earlier that employ MIS as a building block. Moreover, since MIS algorithms are useful for parallelizing computations with complex conflict patterns, a high-performance parallel MIS implementation could prove essential in a future dominated by massively parallel devices.

The ECL-MIS algorithm presented in this article exhibits both traits: it is faster than preexisting algorithms and produces larger sets. Moreover, it uses less auxiliary memory than prior implementations. It is asynchronous and atomic free, making it a particularly good fit for highly parallel architectures such as GPUs. ECL-MIS employs a novel priority-assignment technique to boost the expected set size, which is general and can also be used in other MIS codes. This article makes the following main contributions.

- It introduces GPU-friendly optimizations to maximize the performance and reduce the memory footprint of the random-permutation parallel MIS algorithm.
- It presents a partially randomized priority assignment function to boost the size of the MIS.
- It demonstrates that the CUDA implementation of ECL-MIS outperforms the fastest prior GPU and multicore CPU codes severalfold in runtime while, at the same time, producing larger sets.

- It shows that ECL-MIS removes nearly two-thirds of the gap in the set size between conventional MIS algorithms and the MIS size, that is, the largest possible MIS.

The ECL-MIS CUDA code is available at http://cs.txstate.edu/~burtscher/research/ECL-MIS/. The rest of this article is organized as follows. Section 2 discusses related work and surveys prior parallel MIS algorithms. Section 3 describes the design and implementation of ECL-MIS. Section 4 explains the evaluation methodology. Section 5 presents and analyzes the experimental results. Section 6 concludes with a summary.

2 RELATED WORK

A large amount of related work on MIS algorithms exists. In this section, we focus on the work that first introduced the ideas we build on as well as the implementations with which we compare ECL-MIS.

It is easy to compute an MIS sequentially. Just mark all the vertices in the graph $G = (V, E)$ as "undecided" and then visit them in any order. If a visited vertex is still undecided, mark it as belonging to the MIS and mark all of its neighbors as not belonging to the MIS. If a visited vertex is already decided, simply skip it. This algorithm runs in $O(m+n)$ time, where $m = |E|$ denotes the number of edges and $n = |V|$ the number of vertices in the graph, as each edge is traversed at most once and each vertex is visited. In this article, we consider only undirected graphs without loops and without multiple edges between the same two vertices.

Even though it is trivial to compute an MIS serially, it was initially believed that no efficient parallel algorithm may exist [35]. Yet, in the 1980s, Karp and Wigderson [20] developed a fast parallel MIS algorithm, including a more resource-intensive version that is deterministic. Their base algorithm runs in $O(\log^4 n)$ time on $O(n^2)$ processors. At a high level, it works as outlined in Algorithm 1, in which each iteration of the *while* loop is a synchronous round, I is the independent set, H is the remaining graph that still needs to be processed, and $N(S)$ denotes the neighbors of the vertices in S.

ALGORITHM 1: Karp and Wigderson High-Level Parallel MIS Algorithm

```
$I \leftarrow \emptyset$
$H \leftarrow G$
$while\ H \neq \emptyset$ {
    $S \leftarrow independent\ set\ of\ subgraph\ of\ H$
    $I \leftarrow I \cup S$
    $H \leftarrow H - (S \cup N(S))$
}
```

Of course, the key is to determine a suitable subgraph such that an independent set thereof can easily be computed in parallel. Karp and Wigderson's algorithm for accomplishing that is rather involved, but its existence opened the door to simpler and more efficient approaches.

In the following year, Luby [22] designed multiple parallel MIS algorithms that run in expected time $O(\log^2 n)$ given $O(m)$ processors. In 1986, he published the seminal parallel MIS algorithm that is not only simpler than the earlier algorithms but also more efficient [23]. It runs in expected $O(\log n)$ time on $O(m)$ processors and forms the basis of many parallel MIS implementations.

All of Luby's MIS algorithms follow Karp and Wigderson's high-level outline (Algorithm 1) but employ different subgraph extraction schemes. One of the simplest and fastest such approach is the following. (1) Copy H into S. (2) Assign a random number to all vertices in S. (3) Check all edges in S. For each edge, mark the endpoint vertex with the higher random number as deleted. Break

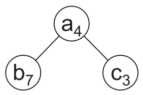

Fig. 1. Sample graph that may require multiple rounds of computation.

ties, for example, using unique vertex IDs. Crucially, all of these steps can be executed in parallel. While they are guaranteed to result in an independent set of (nondeleted) vertices in S, this set is generally not maximal, which is why multiple rounds are needed. Luby proves that, with very high probability, this algorithm needs no more than O(log n) rounds [23]. It is sometimes referred to as the *random-priority* parallel MIS algorithm.

A variation of this approach, also by Luby, is the *random-selection* parallel MIS algorithm [23]. It uses random numbers to select vertices with probability $0.5/d(v)$, where $d(v)$ denotes the degree of vertex v. For all edges whose two endpoints are both selected, deselect the one with the lower degree and break ties using, for example, unique vertex IDs. This algorithm chooses lower-degree vertices with higher probability for inclusion in the MIS. Since low-degree vertices have few neighbors, including such a vertex disqualifies only few other vertices (the neighbors) from inclusion, thus increasing the chance of finding a larger MIS.

A third algorithm, called the *random-permutation* parallel MIS algorithm, operates like Luby's random-priority algorithm except that it does not assign new random numbers in each round. Rather, it reuses the previously assigned numbers. In other words, it assigns random numbers only once, which is tantamount to selecting a random permutation of all the vertices in the original graph. Blelloch et al. [3] show that the resulting algorithm runs in O($\log^2 n$) time on a CRCW PRAM, is deterministic, and, based on the chosen permutation, produces the same result as the corresponding serial algorithm. Our ECL-MIS code implements this random-permutation parallel MIS algorithm except that it employs a partial random permutation to increase the set size.

We compare ECL-MIS to the following three GPU suites that include high-performance MIS implementations: CUSP [8], Pannotia [5], and IrGL [29]. Pannotia is written in OpenCL, CUSP in CUDA, and IrGL in its own language. All three codes are deterministic and implement the random-permutation parallel MIS algorithm. CUSP is an open-source CUDA library of generic parallel algorithms for sparse linear algebra and graph computations. It uses a sparse-matrix representation of the graph, which is processed through repeated calls into the Thrust library [28]. Pannotia employs a compressed sparse row (CSR) data structure for representing the graph and directly operates on it without library calls. The IrGL code is expressed in a high-level language that is automatically optimized and compiled into CUDA [29]. It also uses the CSR graph representation, upon which it operates directly. However, it incorporates a different random number generator and therefore produces slightly different MISs.

All of the algorithms described so far operate in multiple synchronous rounds, where the output of one round is the input to the next round. The three-vertex graph depicted in Figure 1, with the subscripts indicating the random numbers, illustrates why multiple rounds may be necessary.

Depending on the timing of the parallel execution, vertex a may temporarily prevent vertex b from being included in the MIS because a's random number is lower than b's. However, a has another neighbor c with an even lower random number. Including c in the MIS will result in the dropping of all of its neighbors, including a. This, in turn, causes b's random number to now be the lowest, thus allowing b to join the MIS. Since b's neighbors might have been checked before a was dropped, b will have to again check its neighbors' random values in the next round.

It is possible to implement these rounds in an asynchronous manner [1]. Doing so allows a thread to retry its vertices before the other threads have finished their work, which reduces waiting at

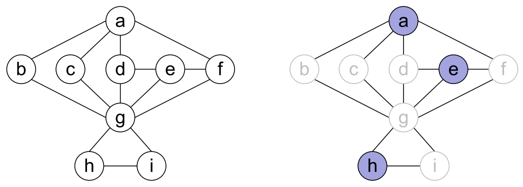

Fig. 2. Sample graph (left panel) and MIS solution of serial algorithm (right panel).

the cost of possibly performing redundant work. ECL-MIS operates asynchronously to boost its performance and to minimize PCI bus transfers.

The last few years have seen a surge in work on identifying the largest possible MIS, that is, the maximum independent set (MuIS). Whereas the MuIS is intractable to compute in general, it can often be computed precisely and closely approximated in most of the remaining cases. Finding an MuIS is typically done using a branch-and-reduce algorithm [2], that is, based on a completely different approach than the MIS algorithms described earlier. Most of the MuIS work focuses on improving techniques for reducing the exponential search space. For example, Dahlum et al. propose to cut high-degree vertices to speed up the search [7], Ghaffari employs randomization [11], and Jin and Hao use swap operators and tabu search [18]. Lamm et al. combine heuristics with an evolutionary algorithm to determine vertices that are very likely to be in large independent sets [21]. They implemented their approach in the KaMIS code [19], which we evaluate in our Results section. To avoid having to try all reduction rules on every vertex, Chang et al. initially apply only low-degree reductions to shrink the graph size while preserving the maximality of the independent sets [4]. Moreover, they temporarily remove (peel) the vertex with the highest degree if the reduction rules cannot be applied because high-degree vertices are unlikely to be in an MuIS. The resulting approach, which we compare to in our Results section, is implemented in the NearLinear code [26]. Note that these codes employ sophisticated auxiliary data structures to manage the search space and, while they are very concerned about speed to make the search tractable, they do not focus on parallelization.

Examples

This section illustrates how the MIS algorithms described earlier work on the example graph shown in the left panel of Figure 2. The right panel shows a possible MIS of this graph, where the colored vertices belong to the set. The depicted MIS of size three is maximal, as all of the remaining (grayed out) vertices have at least one neighbor that is in the set.

In the current MIS literature, lower random numbers (priorities) trump higher numbers. However, we use a higher-is-better approach in the following examples and throughout the rest of this article, which we find more intuitive. This way, a higher random number signifies a higher priority. This switch in meaning is immaterial to the execution of the algorithms but, in our opinion, clarifies the discussion.

The right panel of Figure 2 shows the result of the serial MIS algorithm outlined earlier assuming that it visits the vertices in alphabetical order. It starts with an empty set. Then, it visits vertex *a*, which has no neighbors in the set and is therefore included (marked). Next, vertices *b, c*, and *d* each have a neighbor in the set; thus, they are excluded (grayed out). Vertex *e* can be included in the set, but vertices *f* and *g* cannot. Finally, vertex *h* can again be included but not vertex *i*. The resulting MIS is {*a, e, h*}.

Figure 3 illustrates how the random-priority algorithm works. It starts by assigning random priorities as shown in panel (a). Then, in parallel, it includes all vertices in the set that have the

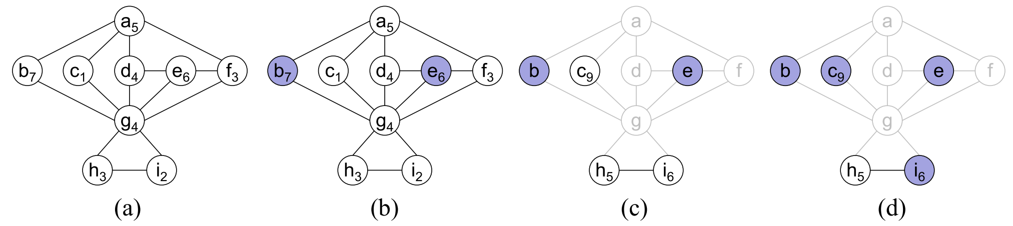

Fig. 3. Steps of the random-priority parallel MIS algorithm: (a) inserting random priorities, (b) selecting vertices with locally maximal priorities, (c) removing their neighbors (grayed out), and (d) repeating these steps for the second round.

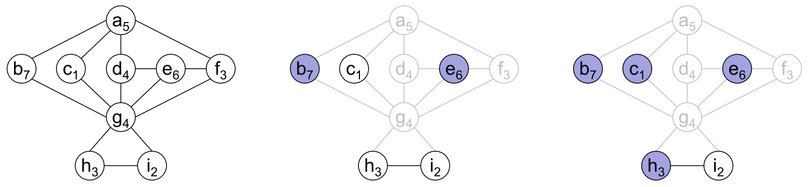

Fig. 4. Steps of the random-permutation parallel MIS algorithm: (a) inserting random priorities, (b) selecting vertices with locally maximal priorities and removing their neighbors (grayed out), and (c) repeating these steps for the second round using the same priorities.

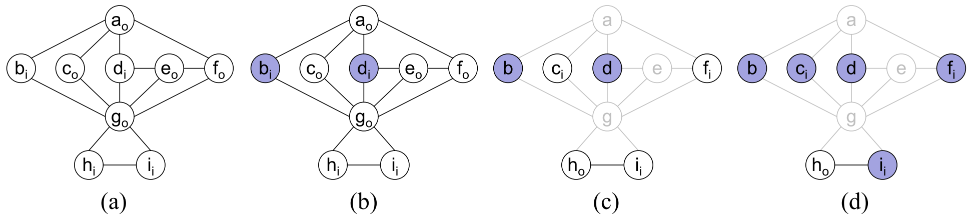

Fig. 5. Steps of the random-selection parallel MIS algorithm: (a) biased-randomly selecting *in* and *out* vertices, (b) marking *in* vertices without any *in* neighbors, (c) removing their neighbors (grayed out) and biased-randomly selecting new *in* and *out* vertices, and (d) performing the next round of selecting *in* vertices without any *in* neighbors.

highest priority among their neighbors. In this example, that is vertices *b* and *e*, as shown in panel (b). Next, their neighbors are excluded and the remaining undecided vertices receive new random priorities, as shown in panel (c). Finally, the process of finding vertices with locally maximal priorities repeats, which results in including vertices *c* and *i*, as shown in panel (d). After excluding their neighbors (not shown), no vertices are left and the algorithm terminates with an MIS of {*b, c, e, i*}.

Figure 4 illustrates how the random-permutation algorithm works. The first steps are the same as before. It starts by assigning random priorities (left panel). Then, it includes all vertices in the set that have the highest priority among their neighbors, which are again vertices *b* and *e*, and their neighbors are excluded (middle panel). Importantly, the remaining undecided vertices retain their prior random values. Then, the process of finding vertices with locally maximal priorities repeats, which results in including vertices *c* and *h* (right panel). After excluding their neighbors, the algorithm terminates with an MIS of {*b, c, e, h*}.

Figure 5 illustrates how the random-selection algorithm works. It starts by randomly picking vertices as being “in” (“i” subscript) or “out” (“o” subscript) with probability $0.5/d(v)$, where $d(v)$

denotes the degree of vertex *v*, as shown in panel (a). Then, in parallel, it includes all vertices in the set that are "in" and have only neighbors that are "out." In this example, that is vertices *b* and *d*, as shown in panel (b). Next, their neighbors are excluded and the remaining undecided vertices are newly marked as "in" or "out." Note, however, that the probabilities are now different since the degrees of the vertices may have changed owing to the excluded (grayed out) edges as shown in panel (c). Finally, the process of finding vertices that are "in" and have only "out" neighbors repeats, which results in including vertices *c, f,* and *i*, as shown in panel (d). After excluding their neighbors (not shown), no vertices are left and the algorithm terminates with an MIS of {*b, c, d, f, i*}.

3 ECL-MIS IMPLEMENTATION

ECL-MIS, like many other MIS implementations, is based on the random-permutation parallel MIS algorithm. Similar to Pannotia and IrGL, it operates on the graph in CSR format. The following sections provide more detail on the implementation. First, we discuss optimizations to increase performance and reduce the memory footprint. Second, we present a distribution of priorities among the vertices that improves the quality of the solution for a wide range of graphs. In case two adjacent vertices have the same priority, we resort to the unique vertex IDs to break the tie. Conceptually, this approach concatenates the priorities with the vertex IDs to form unique priorities. We assume such concatenated values to simplify the discussion.

The priorities impose a fixed ordering on the vertices, that is, they select a specific permutation, which Blelloch et al. showed to run in no more than O($\log^2 n$) rounds *with high probability* [3]. Also, any serial or parallel execution that adheres to a given permutation always yields the same MIS, that is, the result is deterministic regardless of the parallel execution's internal timing.

Base Implementation

The pseudo-code in Listing 1 provides a baseline GPU implementation of the random-permutation parallel MIS algorithm. We assume that the following kernels (i.e., functions that execute on the GPU but are called from the CPU) run with *k* threads, each of which has a unique thread ID *tid* $\in$ N, $0 \leq tid < k$, except the *work_remains_kernel*, which needs only one thread. All undeclared variables are shared and global.

The code first assigns the random priorities and sets the status of each vertex to "undecided." Then, it repeatedly calls the compute kernel until no undecided vertices are left. In each round, the vertices are assigned to the threads in a cyclic fashion (line 07). This is important on GPUs as it enables coalesced (fast) memory accesses, which is also the reason why the random numbers and status information are kept in two separate arrays. If the current vertex is still undecided (line 08), its neighbors are visited to find the highest priority (line 09). This search must exclude all neighbors that have already been decided to be "out" (see the example in Figure 1). If the current vertex's priority is higher than those of its neighbors (line 10), it is "in" the MIS (line 11) and all of its neighbors are "out" (line 12). Otherwise, the status of the current vertex cannot yet be determined and another round is needed (line 14).

Each vertex is initially "undecided" and changes its status exactly once to either "in" or "out." When its status has been decided, it will no longer change. However, it is possible for a vertex to redundantly be marked as "out" by multiple threads. Also note that only "in" threads ever update the status array.

Performance Optimizations

CUSP's, Pannotia's, and IrGL's MIS implementations roughly follow the baseline code described here. However, this approach is somewhat inefficient owing to the search for the neighbor with the highest priority (line 09). It is not actually necessary to identify this neighbor or its priority,

```
01: void init_kernel() {
02:   random[n] = …; //init random priorities
03:   status[n] = …; //init to undecided
04:   need_another_round = false;
05: }

06: void compute_kernel() {
07:   for (int v = tid; v < n; v += k) {
08:     if (status[v] == undecided) {
09:       int best = get_best_neighbor(v);
10:       if (random[v] > best) {
11:         status[v] = in;
12:         mark_neighbors_out(v);
13:       } else {
14:         need_another_round = true;
15:       }
16:     }
17:   }
18: }

19: bool work_remains_kernel() {
20:   bool wr = need_another_round;
21:   need_another_round = false;
22:   return wr;
23: }

24: void MIS() {
25:   //allocate memory and transfer graph
26:   init_kernel();
27:   do {
28:     compute_kernel();
29:   } while (work_remains_kernel());
30: }
```

Listing 1: Baseline Parallel GPU MIS Code

as the test (line 10) needs to check only whether any neighbor with a higher priority exists. As soon as such a neighbor is found, the search can be terminated and the remaining neighbors do not have to be visited. This inefficiency is also present in some parallel CPU codes [12]. However, the PBBS [30] multicore CPU implementation of MIS employs the same early-out optimization as ECL-MIS.

Many parallel MIS codes maintain two pieces of information per vertex: a random number and a status. Since a vertex can be in one of three states, 'undecided', 'in', or 'out', usually at least a byte is allocated per vertex to hold the status information. The random number, i.e., the priority, is kept in a separate field that is typically an integer (four bytes). This arrangement not only results in a significant memory footprint but also requires two memory accesses to read or write this information. We combine these two pieces of information into a single byte by exploiting the following properties. First, as mentioned, all vertices are initially 'undecided' and switch to either 'in' or 'out' exactly once. This is a form of monotonicity, i.e., the status moves in a specific direction and never back [25]. Second, in the random-permutation MIS algorithm, the random numbers do not change during the course of the execution and are only used while a vertex is 'undecided'. Third, only threads processing an 'in' vertex update the status information of that vertex and of its neighbors.

In our implementation, we reserve the highest priority to indicate an “in” vertex. Similarly, the lowest priority is reserved to denote “out.” This is accomplished by ensuring that “undecided” vertices are only assigned priorities between the highest and lowest value. As a result, we can express the status and the priority of a vertex in a single byte, in which the lowest value represents “out,” the highest value represents “in,” and all values in between represent “undecided” and simultaneously provide the priority. This approach reduces the memory footprint as well as the number of memory accesses.

Since the priority is no longer needed once a vertex’s status has been decided, it is safe to overwrite and lose this information. However, doing so must not invalidate any prior, ongoing, or future decisions that the program makes. Recall that the algorithm only changes a vertex’s status to “in” only if the vertex has the highest priority among its neighbors. Since we assigned “in” the highest overall priority, overwriting such a vertex’s priority with “in” maintains this invariant. The neighboring vertices still have a neighbor with a higher priority. Importantly, it does not matter when they observe this update to “in.” They will determine that they are not the highest priority vertex independent of whether they see the higher undecided priority or the even higher “in” priority of their neighbor. Hence, the vertex’s status can safely be updated to “in” at an arbitrary time. Note that writing a byte is a naturally atomic operation.

Changing a vertex’s status to “out” works similarly. Only vertices that are guaranteed to have a higher-priority neighbor might have their priority lowered to “out,” after which point they still have a higher-priority neighbor. Moreover, any thread processing a vertex that has a higher-priority neighbor will not update the status of any vertex. Therefore, this lowering in priority can also be done asynchronously.

“Out” vertices must not prevent any of their neighbors from joining the MIS, which is automatically achieved by assigning “out” the lowest priority. In fact, and as outlined in the three-vertex example in Figure 1, determining that a vertex is “out” often allows other vertices to be “in” that previously had a neighbor with a higher priority. The combined status and priority value maintains this behavior and eliminates any need for synchronization as long as the neighboring vertices eventually see the “out” information. Importantly, assigning “out” the lowest priority allows the code to work correctly without actually removing any vertices or edges from the graph, which simplifies the implementation.

The remaining concern is termination. Due to the monotonicity, a thread can simply terminate once all vertices assigned to it have been decided. The resulting implementation performs a chaotic relaxation in which threads repeatedly try to determine their vertices’ status until they are either “in” or “out.” Forward progress, and therefore termination, is guaranteed as either all vertices are decided or there is one undecided vertex with the globally highest non-“in” priority (owing to the tie breaker) that can unequivocally be decided to be “in.” ECL-MIS employs a persistent-thread implementation [14], in which the threads process the vertices assigned to them in a round-robin fashion, to guarantee that this highest non-“in” priority vertex is eventually processed regardless of the status and priority of any other vertex.

ECL-MIS requires only one invocation of each kernel, no explicit synchronization primitives, a single combined status/priority per vertex, and no termination information to be sent back to the CPU. Since it implements the random-permutation algorithm, it is deterministic. It is faster because it uses less memory, executes fewer instructions (memory accesses, synchronization operations, and comparisons as the status/priority checks are combined), minimizes PCI bus transfers, and reduces waiting since there are no barriers between rounds. Note that combining the status and priority in a single byte is compatible with the priority-assignment function presented in the next section.

Interestingly, the code spends most of its execution time checking whether a vertex's status is still undecided (line 08). This test would normally require two comparisons because there are two decided states, "in" and "out," that are separated by many "undecided" states. To speed up this check without breaking any of the previously described optimizations, we use even values to represent "in" and "out" and use odd priorities that only lie between "in" and "out." As a consequence, testing whether a vertex has been decided is simplified to checking whether its combined status/priority is even. This approach is tantamount to making the least significant bit (LSB) a flag that indicates whether a vertex has been decided or not. If it is zero (even), the vertex is decided and the remaining bits indicate whether it is "in" or "out." If it is one (odd), the remaining bits specify the priority. Note that the LSB must be used for this purpose so as not to break the invariants described earlier.

Boosting the Set Size

The random-priority and random-permutation MIS algorithms assign uniformly distributed random numbers to the vertices. In contrast, the random-selection MIS algorithm selects vertices with probability $0.5/d(v)$, where $d(v)$ is the degree of vertex v. As mentioned, this biased assignment increases the chance of finding a larger MIS. Since these assignments were chosen so that theoretical bounds for the runtime of the algorithms could be proven, it is quite possible that other assignments exist that yield larger sets.

Based on the observation that the $0.5/d(v)$ approach tends to produce larger sets because it prefers low-degree vertices (see Section 2) and encouraged by the result from Blelloch et al. that almost all vertex permutations in the random-permutation algorithm end up requiring just a few parallel rounds to find an MIS [3], we set out to seek a better permutation to boost the set size. So as not to hurt the execution time, we limited ourselves to approaches that require only O(1) time per vertex. For example, we did not want to use information such as the maximum vertex degree because this information is not readily available in the CSR graph representation. However, the degree of a vertex as well as the average degree can be obtained in constant time.

We arbitrarily decided to compute priorities between zero and one, which we then scale and offset to the desired range (presented later). Unlike the random-selection approach, which only *prefers* low-degree vertices by selecting them with higher probability, we want to *always* give lower-degree vertices a higher priority because high-degree vertices are unlikely to be in large independent sets [4, 7]. Moreover, prioritizing vertices in a similar manner has proven useful in other related algorithms, such as graph coloring [15]. Hence, we need to construct a priority-assignment function that monotonically decreases for higher degrees. Since zero-degree vertices can trivially be included in the MIS, the function should assign the highest priority to vertices of degree one. Vertices with more neighbors must receive lower but still positive priorities. To spread out the priorities irrespective of how dense the graph is, that is, irrespective of the average degree, we opted for a function that maps all vertices with below average degrees to the upper half of the priority range and the rest to the lower half. Finally, we still want randomization in case the graph has many vertices with the same degree, which is common for some graph types, such as road maps. To incorporate randomization without altering the priority imposed by the vertex degree, we subtract a random number in the range (0, 1] from each vertex's degree. The following function fulfills these criteria:

$$f(x) = d_{avg}/(d_{avg} + x), \textit{ where } x = d(v) - rand(id(v)).$$

Here, d_{avg} refers to the average degree, that is, the number of edges divided by the number of vertices, $d(v)$ denotes the degree of vertex v, $id(v)$ is the unique identifier of vertex v, and $rand(z)$ deterministically maps the integer z to a random value between zero and one but not including one. Using the vertex ID as a random number seed guarantees that $f(x)$ always produces the same

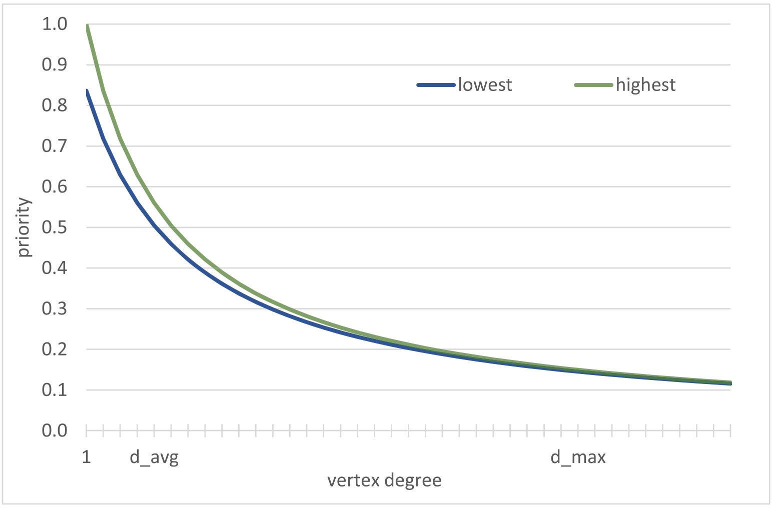

Fig. 6. Priority range as a function of the vertex degree (between the lowest and highest random value) for $d_{avg} = 5.1$.

priority for a given vertex of a given graph and therefore the same permutation of the vertices. This, in turn, makes the overall MIS algorithm deterministic.

Figure 6 plots *f(x)* for an arbitrarily chosen average degree of 5.1. It shows the lowest possible function value, which assumes that the random number is zero, and an upper bound for the highest possible value, which assumes that the random number is one. The range between the lowest and highest value specifies the range of possible priorities that a vertex of a given degree can be assigned.

Whereas the function values depend on d_{avg}, other average degrees simply stretch or contract the function "horizontally" such that the lowest value at $f(d_{avg})$ is always 0.5. Together, the ranges between the lowest and highest value for the integer vertex degrees 1 through ∞ form a partition, that is, they do not overlap and cover the entire range $[0, 1)$. For instance, in the example presented earlier with $d_{avg} = 5.1$, all vertices with degree one will be assigned a priority between 0.84 and 1.0, all vertices with degree two will be assigned a priority between 0.72 and 0.84, and so on. In practice, the range between zero and $f(d_{max}+1)$, where d_{max} is the largest occurring degree, remains unused. However, we assume that d_{max} is not available, which is why we cannot just stretch the function "vertically" to eliminate this gap. Of course, d_{max} is bounded by the number of vertices in the graph, but that number tends to be too large to make a difference. For example, with one million vertices, the guaranteed unused range is just 0.0005%, which we simply ignore. Nevertheless, small actual d_{max} values can result in a significant unused range, but only if d_{max} is close to d_{avg}. In the worst case, where d_{avg} and d_{max} are equal, half of the function range cannot be reached.

To obtain an integer priority, we multiply the function value by the largest desirable priority and round the result to the nearest integer. Owing to the nature of *f(x)*, this discretization yields a larger number of possible priorities for lower-degree vertices than for higher-degree vertices (see Figure 6), which is important when only a small range of priorities is available. After all, the randomization is primarily needed to permute the low-degree vertices relative to each other, which have a high chance of being included in the MIS. Not (well) permuting the high-degree vertices is unlikely to negatively affect the MIS quality, as those vertices have a low probability of being included in the set. Hence, our priority-assignment function *f(x)* not only prioritizes lower-degree

```
01: void init_kernel() {
02:   statprio[n] = …; //init combined status and priority using f(x)
03: }
04: void compute_kernel() {
05:    bool done;
06:    do {
07:       done = true;
08:       for (int v = tid; v < n; v += k) {
09:          if (statprio[v] & 1) {
10:             if (best_among_neighbors(v)) {
11:                mark_neighbors_out(v);
12:                statprio[v] = in;
13:             } else {
14:                done = false;
15:             }
16:          }
17:       }
18:    } while (!done);
19: }
20: void MIS() {
21:    init_kernel();
22:    compute_kernel();
23: }
```

Listing 2: Parallel ECL-MIS GPU Code

vertices but, at the same time, makes it possible to use a narrow range of integers for representing the priorities. In other words, we can use a small-integer type to store the priorities without ill effects on the resulting solution quality, thus allowing substantial reduction (by a factor of four when going from *int* to *char*) of the memory footprint for storing the priorities. Doing so lowers the memory bandwidth requirement and should improve in-cache presence, the latter of which is particularly important on GPUs, which tend to have relatively small caches.

ECL-MIS Code

The pseudo-code in Listing 2 includes all of the aforementioned enhancements. Undeclared variables are global and shared. The complete CUDA code is available at http://cs.txstate.edu/~burtscher/research/ECL-MIS/.

There is only a single *statprio* array as opposed to separate status and priority arrays. Moreover, the *statprio* array is an array of bytes (unsigned characters) rather than 4B integers. It is initialized using the function *f(x)* described earlier to boost the size of the resulting MIS. As mentioned, only odd priorities are used that lie between the values denoting "in" and "out." Vertices without neighbors are initialized to "in."

The compute kernel is called only once and no data are transferred over the PCI express bus to indicate whether the kernel needs to be called again. Instead, each thread uses a local variable (line 05) to track whether it is done with its work and can terminate (line 18). In each iteration of the *do* loop, a thread goes over all of its cyclically assigned vertices (line 08) and performs a quick check for whether a vertex is still undecided, that is, odd (line 09). If so, it checks whether the current vertex has the highest priority among its neighbors using the unique vertex IDs as tie breaker if necessary (line 10). Importantly, this test uses a short-circuit evaluation, that is, it stops as soon as a higher-priority neighbor has been found rather than identifying the actual

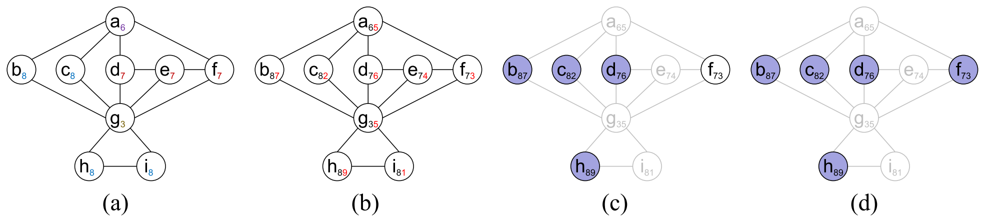
a6 b8 c6 d7 e7 f7 g3 h8 i8
(a)
a65 b87 c82 d76 e74 f73 g35 h89 i81
(b)
a65 b87 c82 d76 e74 f73 g35 h89 i81
(c)
a65 b87 c82 d76 e74 f73 g35 h89 i81
(d)

Fig. 7. Steps of the parallel ECL-MIS algorithm: (a) inserting priorities based on vertex degree, (b) randomizing the priorities, (c) selecting vertices with locally maximal priorities and removing their neighbors (grayed out), and (d) repeating these steps in the second round using the same priorities.

highest-priority neighbor. If the current vertex has the highest priority, it marks all of its neighbors as "out" (line 11) and then marks itself as being "in" the MIS (line 12). Otherwise, the current vertex's status cannot be decided and needs to be revisited later (line 14). The final result is returned in the *statprio* array.

Figure 7 illustrates how the ECL-MIS algorithm works on the graph that was used as an example in the previous section. It starts by assigning priorities that are inversely proportional to the vertex degree. This results in vertices with the same degree getting the same priority as shown in panel (a). To reduce the many ties, the priorities are randomized between vertices of the same degree. This is accomplished in panel (b) by appending a random least significant digit to each priority value. The following steps are identical to those of the parallel random-permutation MIS algorithm. First, all vertices are included in the set that have the highest priority among their neighbors. In this example, those are vertices *b, c, d*, and *h*, and their neighbors are excluded, as shown in panel (c). Then, the process of finding vertices with locally maximal priorities repeats using the same random values, which results in including vertex *f*, as shown in panel (d). There are no undecided vertices left; thus, the algorithm terminates with an MIS of {*b, c, d, f, h*}.

4 EXPERIMENTAL METHODOLOGY

In addition to ECL-MIS, we evaluate the following three GPU codes that compute an MIS. The first is part of NVIDIA's CUSP library 0.3.1 [27], the second is from the Pannotia suite 0.9 [13], and the third is from the IrGL system [29]. In addition, we measured the parallel CPU codes from the Ligra/Ligra+ framework [12] and the Problem-Based Benchmark Suite (PBBS) 0.1 [30]. Ligra+ is a variant of Ligra that uses and operates on a compressed graph representation to minimize its memory footprint. For PBBS, which currently provides the fastest CPU implementation of MIS, we tested four parallel versions: incremental OpenMP, incremental CILK, nondeterministic OpenMP, and nondeterministic CILK. On average, the nondeterministic CILK version is the fastest on our system, which is why we present results for it and omit the other three versions to improve readability. For reference, we also show results for serial PBBS. Except for Pannotia, in which we replaced the hardcoded MIS example with general code to read in a graph from secondary storage, we did not alter these downloaded codes and installed them as prescribed by the respective authors.

We present throughputs for two different GPU generations. The first GPU is a GeForce GTX Titan X, which is based on the Maxwell architecture. The second GPU is a Tesla K40c, which is based on the older Kepler architecture. The Titan X has 3072 processing elements distributed over 24 multiprocessors that can hold the contexts of 49,152 threads. Each multiprocessor has 48KB of L1 data cache. The 24 multiprocessors share a 2MB L2 cache as well as 12GB of global memory with a peak bandwidth of 336GB/s. We use the default clock frequencies of 1.1GHz for the processing elements and 3.5GHz for the GDDR5 memory. The K40 has 2880 processing elements distributed

Table 1. Information about the 16 Graphs Used as Inputs (d = degree)

graph name	type	origin	vertices	edges	d_{min}	d_{avg}	d_{max}
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	2	4.0	4
amazon0601	product co-purchases	SNAP	403,394	4,886,816	1	12.1	2,752
as-skitter	Internet topology	SNAP	1,696,415	22,190,596	1	13.1	35,455
citationCiteseer	publication citations	SMC	268,495	2,313,294	1	8.6	1,318
cit-Patents	patent citations	SMC	3,774,768	33,037,894	1	8.8	793
coPapersDBLP	publication citations	SMC	540,486	30,491,458	1	56.4	3,299
delaunay n24	triangulation	SMC	16,777,216	100,663,202	3	6.0	26
in-2004	web links	SMC	1,382,908	27,182,946	0	19.7	21,869
internet	Internet topology	SMC	124,651	387,240	1	3.1	151
kron_g500-logn21	Kronecker	SMC	2,097,152	182,081,864	0	86.8	213,904
r4-2e23.sym	random	Galois	8,388,608	67,108,846	2	8.0	26
rmat16.sym	RMAT	Galois	65,536	967,866	0	14.8	569
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	0	15.7	3,687
uk-2002	web links	SMC	18,520,486	523,574,516	0	28.3	194,955
USA-road-d.NY	road map	Dimacs	264,346	730,100	1	2.8	8
USA-road-d.USA	road map	Dimacs	23,947,347	57,708,624	1	2.4	9

over 15 multiprocessors that can hold the contexts of 30,720 threads. Each multiprocessor is configured to have 48KB of L1 data cache. The 15 multiprocessors share a 1.5MB L2 cache as well as 12GB of global memory with a peak bandwidth of 288GB/s. We disabled ECC protection of the main memory and use the default clock frequencies of 745MHz for the processing elements and 3GHz for the GDDR5 memory. Both GPUs are plugged into 16x PCIe 3.0 slots in the same system, which has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1GHz. The host memory size is 128GB and the operating system is Fedora 22. The CUDA driver is 361.42.

We compiled all codes with nvcc 7.5. In all cases, we used the "-O3 -arch=sm_52" compiler flags for the Titan X and "-O3 -arch=sm_35" for the K40. The CPU codes were compiled with g++ 5.3.1 using the "-O3 -march=native" flags.

When measuring the performance, we consider only the computation time, excluding the time it takes to read in the graphs, transfer the graphs to the GPU, or to transfer the result back. In other words, we assume the graph to already be on the GPU (or CPU) from a prior processing step and the result of the MIS computation to be needed on the GPU (or CPU) by a later processing step. We repeated each experiment seven times and report the median performance. To measure the quality of the solution, we use the MIS size.

We evaluated the codes on the 16 graphs listed in Table 1, which were obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (DIMACS) [9], the Galois framework (Galois) [17], the Stanford Network Analysis Platform (SNAP) [32], and the Sparse Matrix Collection (SMC) [33]. When necessary, we modified them to eliminate loops and multiple edges between the same two vertices. Furthermore, we added any missing back edges to make the graphs undirected.

While it may or may not be useful to compute an MIS on some of these graphs, we selected them to be able to measure the performance and quality of the tested codes on a wide variety of graphs. In particular, the number of vertices differs by up to a factor of 365, the number of edges by up to a factor of 1352, the average degree by up to a factor of 36, and the maximum degree by up to a factor of 53,476.

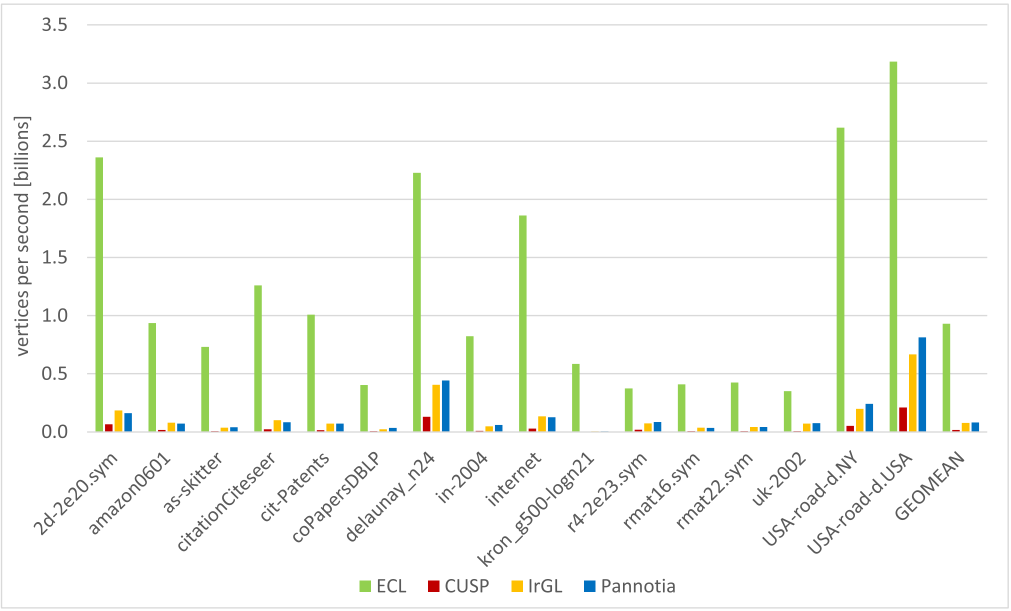

Fig. 8. Throughput in billions of CVPS on the Titan X.

5 RESULTS AND ANALYSIS

This section presents and discusses the results of our measurements. First, we investigate the throughput. Second, we study the solution quality. Third, we evaluate different priority assignment functions. Fourth, we take a look at several code optimizations. Fifth, we compare the throughput and set size to those of parallel and serial CPU codes. Finally, we assess the MIS sizes in comparison with the MuIS sizes. All averages in this section refer to the geometric mean.

Throughput

Since the runtimes vary by over a factor of 40,000, we present the performance in terms of throughput. For this purpose, we use the completed-vertices-per-second (CVPS) metric, that is, the number of vertices in the graph divided by the median runtime, which is related to the traversed-edges-per-second (TEPS) metric used in the Graph 500 [34]. Unlike runtime, CVPS is a higher-is-better metric. Figure 8 shows the throughputs of the four GPU-based MIS codes on the Titan X and Figure 9 on the K40.

ECL-MIS's throughput ranges from 0.35 to 3.2 giga-CVPS (GCVPS) on the Titan X with a geometric mean of 0.93 GCVPS. On the K40, it ranges from 0.2 to 1.8 GCVPS with a geometric mean of 0.51 GCVPS. ECL-MIS is faster than the other three codes on all 16 tested graphs on both GPUs. On average, it outperforms CUSP, IrGL, and Pannotia by a factor of 56.5, 12.2, and 11.5 on the Titan X, respectively, and by a factor of 43.7, 10.6, and 9.4 on the K40, respectively. CUSP yields the lowest throughput because of repeatedly calling a library. For reference, Table 2 lists the runtimes (in milliseconds) and the corresponding speedups on the Titan X and Table 3 on the K40.

ECL-MIS's performance advantage correlates with the average degree of the input graph. This is expected, as the other codes visit all neighbors of each vertex whereas ECL-MIS looks only for the first neighbor whose priority is higher than that of the current vertex. Consequently, the throughput advantage is the lowest on the USA-road-d.USA input (14.3, 4.6, and 3.8, respectively,

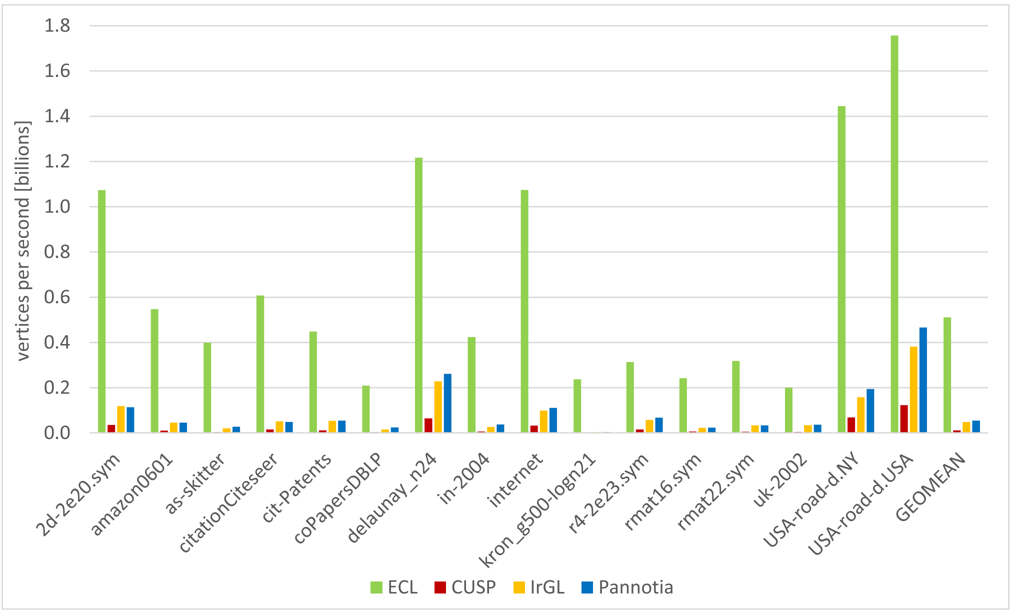

Fig. 9. Throughput in billions of CVPS on the K40.

on the K40), which has an average degree of only 2.4, and the highest on the kron_g500-logn21 input (773.2, 103.9, and 100.1, respectively, on the Titan X), which has an average degree of 86.8.

To further study this difference in behavior, Table 4 lists the correlation between the runtimes and four properties of the graphs: the number of vertices, number of edges, average degree, and maximum degree. A correlation coefficient of 1.0 indicates perfect linear correlation, a coefficient of −1.0 indicates perfect linear anticorrelation, and a coefficient of 0.0 indicates no linear correlation.

The coefficients reveal that ECL-MIS's runtime most strongly correlates with the number of edges in the graph. Whereas the other three codes also exhibit relatively strong correlations with the number of edges, their runtimes most strongly correlate with the largest degree. In the case of CUSP, the runtime is almost perfectly proportional to the maximum degree. Interestingly, ECL-MIS's runtime essentially does not correlate with the average vertex degree whereas the other three codes exhibit a relatively strong correlation. Together, these results again show that ECL-MIS's performance advantage tends to be lower for graphs with a low maximum degree and higher for graphs with a high maximum degree.

Since ECL-MIS uses an asynchronous implementation, its throughput depends on internal timing behavior. Figure 10 shows by how much the throughput of seven runs with the same input varies relative to the median throughput on the K40. The results for the Titan X are similar and not shown.

The asynchronous nature of ECL-MIS does not result in large throughput variations. The average variability is within 1% (±0.7%) and the largest observed variation is a little over 2% (±2.2%), meaning that the throughputs are quite stable. This variability is insignificant compared to ECL-MIS's performance advantage over the other codes of at least 380%.

Set Size

As the absolute MIS sizes vary by nearly a factor of 500, we present the set sizes as a fraction of the total number of vertices. Figure 11 shows the results for the four codes. Since the codes are

Table 2. Absolute Runtimes (in Milliseconds) of the Four GPU Codes Executing on the Titan X as well as the Corresponding ECL-MIS Speedups

graph name	**CUSP runtime**	**IrGL runtime**	**Pannotia runtime**	**ECL-MIS runtime**	**speedup over CUSP**	**speedup over IrGL**	**speedup over Pannotia**
2d-2e20.sym	15.94	5.66	6.51	0.44	35.9	12.8	14.7
amazon0601	22.94	5.13	5.72	0.43	53.2	11.9	13.3
as-skitter	270.34	47.10	41.31	2.32	116.5	20.3	17.8
citationCiteseer	11.42	2.66	3.23	0.21	53.6	12.5	15.2
cit-Patents	246.16	53.28	53.52	3.75	65.7	14.2	14.3
coPapersDBLP	88.00	23.28	15.46	1.34	65.6	17.4	11.5
delaunay_n24	129.53	41.47	37.90	7.53	17.2	5.5	5.0
in-2004	163.11	28.35	23.37	1.68	97.1	16.9	13.9
internet	4.47	0.93	0.98	0.07	66.8	13.9	14.7
kron_g500-logn21	2,770.25	372.44	358.75	3.58	773.2	103.9	100.1
r4-2e23.sym	434.87	112.93	97.76	22.40	19.4	5.0	4.4
rmat16.sym	9.36	1.83	1.93	0.16	58.5	11.5	12.1
rmat22.sym	617.26	100.03	99.53	9.86	62.6	10.1	10.1
uk-2002	2,387.86	257.90	244.31	52.78	45.2	4.9	4.6
USA-road-d.NY	5.08	1.33	1.10	0.10	50.3	13.2	10.9
USA-road-d.USA	113.95	35.87	29.49	7.52	15.1	4.8	3.9
GEOMEAN	91.03	19.59	18.51	1.61	56.5	12.2	11.5

Table 3. Absolute Runtimes (in Milliseconds) of the Four GPU Codes Executing on the K40 as well as the Corresponding ECL-MIS Speedups

graph name	**CUSP runtime**	**IrGL runtime**	**Pannotia runtime**	**ECL-MIS runtime**	**speedup over CUSP**	**speedup over IrGL**	**speedup over Pannotia**
2d-2e20.sym	29.15	8.80	9.18	0.98	29.8	9.0	9.4
amazon0601	40.50	8.86	8.86	0.74	54.9	12.0	12.0
as-skitter	456.22	83.15	62.22	4.25	107.3	19.6	14.6
citationCiteseer	17.65	5.22	5.52	0.44	39.9	11.8	12.5
cit-Patents	340.47	70.15	69.76	8.42	40.4	8.3	8.3
coPapersDBLP	140.57	35.28	22.24	2.58	54.5	13.7	8.6
delaunay n24	261.65	73.62	64.22	13.78	19.0	5.3	4.7
in-2004	228.62	51.36	36.68	3.26	70.0	15.7	11.2
internet	3.87	1.26	1.13	0.12	33.4	10.9	9.7
kron_g500-logn21	3,243.49	649.32	680.14	8.86	366.2	73.3	76.8
r4-2e23.sym	559.95	144.42	123.74	26.79	20.9	5.4	4.6
rmat16.sym	10.39	2.88	2.81	0.27	38.5	10.7	10.4
rmat22.sym	845.49	127.07	125.68	13.16	64.3	9.7	9.6
uk-2002	4,196.61	541.99	504.70	92.41	45.4	5.9	5.5
USA-road-d.NY	3.83	1.68	1.36	0.18	20.9	9.2	7.4
USA-road-d.USA	194.98	62.69	51.32	13.63	14.3	4.6	3.8
GEOMEAN	128.12	31.13	27.71	2.93	43.7	10.6	9.4

Table 4. Linear Correlation Coefficients between the Runtimes and Various Graph Properties

	ECL	CUSP	IrGL	Pannotia
vertices	0.61	0.25	0.26	0.25
edges	0.91	0.82	0.74	0.74
avg deg	0.07	0.71	0.73	0.72
max deg	0.56	0.98	0.93	0.93

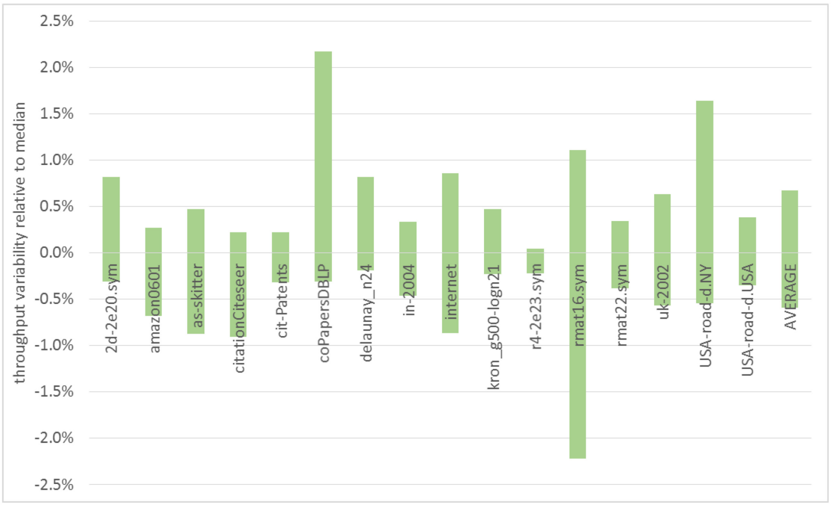

Fig. 10. Throughput variability between seven runs of the same input on the K40 relative to the median throughput.

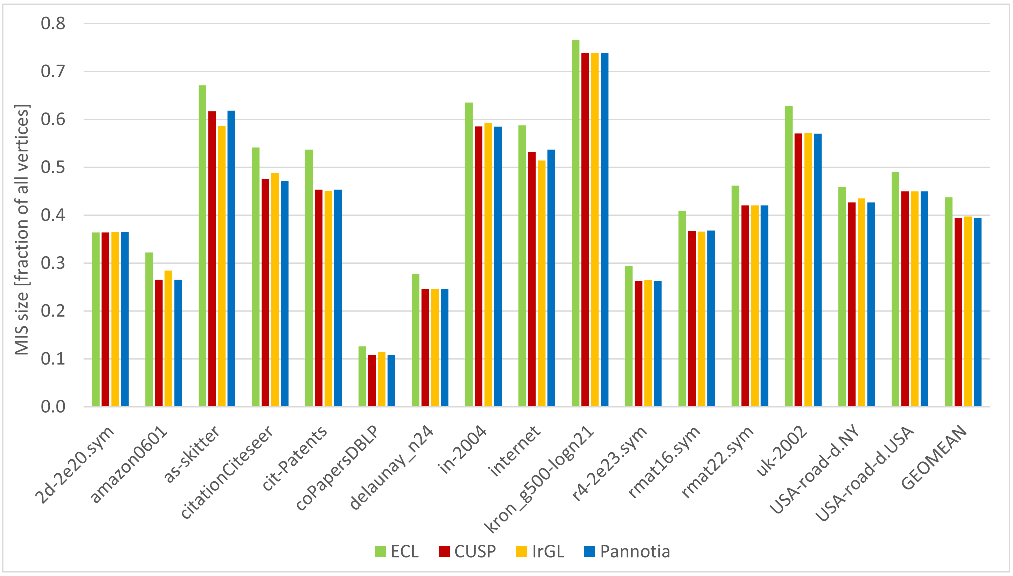

Fig. 11. MIS size as a fraction of all vertices for different GPU codes.

deterministic, they always produce the same set for a given input, irrespective of the GPU that they run on.

CUSP and Pannotia's solutions are almost identical. Even though the IrGL code is based on the same algorithm as Pannotia, it uses a different random number generator, which is why their solution quality sometimes differs. Nevertheless, on average, the three codes result in nearly identical set sizes.

ECL-MIS yields larger sets than CUSP on all tested inputs. It also produces larger sets than IrGL and Pannotia except on the 2d-2e20.sym input, where the other two codes result in sets that are 0.1% larger. As discussed in Section 1, larger sets are often preferred. On average, ECL-MIS's priority assignment yields 10.1% larger MIS sizes. On cit-Patents, the set is at least 18.5% larger than that of the other three codes.

Since ECL-MIS employs randomization, the solution it produces depends on random chance. To test the effect on the quality of the produced MIS when using different random numbers, we also ran ECL-MIS when adding 10,000 to each vertex ID and when adding 10 million to each vertex ID before using them as seeds. The resulting MIS sizes are almost identical. Even in the most extreme case, they differ by less than 0.1%. Clearly, the solution quality of ECL-MIS does not depend much on the used seeds. This is expected, as the priority assignment function uses randomization only among vertices of the same degree to break ties.

ECL-MIS stores the combined status and priority information of each vertex in a single byte (see Section 3). To test whether using only 8b degrades the solution quality, we ran two modified versions of ECL-MIS, one that uses 2B words (short integers) and another that uses 4B words (integers) to hold this information, which results in a much larger range of possible priorities. This experiment yielded almost identical set sizes that differ by no more than 0.07% on the 16 graphs. Evidently, a single byte provides a large enough range of values for recording the priorities.

Priority Assignments

This section evaluates how different priority assignments affect MIS quality. In addition to the function from Section 3, we also study ECL-MIS without randomization (i.e., the random number is always zero) with the approach used by the random-permutation algorithm (i.e., a uniformly distributed random priority) using 4B to hold the priorities and with the approach used by the random-selection algorithm (i.e., vertices are selected with probability $0.5/d(v)$). Figure 12 shows that not randomizing in ECL-MIS helps a little in some cases and hurts a little in other cases, but never by much. The average resulting set size is almost identical. This seems to indicate that the randomization is useless. However, that is not the case. Without randomization, there are many ties in priority, which necessitate the invocation of the tiebreaker code. This extra code slows down the execution significantly, as we will show in the next section.

The random-permutation approach as used by CUSP, IrGL, and Pannotia yields the smallest MIS sizes on all inputs except 2d-2e20.sym. In fact, using this approach in ECL-MIS results in set sizes that are within 0.5% of those of CUSP and Pannotia on all tested inputs. In contrast, the random-selection algorithm produces larger sets on every tested graph. On average, its sets are 8.6% larger. ECL-MIS surpasses the random-selection approach on all but one of the 16 inputs. On 2d-2e20.sym, it produces an MIS that is 0.2% smaller. In the other cases, its sets are 0.2% to 5.2% larger. On average, ECL-MIS's priority assignment function yields sets that are 2.0% larger than those of the random-selection and 10.8% larger than those of the random-permutation approach.

The ECL-MIS code with the random-permutation approach is a genuine implementation of the random-permutation parallel MIS algorithm. However, ECL-MIS with our new priority-assignment function is not, as it selects a *nonrandom* permutation. Whereas Blelloch et al. [3]

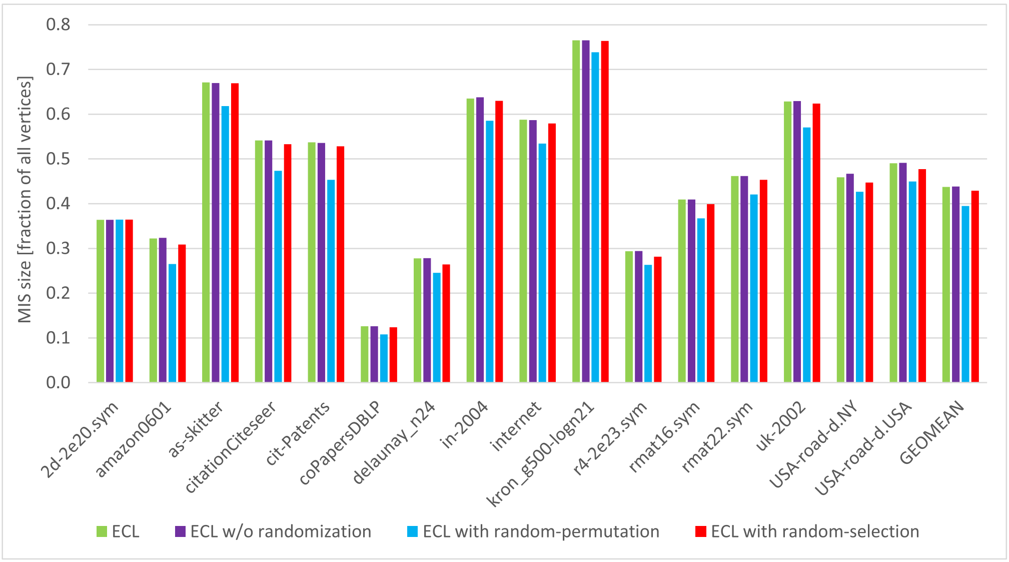

Fig. 12. MIS size as a fraction of all vertices for different priority assignment functions.

showed that most permutations require no more than O($\log^2 n$) rounds with high probability, it is not guaranteed that the specific permutation used by ECL-MIS belongs to this category.

All of the priority assignment functions discussed in this article are transitive and impose a directed acyclic graph (DAG) upon the graphs by essentially turning their undirected edges into directed edges. This DAG represents the partial order in which to process the vertices. In particular, vertices at the same level are independent and can be processed in parallel. Hence, the maximum depth of the DAG indicates how many rounds are needed. Similarly, the average depth of the DAG (the sum of the depths of all vertices divided by the number of vertices) provides an indication of how long a thread will be busy on average and is inversely proportional to the average amount of parallelism per level.

Table 5 shows the DAG depths of the 16 graphs that we studied. The first three data columns refer to the maximum depth; the last three refer to the average depth. Within each set of three columns, the first column lists the results with ECL-MIS's priority-assignment function, the second column with the random-permutation algorithm, and the third column shows the ratio of ECL-MIS's depth over that of the random-permutation approach.

In the worst case (coPapersDBLP), ECL-MIS produces a DAG that is twice as deep. In the best case (in 2004), it results in a DAG that is 30% shallower. On average, it yields a DAG that is 14% deeper. Hence, ECL-MIS's priority-assignment function tends to deepen the DAG a little, meaning that more rounds are needed. However, because ECL-MIS operates asynchronously, these extra rounds may not impact performance because faster threads (or warps) can perform more rounds in the same time as slower threads execute just a few rounds.

Interestingly, on almost all tested graphs, ECL-MIS's priority-assignment function substantially shortens the average depth. In the best case (kron_g500-logn21), the DAG is 53 times shorter, on average. Note that this is the input on which ECL-MIS outperforms the other codes by the largest margin. The geometric mean shows that the average DAG depth is about one-quarter that of the random-permutation approach, which may be another reason for ECL-MIS's higher performance. A shorter average depth implies a wider average width of the DAG, which is tantamount to more

Table 5. Depth of the DAG Imposed on the Graphs by the Priority Assignment

	maximum DAG depth			**average DAG depth**		
graph name	**ECL-MIS**	**rand perm**	**ECL/rand**	**ECL-MIS**	**rand perm**	**ECL/rand**
2d-2e20.sym	15	15	1.0	3.3	3.3	1.0
amazon0601	56	62	0.9	5.3	15.0	0.4
as-skitter	476	390	1.2	3.8	99.0	0.0
citationCiteseer	69	79	0.9	4.0	16.5	0.2
cit-Patents	136	114	1.2	5.4	16.3	0.3
coPapersDBLP	794	401	2.0	40.2	126.6	0.3
delaunay n24	22	17	1.3	3.8	4.3	0.9
in-2004	502	653	0.8	3.8	31.9	0.1
Internet	33	26	1.3	2.0	3.4	0.6
kron_g500-logn21	4024	4451	0.9	22.9	1211.9	0.0
r4-2e23.sym	30	24	1.3	5.9	6.4	0.9
rmat16.sym	188	148	1.3	9.6	41.6	0.2
rmat22.sym	643	473	1.4	10.3	123.6	0.1
uk-2002	944	944	1.0	5.2	78.0	0.1
USA-road-d.NY	12	10	1.2	2.4	2.5	1.0
USA-road-d.USA	16	14	1.1	2.2	2.3	0.9
GEOMEAN	123.35	108.59	1.14	5.45	20.94	0.26

parallelism. Hence, ECL-MIS's approach yields four times as much parallelism, on average, as the random-permutation algorithm on our graphs.

Code Transformations

This section studies how various code transformations affect performance. For improved readability, we show only the geometric mean over all inputs. Figure 13 displays the throughputs of different versions of ECL-MIS. For reference, the first bar for both GPUs gives the performance of the default ECL-MIS code and the last bar the performance of Pannotia, the fastest of the three other tested GPU codes. The ECL-2B bars show the throughput when using 2B words for expressing the combined vertex status and priority and the ECL-4B bars when using 4B words. The ECL-no-rand bars list the throughput when disabling randomization. The remaining bars show the effect of the optimizations discussed in Section 3. The ECL-sync bars show the throughput when using a synchronous implementation with multiple explicit rounds, the ECL-all-neigh bars when determining the highest-priority neighbor rather than just searching for the first neighbor with a higher priority, and the ECL-2-arrays bars when keeping the status information and the priority (random number) in separate byte arrays.

The default ECL-MIS code yields the highest throughput. Whereas disabling randomization does not affect the result quality, as discussed earlier, it does lower the average throughput by 9.1% on the K40 and 7.9% on the Titan X. This is because the tiebreaker code has to be invoked more often. Using larger word sizes, which provide a wider range of possible priorities, also does not affect the result quality but reduces the throughput. Using 2B words lowers the throughput by 8.6% and 13.7% and using 4B words lowers the throughput by 18.9% and 31.6% on the K40 and the Titan X, respectively. This is because the larger word sizes increase the memory footprint, which decreases locality and reduces the transfer efficiency (fewer words are transferred per bus transaction).

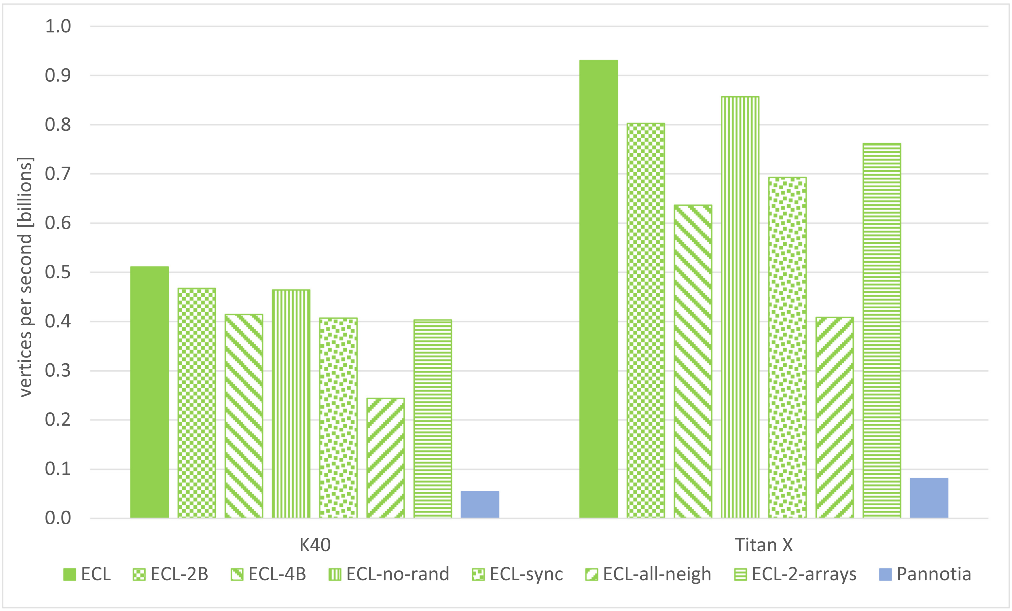

Fig. 13. Geometric-mean throughput of different code versions.

Forcing ECL-MIS to run synchronously can hurt performance drastically. For example, on the internet input, the Titan X throughput drops by 61.3%. The average drop is 20.4% and 25.5% on the K40 and the Titan X. Using separate arrays to hold the status and priority information lowers the throughput by 21.1% and 18.1%. Nevertheless, the most important optimization is to not visit all neighbors but to look only for the first neighbor with a higher priority. Always visiting all neighbors yields a throughput drop of 52.2% and 56.1%, on average, that is, the performance is less than half. The low-degree road maps are affected the least because visiting all neighbors does not take long when there are never more than a few neighbors. Finally, the comparison with the throughput of Pannotia shows that ECL-MIS's higher performance is not simply the result of a single optimization but rather the combination of multiple optimizations working together.

CPU Comparison

This section compares ECL-MIS running on the Titan X GPU to Ligra's, Ligra+'s, and PBBS's parallel CPU codes running with 40 threads on a hyperthreaded dual 10-core Xeon E5-2687W v3 system. Figure 14 shows the throughput results.

Except on coPapersDBLP, where PBBS is 25% faster, ECL-MIS yields a higher throughput on the tested graphs than the CPU codes do. On average, it is 71.7, 103.1, 4.1, and 17.4 times faster than Ligra, Ligra+, PBBS, and the serial version of PBBS, respectively. Note that the serial version of PBBS is, on average, faster than Ligra/Ligra+ because of the early-out optimization. ECL-MIS outperforms Ligra by up to 6200-fold on r4-2e23.sym, Ligra+ by up to 8100-fold also on r4-2e23.sym, parallel PBBS by up to 11-fold on internet, and serial PBBS by up to 48-fold on citationCiteseer. It is also faster than the other three parallel versions of PBBS (not shown) by up to 300-fold as well as on every tested graph, including coPapersDBLP. Note that, on average, the CPU-based parallel PBBS code is substantially faster than the GPU-based CUSP, IrGL, and Pannotia codes. Hence, the optimizations presented in this article and included in ECL-MIS are essential to making the Titan X and the K40 outperform the two Xeon sockets in our system on MIS computations.

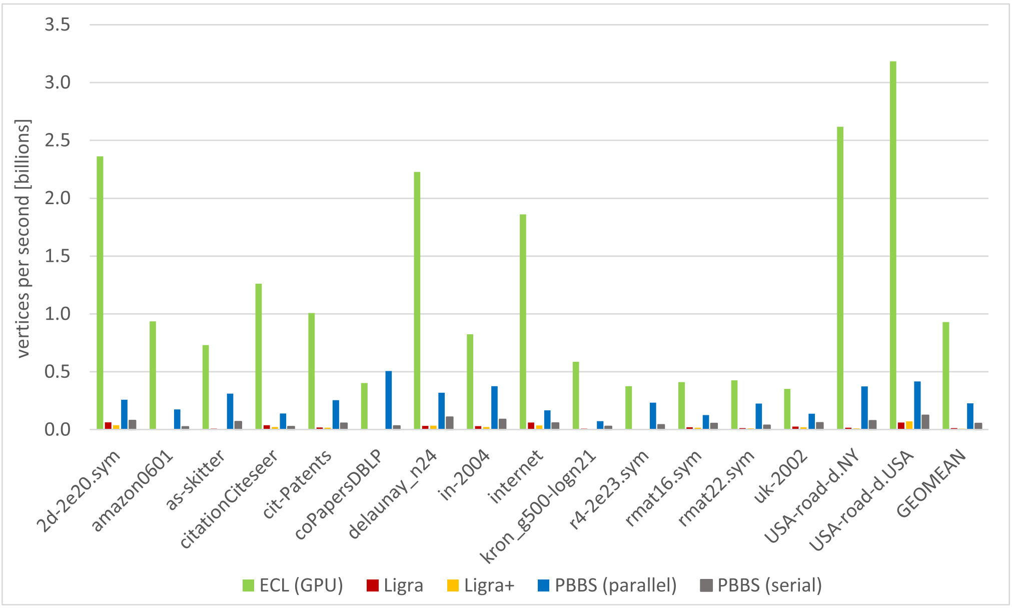

Fig. 14. Throughput in billions of CVPS on the Titan X GPU (ECL-MIS) and 20 Xeon CPU cores (other codes).

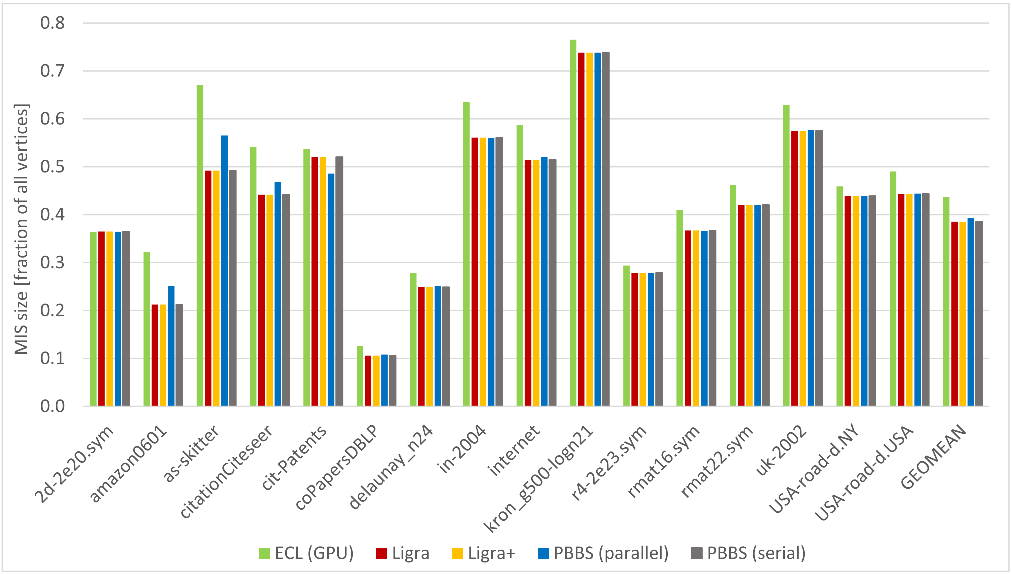

Fig. 15. MIS size as a fraction of all vertices for ECL-MIS and the CPU codes.

Figure 15 compares ECL-MIS to the CPU codes in terms of set size. As before, the sizes are listed as a fraction of the total number of vertices. Except for PBBS, the set sizes are deterministic and the same on all systems for a given graph.

Again, ECL-MIS does not produce the largest MIS on 2d-2e20.sym, where serial PBBS's set is 0.2% larger. In all other cases, ECL-MIS yields the largest sets. On average, they are 12% larger than those of Ligra, Ligra+, and serial PBBS and 10% larger than those of parallel PBBS. ECL-MIS's sets

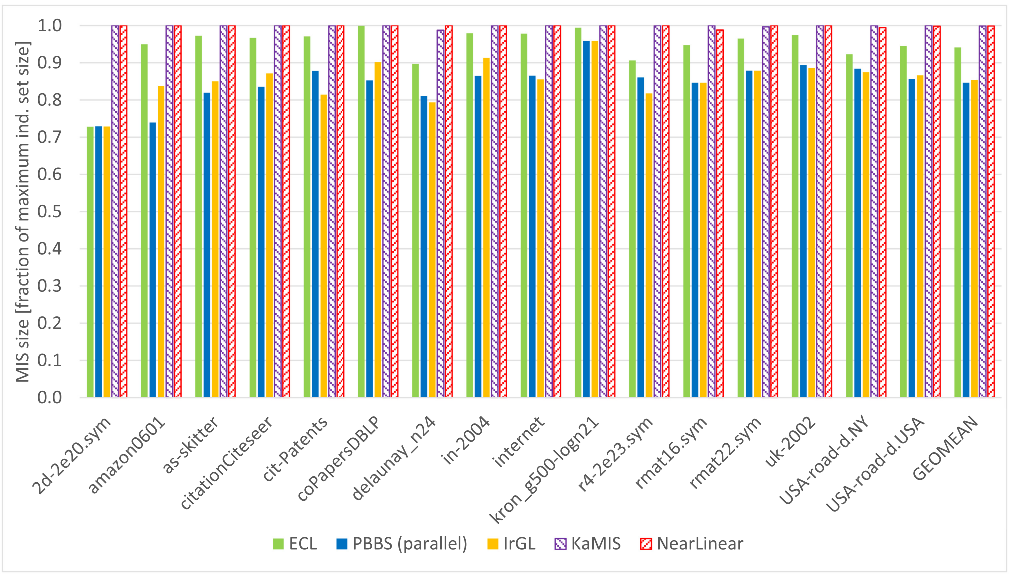

Fig. 16. MIS size as a fraction of the approximate MuIS size.

are up to 52% larger than Ligra's, Ligra+'s, and serial PBBS's sets and up to 28% larger than parallel PBBS's sets. In all cases, this maximum difference is reached on amazon0601. The nondeterministic algorithm used by parallel PBBS sometimes helps (e.g., on as-skitter) and sometimes hurts (e.g., on cit-Patents) compared to its serial counterpart. The set sizes of the serial PBBS code are on par with those of the two Ligra codes, CUSP, IrGL, and Pannotia, indicating that they all probably use similar priority-assignment functions.

Comparison to Maximum Independent Sets

This section evaluates how close the set sizes of the GPU and CPU MIS codes are to the largest possible maximal independent set, that is, the *maximum* independent set (MuIS). To do so, we used the KaMIS [19] and NearLinear [26] codes with their default configuration and selected the NearLinear algorithm for the latter. Both codes are able to compute the MuIS either precisely or approximate it closely. Figure 16 presents the results, which are normalized such that the largest set size is 1.0. To improve the readability, we show results only for IrGL, the best of the tested GPU codes, PBBS, the best of the tested CPU codes, and our ECL-MIS code in addition to KaMIS and NearLinear. Note that KaMIS produces a 1.2% larger set than NearLinear on rmat16.sym and that NearLinear produces a 1.3% larger set than KaMIS on delaunay_n24. However, in most cases, their set sizes are within 0.1% of each other. They are identical on 5 of the 16 graphs.

The figure shows that, in the worst case (2d-2e20.sym), the sets produced by the three MIS codes contain only 72.8% as many vertices as the MuIS. PBBS and IrGL perform best on kron_g500-logn21, where they reach 95.9% of the MuIS size. ECL-MIS performs best on coPapersDBLP, where it reaches 99.9% of the MuIS size. On average, PBBS's sets are 15.4% smaller than the MuIS and IrGL's are 14.6% smaller. In contrast, ECL-MIS's sets are only 5.9% smaller. In other words, ECL-MIS's priority assignment function is able to close nearly two-thirds of the gap in set size between commonly used parallel MIS algorithms and MuIS algorithms while also running faster than the other tested MIS implementations.

Table 6. Absolute Runtimes (in Seconds) of KaMIS and NearLinear Executing on the CPU and ECL-MIS Executing on the Titan X GPU as well as the Resulting Speedups

graph name	**KaMIS CPU time (s)**	**NearLinear CPU time (s)**	**ECL-MIS GPU time (s)**	**speedup over KaMIS**	**speedup over NearLinear**
2d-2e20.sym	272.4	30.7	0.00044	613,552	69,165
amazon0601	54.7	4.4	0.00043	126,881	10,114
as-skitter	119.2	4.6	0.00232	51,346	1,967
citationCiteseer	0.6	0.7	0.00021	2,969	3,442
cit-Patents	764.7	126.5	0.00375	204,201	33,771
coPapersDBLP	4.1	4.1	0.00134	3,058	3,044
delaunay n24	3,731.0	621.1	0.00753	495,412	82,469
in-2004	56.3	3.4	0.00168	33,559	2,032
internet	7.2	0.3	0.00007	107,828	4,741
kron_g500-logn21	22.6	17.2	0.00358	6,314	4,811
r4-2e23.sym	18,028.6	819.8	0.02240	804,740	36,594
rmat16.sym	604.1	2.3	0.00016	3,775,750	14,674
rmat22.sym	9,483.1	693.3	0.00986	961,871	70,325
uk-2002	666.3	126.7	0.05278	12,623	2,401
USA-road-d.NY	21.9	1.7	0.00010	217,233	16,507
USA-road-d.USA	48.4	256,936.2	0.00752	6,430	34,157,957
GEOMEAN	132.3	28.5	0.00161	82,174	17,728

For reference, the absolute runtimes of KaMIS, NearLinear, and ECL-MIS—as well as the speedups of ECL-MIS over the other two codes—are listed in Table 6. Note that the KaMIS and NearLinear runtimes are from serial CPU executions whereas ECL-MIS' runtimes are from parallel GPU executions. Moreover, KaMIS and NearLinear are primarily designed to maximize the set size. On average (geometric mean), ECL-MIS is over 82,000 times faster than KaMIS and almost 18,000 times faster than NearLinear. In the worst case (as-skitter), it is still about 2000 times faster and in the best case (USA-road-d.USA), it is over 34 million times faster. These numbers give an indication of how expensive it is to compute an MuIS compared to an MIS, the latter of which is, on average, less than 6% smaller.

6 SUMMARY AND FUTURE WORK

Computing an MIS is an important step in many parallel algorithms and can also be used to dynamically parallelize some complex codes. This article presents ECL-MIS, an MIS implementation that produces larger sets in most cases (10%, on average) and is consistently faster (11.5 times, on average, on a Titan X) than the MIS codes from CUSP, IrGL, and Pannotia on a wide variety of graphs. ECL-MIS is GPU friendly, as it operates asynchronously and thus avoids not only synchronization overhead but also PCI transfers between each round. Its open-source CUDA implementation is available at http://cs.txstate.edu/~burtscher/research/ECL-MIS/. ECL-MIS also outperforms the parallel CPU codes Ligra, Ligra+, and PBBS running on 20 Xeon cores both in terms of throughput (4.1 times, on average) and set size (10%, on average). Note that this 10% improvement in set size is substantial, as it reduces the gap to the largest possible set, the *maximum* independent set, from about 15% to under 6%, leaving relatively little room for future improvements.

ECL-MIS includes several key performance optimizations. For instance, during the main computation, it does not visit all neighbors of a vertex but stops as soon as a neighbor with a higher priority has been found. This optimization is especially important for high-degree graphs. Another important optimization is that ECL-MIS combines the priority and status information in a single byte per vertex. This not only reduces the memory footprint but also reduces the runtime because fewer memory accesses and fewer comparisons have to be performed. ECL-MIS incorporates an optimization to improve the quality of the produced solution. It does this by giving lower-degree vertices a higher priority, which tends to increase the set size. The priority assignment still employs randomization to boost the performance by avoiding ties.

These performance and quality optimizations are orthogonal to each other. They can easily be applied to other MIS implementations separately or in combination. In particular, the memory reduction optimization and the priority assignment for boosting the set size are likely also useful in other parallel MIS implementations, such as for Xeon Phis or multicore CPUs and probably even for serial code.

Studying the benefit of these optimizations on non-GPU devices is future work. Moreover, we would like to combine the idea of compressing the graph data structure, as is done in Ligra+, with our optimization for minimizing the auxiliary data to further reduce the memory footprint and hopefully improve performance. Looking forward, we hope to be able to speed up other important graph algorithms and create (deterministic) asynchronous implementations thereof, which is especially critical because the future will undoubtedly bring devices with even higher degrees of parallelism whose performance is likely to be limited by the amount of synchronization performed.

ACKNOWLEDGMENTS

We are very grateful to Sreepathi Pai for providing us with the IrGL code, to Duane Merrill for suggesting the use of correlations, and to the anonymous reviewers whose suggestions greatly helped improve this article.

REFERENCES

[1] Mark F. Adams. 1998. A parallel maximal independent set algorithm. In *Proceedings of the 5th Copper Mountain Conference on Iterative Methods*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.8968.

[2] Takuya Akiba and Yoichi Iwata. 2016. Branch-and-reduce exponential/FPT algorithms in practice: a case study of vertex cover. *In Theoretical Computer Science* 609, 1 (2016), 211–225. DOI: https://doi.org/10.1016/j.tcs.2015.09.023

[3] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 308–317. DOI: http://dx.doi.org/10.1145/2312005.2312058

[4] Lijun Chang, Wei Li, and Wenjie Zhang. 2017. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 1181–1196. DOI: https://doi.org/10.1145/3035918.3035939

[5] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, Kevin Skadron. 2013. Pannotia; Understanding irregular GPGPU graph applications. In *IEEE International Symposium on Workload Characterization (IISWC'13)*. IEEE, Portland, OR, 185–195. DOI: http://dx.doi.org/10.1109/IISWC.2013.6704684

[6] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. 2006. A survey of parallelization techniques for multigrid solvers. *Parallel Processing for Scientific Computing* 20 (2006), 179–201.

[7] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. 2016. Accelerating local search for the maximum independent set problem. In *Experimental Algorithms (SEA'16). Lecture Notes in Computer Science*, Vol. 9685, A. Goldberg, A. Kulikov (Eds.). Springer.

[8] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. CUSP: Generic parallel algorithms for sparse matrix and graph computations. Version 0.5.0. http://cusplibrary.github.io/.

[9] Camil Demetrescu. 2010. DIMACS9 (June 2010). Retrieved June 6, 2017 from http://www.dis.uniroma1.it/challenge9/download.shtml.

[10] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. 1988. *Solving Problems on Concurrent Processors, General Techniques and Regular Problems*, Vol. 1. Prentice-Hall, Inc., Upper Saddle, NJ.

[11] Mohsen Ghaffari. 2016. An improved distributed algorithm for maximal independent set. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'16)*, Robert Kraughgamer (Ed.). SIAM, Philadelphia, PA, 270–277.

[12] Git Hub. 2016. Ligra. Retrieved November 15, 2018 from https://github.com/jshun/ligra.

[13] Git Hub. 2017. Pannotia. Retrieved November 15, 2018 from https://github.com/pannotia/pannotia.

[14] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar'12)*. San Jose, CA, 1–14. DOI: 10.1109/InPar.2012.6339596

[15] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering Heuristics for Parallel Graph Coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. ACM, New York, NY, 166–177. DOI: 10.1145/2612669.2612697 http://doi.acm.org/10.1145/2612669.2612697

[16] Benoît Hudson, Gary L. Miller, and Todd Phillips. 2007. Sparse parallel Delaunay mesh refinement. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*. ACM, New York, NY, 339–347. DOI: http://dx.doi.org/10.1145/1248377.1248435

[17] ISS. 2014. Galois. Retrieved November 15, 2018 from http://iss.ices.utexas.edu/?p=projects/galois/download.

[18] Yan Jin and Jin-Kao Hao. 2015. General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Engineering Applications of Artificial Intelligence* 37 (2015), 20–33.

[19] KaMIS. 2017. Retrieved November 15, 2018 from http://algo2.iti.kit.edu/kamis/.

[20] Richard M. Karp and Avi Wigderson. 1984. A fast parallel algorithm for the maximal independent set problem. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC'84)*. ACM, New York, NY, 266–272. DOI: http://dx.doi.org/10.1145/800057.808690

[21] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. 2017. Finding near-optimal independent sets at scale. *Journal of Heuristics* 23, 4 (2017), 207–229. DOI: https://doi.org/10.1007/s10732-017-9337-x

[22] Michael Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC'85)*. ACM, New York, NY, 1–10. DOI: http://dx.doi.org/10.1145/22145.22146

[23] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing* 15, 4 (1986), 1036–1053.

[24] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. 2010. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, 3–14. DOI: http://dx.doi.org/10.1145/1693453.1693457

[25] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*, John Cavazos, Xiang Gong, and David Kaeli (Eds.). ACM, New York, NY, 96–107. DOI: http://dx.doi.org/10.1145/2458523.2458533

[26] NearLinear. Retrieved November 15, 2018 from https://github.com/LijunChang/Near-Maximum-Independent-Set.

[27] NVIDIA. 2017. CUSP. Retrieved November 15, 2018 from https://developer.nvidia.com/cusp.

[28] NVIDIA. 2017. THRUST. Retrieved November 15, 2018 from https://developer.nvidia.com/thrust.

[29] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. *SIGPLAN Not.* 51, 10 (2016), 1–19. DOI: https://doi.org/10.1145/3022671.2984015

[30] PBBS. 2014. Retrieved November 15, 2018 from http://www.cs.cmu.edu/~pbbs/.

[31] Jonathan R. Shewchuk. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications* 22, 1-3 (2002), 21–74. DOI: http://10.1016/S0925-7721(01)00047-5

[32] SNAP. Retrieved November 15, 2018 from https://snap.stanford.edu/data/.

[33] Sparse Matrix Collection. Retrieved November 15, 2018 from https://www.cise.ufl.edu/research/sparse/matrices/.

[34] TEPS. Retrieved November 15, 2018 from http://www.graph500.org/specifications#sec-8_2.

[35] Leslie G. Valiant. 1982. Parallel computation. In *Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science*. 171–189.

Received November 2017; revised March 2018; accepted March 2018